

Security Research

PASR

Preparatory Action on the enhancement of the European industrial potential in the field of Security research



Grant Agreement no. SEC5-PR-104600
ROBIN

Open Robust Infrastructures

Project

Deliverable D.13

Machine-Checked Verification Appendix

Due date of deliverable: 31/04/2008
Actual submission date: 15/05/2008

Start date of Activity: 31/01/2006

Duration: 31/04/2008

Organisation name of lead beneficiary for this deliverable: Radboud Universiteit Nijmegen

Revision 1

CLASSIFICATION: Public

15/05/2008

**Appendices with PVS source code and
proofs for the document**

Nova Micro-Hypervisor Verification

B PVS Theory Sources

Contents for Appendix B

B.1	abstract_data.pvs	64
B.2	allocators.pvs	67
B.3	bits.pvs	83
B.4	challenge-linear.pvs	94
B.5	challenge-phymem.pvs	115
B.6	challenge-ptab-sync-master.pvs	119
B.7	constants.pvs	120
B.8	conversions.pvs	124
B.9	cpp-examples.pvs	131
B.10	cpp-verification.pvs	142
B.11	datatype_model.pvs	142
B.12	device_memory.pvs	147
B.13	everything.pvs	155
B.14	expressions.pvs	155
B.15	graph.pvs	184
B.16	hoare.pvs	192
B.17	linear_memory.pvs	194
B.18	memory.pvs	203
B.19	paging-data-models.pvs	225
B.20	paging-data.pvs	239
B.21	physical_memory.pvs	249
B.22	plain_memory.pvs	252
B.23	plain_memory_rewrites.pvs	272
B.24	ptab-sync-master-defs.pvs	288
B.25	ptab-sync-master.pvs	290
B.26	random_device.pvs	294
B.27	result.pvs	300
B.28	search-example.pvs	306
B.29	statement-rewrites.pvs	314
B.30	statements.pvs	336
B.31	state-transformer.pvs	351
B.32	types.pvs	370
B.33	viasco-prelude.pvs	434

B.1 abstract_data.pvs

Uninterpreted_Data : **Theory**

Begin

Importing IA32

Uninterpreted_data_type : **Type** =

```
[#
  size : nat,
  valid? : [list[Byte], Address -> bool]
#]
```

uninterpreted_data_type? : PRED[Uninterpreted_data_type] =
Lambda(uidt : Uninterpreted_data_type) :

```
(Forall(l : list[Byte], a : Address) :
  not length(l) = size(uidt) Implies
  valid?(uidt)(l,a) = False)
```

% proof status :-)

u_data_type_length : **Lemma**

```
Forall(uidt : (uninterpreted_data_type?), l : list[Byte], a : Address) :
  valid?(uidt)(l,a) Implies length(l) = size(uidt)
```

End Uninterpreted_Data

Interpreted_Data[Data : **Type**] : **Theory**

Begin

IMPORTING Uninterpreted_Data, Address_Util, Bitlist_ops

Interpreted_data_type : **Type** =

```
[#
  uidt : Uninterpreted_data_type, % Assumptions:
  to_byte : [Data, Address -> list[Byte]], % to_byte is zero at unmodified bits
  to_mask : [Data, Address -> list[Byte]], % to_mask is one at unmodified bits
  from_byte : [list[Byte], Address -> lift[Data]]
#]
% Thus, new_value = (read & ~to_mask) | to_byte
% and from_byte must be reasonable on &= to_mask values
```

% proof status :-)

% interpreted_data_type?_TCC1 : TCC Obligation

interpreted_data_type? : PRED[Interpreted_data_type] =
Lambda(idt : Interpreted_data_type) :

```
uninterpreted_data_type?(idt'uidt) AND
  % result of to_byte is valid (and has length size)
(Forall(d : Data, a : Address) :
  valid?(uidt(idt))(to_byte(idt)(d,a), a)) And
```

```

% from_byte fails on invalid stuff
(Forall(l : list[Byte], a : Address) :
  valid?(uidt(idt))(l, a) IFF
  up?(from_byte(idt)(l, a))) And

% from_byte is the left inverse of to_byte
% 1. from_byte does not fail on to_byte results
% this follows from the previous two
%
% (Forall(d : Data, a : Address) :
% up?(from_byte(idt)(to_byte(idt)(d,a), a))) AND

% 2. the result is the same
(Forall(d : Data, a : Address) :
  down(from_byte(idt)(to_byte(idt)(d,a), a)) = d)

pod_data_type? : PRED[Interpreted_data_type] =
Lambda(idt : Interpreted_data_type) :
  interpreted_data_type?(idt) AND
  (Forall(l : list[Byte], a1, a2 : Address) :
    valid?(uidt(idt))(l, a1) IFF valid?(uidt(idt))(l, a2)) And
  size(uidt(idt)) > 0

% subtype judgements

% the first two are intuitively right, but are not accepted because
% of the subrecord structure

% interpreted_data_is_uninterpreted_data : Judgement
% (interpreted_data_type?) SUBTYPE_OF (uninterpreted_data_type?)

% pod_id_uninterpreted_data : Judgement
% (pod_data_type?) SUBTYPE_OF (uninterpreted_data_type?)

% proof status :-)
pod_is_interpreted_data : Judgement
  (pod_data_type?) SUBTYPE_OF (interpreted_data_type?)

% proof status :-)
pod_size : Lemma
  Forall (dt : (pod_data_type?)) : size(uidt(dt)) > 0

% Utility predicates

idt : Var (interpreted_data_type?)
addr : Var Address
min : Var Memory_Address
max : Var Memory_Address

in_memory?(idt, addr, min, max) : bool =
  in_memory(min, max)(addr) And in_memory(min, max)(addr + size(uidt(idt)))

```

```

in_blessed_memory?(idt, addr, (address_range : PRED[Address])) : bool =
  subset?(address_block(addr, size(uidt(idt))), address_range)

% proof status :-)
data_type_length : Lemma Forall(l : list[Byte]) :
  valid?(uidt(idt))(l,addr) Implies length(l) = size(uidt(idt))

% proof status :-)
length_to_byte : Lemma Forall(data : Data) :
  length(to_byte(idt)(data, addr)) = size(uidt(idt))

% proof status :-)
valid_from_byte : Lemma Forall(l : list[Byte]) :
  idt'uidt'valid?(l, addr) Implies up?(from_byte(idt)(l, addr))

% proof status :-)
valid_iff_from_byte : Lemma Forall(l : list[Byte]) :
  idt'uidt'valid?(l, addr) <=> up?(from_byte(idt)(l, addr))

% proof status :-)
valid_to_byte : Lemma Forall(data : Data) :
  idt'uidt'valid?(to_byte(idt)(data, addr), addr)

% proof status :-)
up_from_byte_to_byte : Lemma Forall(data : Data) :
  up?(from_byte(idt)(to_byte(idt)(data, addr), addr))

% proof status :-)
from_byte_to_byte : Lemma Forall(data : Data) :
  from_byte(idt)(to_byte(idt)(data, addr), addr) = up(data)

% proof status :-)
in_blessed_memory_subset : Lemma
Forall (address_range : PRED[Address]) :
  in_blessed_memory?(idt, addr, address_range) Implies
  subset?(address_block(addr, size(uidt(idt))), address_range)

```

End Interpreted_Data

Interpreted_Data_Lift[Data1, Data2 : Type] : Theory

Begin

Importing Interpreted_Data

```

dt_lift(dt : Interpreted_data_type[Data1],
  ext : [Data2 -> Data1],
  inj : [Data1 -> Data2]) : Interpreted_data_type[Data2] =
  (# uidt := uidt(dt),
  to_byte := Lambda (d2 : Data2, a : Address) : to_byte(dt)(ext(d2), a),
  to_mask := Lambda (d2 : Data2, a : Address) : to_mask(dt)(ext(d2), a),

```

```

from_byte := Lambda (l : list[Byte], a : Address) :
  Cases from_byte(dt)(l, a) Of
    up(d1) : up(inj(d1)),
    bottom : bottom
  EndCases

```

```
#)
```

```
End Interpreted_Data_Lift
```

```
Abstract_Read_Write[State, Data : Type] : Theory
```

```
Begin
```

```
  IMPORTING Memory_Change[State]
```

```
  IMPORTING State_Transformer_Lift, Interpreted_Data
```

```
  dt : Var (interpreted_data_type?[Data]) % dt aka Data_type
```

```
  addr : Var Address
```

```
  s : Var State % s aka state
```

```
  data : Var Data
```

```
  pm : Var Memory_struct[State]
```

```
  valid_in_mem(pm,dt)(addr)(s) : bool =
```

```
    Cases memory_read_list(pm)(addr, size(uidt(dt)))(s) Of
```

```
      OK(sn, bl) : dt'uidt'valid?(bl, addr)
```

```
      Else true
```

```
    EndCases
```

```
  read_data(pm,dt)(addr) : [State -> ExprResult[State, Data]] =
```

```
    (memory_read_list(pm)(addr, size(uidt(dt))) ##
```

```
      Lambda(bl : list[Byte]) : ok_lift(from_byte(dt)(bl, addr)))
```

```
  write_data(pm,dt)(addr, data) : [State -> ExprResult[State, Unit]] =
```

```
    memory_write_list(pm)(addr, to_byte(dt)(data, addr))
```

```
  % proof status :-)
```

```
  read_data_valid_in_mem : Lemma
```

```
    OK?(read_data(pm, dt)(addr)(s)) Implies valid_in_mem(pm, dt)(addr)(s)
```

```
End Abstract_Read_Write
```

B.2 allocators.pvs

```
Allocator[State : Type] : Theory
```

```
% Specification of memory allocators working on plain memory,
```

```
% such as the stack allocator and the one for global variables.
```

```
Begin
```

```
  Importing Plain_Memory[State]
```

```
  % The allocator interface consists of two real methods and four
```

```
  % logical ones.
```

```
  % alloc allocates memory
```

B PVS Theory Sources

```
% free frees memory
%
% memory_pool The memory from which the allocator allocates. This should
% be disjoint from all other live allocators.
% allocated Gives the set of addresses that have been allocated by this
% allocator. Used to get the disjointness of correctly
% allocated variables.
% private_mem Returns those memory regions that must not be changed
% for this allocator to work properly.
% freed_size Returns the size of the block that will be freed by
% a free in the same state. Needed to require a correct
% adjustment of the allocated predicate during free.

% We exclude the possibility to allocate 0 Bytes here (this is
% permitted by many malloc libraries). We further require that free
% makes a visible change in the allocated predicate. Some malloc
% libraries (or was it the C++ standard?) do not guarantee that freed
% memory is really freed. We only require that the predicate is changed
% (and not that freed memory can be allocated again), nevertheless, this
% excludes empty free functions (such as in conservative garbage
% collectors).
%
% One would expect a valid predicate, which is turned on by init and
% maintained afterwards. This valid predicate would somehow classify
% the set of states in which the allocator is expected to work.
% However, the real methods alloc and free
% have always the possibility to return fatal. The valid predicate would
% only clutter the specification, therefore we keep it implicit.
% The set of states in which the allocator is expected to work is
% given by the states predicate of the involved plain memory.

Allocator : Type = [#
  memory_pool : [(plain_memory?) -> [State -> PRED[Address]]],
  allocated : [(plain_memory?) -> [State -> PRED[Address]]],
  private_mem : [(plain_memory?) -> PRED[Address]],
  freed_size : [(plain_memory?) -> [Address -> [State -> lift[posnat]]]],

  % init: [(plain_memory?) -> [State -> ExprResult[State, Unit]]],
  alloc: [(plain_memory?) ->
    [posnat -> [State -> ExprResult[State, Address]]],
  free: [(plain_memory?) -> [Address -> [State -> ExprResult[State, Unit]]]]
#]

% proof status :-)
% null_address_TCC1 : TCC Obligation

% HT: This should be NULL for an arbitrary pointer type, coming from
% the pointer semantics
null_address : Memory_Address = Mem(0)

% Required properties of allocators:
```

```

% – allocated is a subset of the memory pool
% – Newly allocated memory is disjoint and comes from the memory pool.
% – If allocate and free succeed the allocated predicate is adjusted
% accordingly, while the memory pool stays constant.
% – Free succeeds precisely when freed_size does.
% – Reading memory and writing memory at non-private addresses
% does not change the allocated predicate and the memory pool

% proof status :-)
% allocator?_TCC1 : TCC Obligation

% proof status :-)
% allocator?_TCC2 : TCC Obligation

% proof status :-)
% allocator?_TCC3 : TCC Obligation

% proof status :-)
% allocator?_TCC4 : TCC Obligation

% proof status :-)
% allocator?_TCC5 : TCC Obligation

% proof status :-)
% allocator?_TCC6 : TCC Obligation

% proof status :-)
% allocator?_TCC7 : TCC Obligation

% proof status :-)
% allocator?_TCC8 : TCC Obligation

allocator?(pm : (plain_memory?)) : PRED[Allocator] = Lambda(ac : Allocator) :

  % private memory is in plain memory -- therefore the
  % allocator won't change under read access
  subset?(ac'private_mem(pm), union(pm'ro_addr, pm'rw_addr))
  And

  Forall(s : State) : pm'states(s) Implies
    % allocate from the memory pool
    subset?(ac'allocated(pm)(s), ac'memory_pool(pm)(s)) And

    % alloc properties
    (Forall(size : posnat) :
      OK?(ac'alloc(pm)(size)(s)) And
      Not data(ac'alloc(pm)(size)(s)) = null_address Implies
        disjoint?(ac'allocated(pm)(s),
          address_block(data(ac'alloc(pm)(size)(s)), size))
      And
      ac'allocated(pm)(state(ac'alloc(pm)(size)(s))) =
        union(ac'allocated(pm)(s),

```

```

        address_block(data(ac'alloc(pm)(size)(s)), size)
    And
    ac'memory_pool(pm)(state(ac'alloc(pm)(size)(s))) =
        ac'memory_pool(pm)(s)
And
% connection of free <--> freed_size
(Forall(addr : Address) :
    OK?(ac'free(pm)(addr)(s)) IFF up?(ac'freed_size(pm)(addr)(s)))
And
% free properties
(Forall(addr : Address) :
    OK?(ac'free(pm)(addr)(s)) Implies
        ac'allocated(pm)(state(ac'free(pm)(addr)(s))) =
            difference(ac'allocated(pm)(s),
                address_block(addr, down(ac'freed_size(pm)(addr)(s))))
    And
    ac'memory_pool(pm)(state(ac'free(pm)(addr)(s))) =
        ac'memory_pool(pm)(s)
And
% The private memory contains the state of the allocator. If
% it stays constant, then the allocator is unchanged.
(Forall(s2 : State) :
    pm'states(s2) And
    stays_unchanged(pm)(s, s2, ac'private_mem(pm))
    Implies
        ac'memory_pool(pm)(s) = ac'memory_pool(pm)(s2) And
        ac'allocated(pm)(s) = ac'allocated(pm)(s2))
And
% next_state invariants
%(has_next_state(ac'init(pm)(s)) Implies pm'states(state(ac'init(pm)(s))))
%And
(Forall(size : posnat) :
    has_next_state(ac'alloc(pm)(size)(s)) Implies
        pm'states(state(ac'alloc(pm)(size)(s)))
    And
(Forall(addr : Address) :
    has_next_state(ac'free(pm)(addr)(s)) Implies
        pm'states(state(ac'free(pm)(addr)(s))))
And
% alloc and free only change private memory, more precisely,
% only writable private memory
(Forall(size : posnat) :
    OK?(ac'alloc(pm)(size)(s)) And
    NOT data(ac'alloc(pm)(size)(s)) = null_address
    Implies
        only_changes(pm)(s, state(ac'alloc(pm)(size)(s)),
            ac'private_mem(pm)))

```

And

```
(Forall(free_addr : Address) :
  OK?(ac'free(pm)(free_addr)(s)) Implies
  only_changes(pm)(s, state(ac'free(pm)(free_addr)(s)),
    ac'private_mem(pm)))
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% private memory is inside plain memory
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
% proof status :-)
allocator_subset_private_memory_plain_memory : Lemma
Forall(pm : (plain_memory?), ac : (allocator?(pm))) :
  subset?(ac'private_mem(pm), union(pm'ro_addr, pm'rw_addr))
```

```
% proof status :-)
allocator_private_memory_in_plain_memory : Lemma
Forall(pm : (plain_memory?), ac : (allocator?(pm)), addr : Address) :
  ac'private_mem(pm)(addr) Implies
  union(pm'ro_addr, pm'rw_addr)(addr)
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% allocate from the memory pool
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
% proof status :-)
allocator_subset_allocated_memory_pool : Lemma
Forall(pm : (plain_memory?), ac : (allocator?(pm)), s : State) :
  pm'states(s) Implies
  subset?(ac'allocated(pm)(s), ac'memory_pool(pm)(s))
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% alloc properties
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
% proof status :-)
allocator_alloc_disjoint : Lemma
Forall(pm : (plain_memory?), ac : (allocator?(pm)),
  size : posnat, s : State) :
  pm'states(s) And
  OK?(ac'alloc(pm)(size)(s)) And
  Not data(ac'alloc(pm)(size)(s)) = null_address Implies
  disjoint?(ac'allocated(pm)(s),
```



```

        address_block(data(ac'alloc(pm)(size)(s)), size))

% proof status :-)
% allocator_allocated_alloc_TCC1 : TCC Obligation

% proof status :-)
allocator_allocated_alloc : Lemma
  Forall(pm : (plain_memory?), ac : (allocator?(pm)),
          size : posnat, s : State) :
    pm'states(s) And
    OK?(ac'alloc(pm)(size)(s)) And
    Not data(ac'alloc(pm)(size)(s)) = null_address Implies
      ac'allocated(pm)(state(ac'alloc(pm)(size)(s))) =
        union(ac'allocated(pm)(s),
              address_block(data(ac'alloc(pm)(size)(s)), size))

% proof status :-)
allocator_memory_pool_alloc : Lemma
  Forall(pm : (plain_memory?), ac : (allocator?(pm)),
          size : posnat, s : State) :
    pm'states(s) And
    OK?(ac'alloc(pm)(size)(s)) And
    Not data(ac'alloc(pm)(size)(s)) = null_address Implies
      ac'memory_pool(pm)(state(ac'alloc(pm)(size)(s))) =
        ac'memory_pool(pm)(s)

% proof status :-)
allocator_subset_address_block_alloc_allocated : Lemma
  Forall(pm : (plain_memory?), ac : (allocator?(pm)),
          size : posnat, addr : Address, s : State) :
    pm'states(s) And
    OK?(ac'alloc(pm)(size)(s)) And
    NOT data(ac'alloc(pm)(size)(s)) = null_address Implies
      subset?(address_block(data(ac'alloc(pm)(size)(s)), size),
              ac'allocated(pm)(state(ac'alloc(pm)(size)(s))))

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% connection of free <--> freed_size
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% proof status :-)
allocator_free_freed_size : Lemma
  Forall(pm : (plain_memory?), ac : (allocator?(pm)),
          addr : Address, s : State) :
    pm'states(s) Implies
      (OK?(ac'free(pm)(addr)(s)) IFF up?(ac'freed_size(pm)(addr)(s)))

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

```

```

%
% free properties
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% proof status :-)
% allocator_allocated_free_TCC1 : TCC Obligation

% proof status :-)
% allocator_allocated_free_TCC2 : TCC Obligation

% proof status :-)
allocator_allocated_free : Lemma
  Forall(pm : (plain_memory?), ac : (allocator?(pm)),
          addr : Address, s : State) :
    pm'states(s) And
    OK?(ac'free(pm)(addr)(s)) Implies
    ac'allocated(pm)(state(ac'free(pm)(addr)(s))) =
    difference(ac'allocated(pm)(s),
    address_block(addr, down(ac'freed_size(pm)(addr)(s))))

% proof status :-)
allocator_memory_pool_free : Lemma
  Forall(pm : (plain_memory?), ac : (allocator?(pm)),
          addr : Address, s : State) :
    pm'states(s) And
    OK?(ac'free(pm)(addr)(s)) Implies
    ac'memory_pool(pm)(state(ac'free(pm)(addr)(s))) =
    ac'memory_pool(pm)(s)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% constant private memory implies memory_pool and allocated are constant
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% proof status :-)
allocator_const_private_memory_memory_pool : Lemma
  Forall(pm : (plain_memory?), ac : (allocator?(pm)), s1, s2 : State) :
    pm'states(s1) And
    pm'states(s2) And
    stays_unchanged(pm)(s1, s2, ac'private_mem(pm))
    Implies
    ac'memory_pool(pm)(s1) = ac'memory_pool(pm)(s2)

% proof status :-)
allocator_const_private_memory_allocated : Lemma
  Forall(pm : (plain_memory?), ac : (allocator?(pm)), s1, s2 : State) :
    pm'states(s1) And
    pm'states(s2) And
    stays_unchanged(pm)(s1, s2, ac'private_mem(pm))

```

B PVS Theory Sources

Implies

ac'allocated(pm)(s1) = ac'allocated(pm)(s2)

```

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% next_state invariants
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% proof status :-)
% allocator_next_state_alloc_TCC1 : TCC Obligation

% proof status :-)
allocator_next_state_alloc : Lemma
  Forall(pm : (plain_memory?), ac : (allocator?(pm)),
          size : posnat, s : State) :
    pm'states(s) And
    has_next_state(ac'alloc(pm)(size)(s)) Implies
    pm'states(state(ac'alloc(pm)(size)(s)))

% proof status :-)
% allocator_next_state_free_TCC1 : TCC Obligation

% proof status :-)
allocator_next_state_free : Lemma
  Forall(pm : (plain_memory?), ac : (allocator?(pm)),
          addr : Address, s : State) :
    pm'states(s) And
    has_next_state(ac'free(pm)(addr)(s)) Implies
    pm'states(state(ac'free(pm)(addr)(s)))

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% alloc and free only change private memory
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% proof status :-)
% allocator_only_changes_alloc_TCC1 : TCC Obligation

% proof status :-)
allocator_only_changes_alloc : Lemma
  Forall(pm : (plain_memory?), ac : (allocator?(pm)),
          size : posnat, s : State) :
    pm'states(s) And
    OK?(ac'alloc(pm)(size)(s)) And
    NOT data(ac'alloc(pm)(size)(s)) = null_address
    Implies
    only_changes(pm)(s, state(ac'alloc(pm)(size)(s)),
                ac'private_mem(pm))

```

```

% proof status :-)
% allocator_only_changes_free_TCC1 : TCC Obligation

% proof status :-)
allocator_only_changes_free : Lemma
  Forall(pm : (plain_memory?), ac : (allocator?(pm)),
          free_addr : Address, s : State) :
    pm'states(s) And
    OK?(ac'free(pm)(free_addr)(s)) Implies
      only_changes(pm)(s, state(ac'free(pm)(free_addr)(s)),
                    ac'private_mem(pm))

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% memory_read invariant
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% proof status :-)
% allocator_allocated_memory_read_TCC1 : TCC Obligation

% proof status :-)
allocator_allocated_memory_read : Lemma
  Forall(pm : (plain_memory?), ac : (allocator?(pm)),
          addr : Address, s : State) :
    pm'states(s) And
    union(pm'ro_addr, pm'rw_addr)(addr) Implies
      ac'allocated(pm)(state(memory_read(pm'mem)(addr)(s))) =
      ac'allocated(pm)(s)

% proof status :-)
allocator_memory_pool_memory_read : Lemma
  Forall(pm : (plain_memory?), ac : (allocator?(pm)),
          addr : Address, s : State) :
    pm'states(s) And
    union(pm'ro_addr, pm'rw_addr)(addr) Implies
      ac'memory_pool(pm)(state(memory_read(pm'mem)(addr)(s))) =
      ac'memory_pool(pm)(s)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% memory_write invariant
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% proof status :-)
% allocator_stays_unchanged_memory_write_private_mem_TCC1 : TCC Obligation

% proof status :-)
% allocator_stays_unchanged_memory_write_private_mem_TCC2 : TCC Obligation

```

B PVS Theory Sources

```
% proof status :-)
allocator_stays_unchanged_memory_write_private_mem : Lemma
  Forall(pm : (plain_memory?), ac : (allocator?(pm)),
          addr : Address, byte : Byte, s : State) :
    pm'states(s) And
    pm'rw_addr(addr) And
    Not ac'private_mem(pm)(addr)
    Implies
      stays_unchanged(pm)(s, state(memory_write(pm'mem)(addr, byte)(s)),
                      ac'private_mem(pm))

% proof status :-)
allocator_allocated_memory_write : Lemma
  Forall(pm : (plain_memory?), ac : (allocator?(pm)),
          addr : Address, byte : Byte, s : State) :
    pm'states(s) And
    pm'rw_addr(addr) And
    Not ac'private_mem(pm)(addr) Implies
      ac'allocated(pm)(state(memory_write(pm'mem)(addr, byte)(s))) =
      ac'allocated(pm)(s)

% proof status :-)
allocator_memory_pool_memory_write : Lemma
  Forall(pm : (plain_memory?), ac : (allocator?(pm)),
          addr : Address, byte : Byte, s : State) :
    pm'states(s) And
    pm'rw_addr(addr) And
    Not ac'private_mem(pm)(addr) Implies
      ac'memory_pool(pm)(state(memory_write(pm'mem)(addr, byte)(s))) =
      ac'memory_pool(pm)(s)

%%%%%%%%%%%%%%
%
% more complex properties
%
%%%%%%%%%%%%%%

% proof status :-)
allocator_subset_address_block_alloc_memory_pool : Lemma
  Forall(pm : (plain_memory?), ac : (allocator?(pm)),
          size : posnat, addr : Address, s : State) :
    pm'states(s) And
    OK?(ac'alloc(pm)(size)(s)) And
    NOT data(ac'alloc(pm)(size)(s)) = null_address Implies
      subset?(address_block(data(ac'alloc(pm)(size)(s)), size),
              ac'memory_pool(pm)(s))
```

End Allocator

```

Allocation_Info[State : Type] : Theory
% This theory formalizes the (additional) information that
% we track for allocators and its consistency condition.
Begin
  Importing Allocator[State], More_Relations

  % The allocation info contains the set of allocation points of one
  % specific allocator. An allocation point is an (address, size) pair
  % that has been allocated with this very allocator (that is, the
  % address was returned by an alloc with argument size in some former
  % state and no free has been done so far.
  %
  % Note that the consistency below requires only that the allocation
  % info contains a subset of the allocated memory. In the following
  % cases correctly allocated memory must not be in the allocation info:
  %
  % – Memory allocated in allocator A that is used as memory pool of
  % allocator B.

  Allocator_Info : Type = list[[Address, posnat]]

  allocator_info? : PRED[Allocator_Info] = Lambda(ai : Allocator_Info) :
    blocks_pairwise_disjoint(ai)

  pm : Var (plain_memory?)
  ai : Var Allocator_Info
  s : Var State

  consistent_allocator?(pm)(ac : (allocator?(pm)), ai)(s) : bool =
    allocator_info?(ai) And
    Forall(addr : Address, size : posnat) : member((addr, size), ai) Implies
      subset?(address_block(addr, size), ac'allocated(pm)(s))

  % proof status :-
  consistent_allocator_state_change : Lemma
    Forall(ac : (allocator?(pm)), s1, s2 : State) :
      consistent_allocator?(pm)(ac, ai)(s1) And
      ac'allocated(pm)(s1) = ac'allocated(pm)(s2) Implies
        consistent_allocator?(pm)(ac, ai)(s2)

  % proof status :-
  % consistent_allocator_read_TCC1 : TCC Obligation

  % proof status :-
  consistent_allocator_read : Lemma
    Forall(ac : (allocator?(pm)), addr : Address) :
      pm'states(s) And
      consistent_allocator?(pm)(ac, ai)(s) And
      union(pm'ro_addr, pm'rw_addr)(addr) Implies
        consistent_allocator?(pm)(ac, ai)(state(memory_read(pm'mem)(addr)(s)))

```

B PVS Theory Sources

```
% proof status :-)
% consistent_allocator_write_TCC1 : TCC Obligation

% proof status :-)
consistent_allocator_write : Lemma
  Forall(ac : (allocator?(pm)), addr : Address, byte : Byte) :
    pm'states(s) And
    consistent_allocator?(pm)(ac, ai)(s) And
    pm'rw_addr(addr) And
    Not ac'private_mem(pm)(addr) Implies
      consistent_allocator?(pm)(ac, ai)
        (state(memory_write(pm'mem)(addr, byte)(s)))

% proof status :-)
% consistent_allocator_alloc_TCC1 : TCC Obligation

% proof status :-)
consistent_allocator_alloc : Lemma
  Forall(ac : (allocator?(pm)), size : posnat) :
    pm'states(s) And
    consistent_allocator?(pm)(ac, ai)(s) And
    OK?(ac'alloc(pm)(size)(s)) And
    Not data(ac'alloc(pm)(size)(s)) = null_address Implies
      consistent_allocator?(pm)
        (ac, cons((data(ac'alloc(pm)(size)(s)), size), ai))
          (state(ac'alloc(pm)(size)(s)))

% proof status :-)
% consistent_allocator_free_TCC1 : TCC Obligation

% proof status :-)
% consistent_allocator_free_TCC2 : TCC Obligation

% proof status :-)
consistent_allocator_free : Lemma
  Forall(ac : (allocator?(pm)), addr : Address) :
    pm'states(s) And
    consistent_allocator?(pm)(ac, ai)(s) And
    OK?(ac'free(pm)(addr)(s)) And
    (member((addr, down(ac'freed_size(pm)(addr)(s))), ai) OR
      blocks_pairwise_disjoint(
        cons((addr, down(ac'freed_size(pm)(addr)(s))), ai)))
    Implies
      consistent_allocator?(pm)
        (ac, list_remove((addr, down(ac'freed_size(pm)(addr)(s))), ai))
          (state(ac'free(pm)(addr)(s)))
```

End Allocation_Info

Allocation_Table[State : **Type**] : **Theory**

Begin

Importing Allocation_Info[State], Graph, Finite_Set_Reduce,
More_List_Props_2

pm : **Var** (plain_memory?)

*% An allocation table contains a hierarchy of allocators together
% with their allocation info's. The hierarchy is a finite, directed graph.
% hierarchy(ac1, ac2) means that the memory pool of ac2 is allocated
% inside ac1. Currently the hierarchy must be a tree, that is a
% non-root allocator gets its memory from precisely one allocator.
% Thus Fiascos region manager is currently unsupported.*

Allocation_Table(pm) : **Type** = [#
 hierarchy : (finite_tree?[(allocator?(pm))]),
 info : [(hierarchy'nodes) -> Allocator_Info]
#]

ai : **Var** Allocator_Info
s : **Var** State

allocation_ok?(pm)(at : Allocation_Table(pm), s) : bool =
 (**Forall**(ac1, ac2 : (at'hierarchy'nodes)) :
 at'hierarchy'edges(ac1, ac2) **Implies**
 subset?(ac2'memory_pool(pm)(s), ac1'allocated(pm)(s)) **And**
 block_is_free(ac2'memory_pool(pm)(s), at'info(ac1))
)
 And
 (**Forall**(ac1, ac2, ac3 : (at'hierarchy'nodes)) :
 at'hierarchy'edges(ac1, ac2) **And**
 at'hierarchy'edges(ac1, ac3) **Implies**
 ac2 = ac3 **OR**
 disjoint?(ac2'memory_pool(pm)(s), ac3'memory_pool(pm)(s))
)
 And
 (**Forall**(ac1, ac2 : (roots(at'hierarchy))) :
 ac1 = ac2 **OR** disjoint?(ac1'memory_pool(pm)(s), ac2'memory_pool(pm)(s)))
 And
 (**Forall**(ac1, ac2 : (at'hierarchy'nodes)) :
 ac1 = ac2 **OR** disjoint?(ac1'private_mem(pm), ac2'private_mem(pm)))
 And
 (**Forall**(ac : (at'hierarchy'nodes)) :
 consistent_allocator?(pm)(ac, at'info(ac))(s))

%%
%
% allocation access lemmas
%
%%

% proof status :-)

allocation_finite_tree : **Lemma**
 Forall(at : Allocation_Table(pm)) :


```

finite_tree?(at'hierarchy)

% proof status :-)
allocation_subnodes_allocated_and_free : Lemma
Forall(at : Allocation_Table(pm), ac1, ac2 : (at'hierarchy'nodes)) :
  allocation_ok?(pm)(at, s) And
  at'hierarchy'edges(ac1, ac2) Implies
  subset?(ac2'memory_pool(pm)(s), ac1'allocated(pm)(s)) And
  block_is_free(ac2'memory_pool(pm)(s), at'info(ac1))

% proof status :-)
allocation_subnodes_disjoint : Lemma
Forall(at : Allocation_Table(pm), ac1, ac2, ac3 : (at'hierarchy'nodes)) :
  allocation_ok?(pm)(at, s) And
  at'hierarchy'edges(ac1, ac2) And
  at'hierarchy'edges(ac1, ac3) Implies
  ac2 = ac3 OR
  disjoint?(ac2'memory_pool(pm)(s), ac3'memory_pool(pm)(s))

% proof status :-)
allocation_roots_disjoint : Lemma
Forall(at : Allocation_Table(pm), ac1, ac2 : (roots(at'hierarchy))) :
  allocation_ok?(pm)(at, s) Implies
  ac1 = ac2 OR
  disjoint?(ac1'memory_pool(pm)(s), ac2'memory_pool(pm)(s))

% proof status :-)
allocation_private_memory_disjoint : Lemma
Forall(at : Allocation_Table(pm), ac1, ac2 : (at'hierarchy'nodes)) :
  allocation_ok?(pm)(at, s) Implies
  ac1 = ac2 OR
  disjoint?(ac1'private_mem(pm), ac2'private_mem(pm))

% proof status :-)
allocation_consistent_allocator : Lemma
Forall(at : Allocation_Table(pm), ac : (at'hierarchy'nodes)) :
  allocation_ok?(pm)(at, s) Implies
  consistent_allocator?(pm)(ac, at'info(ac))(s)

% proof status :-)
allocation_allocated_allocation_point : Lemma
Forall(at : Allocation_Table(pm), ac : (at'hierarchy'nodes),
  addr : Address, size : posnat) :
  allocation_ok?(pm)(at, s) And
  member((addr, size), at'info(ac)) Implies
  subset?(address_block(addr, size), ac'allocated(pm)(s))

% proof status :-)
allocation_allocation_point_in_memory_pool : Lemma
Forall(at : Allocation_Table(pm), ac : (at'hierarchy'nodes),
  addr : Address, size : posnat) :
  pm'states(s) And

```

```

allocation_ok?(pm)(at, s) And
member((addr, size), at'info(ac)) Implies
  subset?(address_block(addr, size), ac'memory_pool(pm)(s))

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% general utility lemmas
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% proof status :-)
allocators_different_in_hierachy : Lemma
  Forall(at : Allocation_Table(pm), ac1, ac2 : (at'hierarchy'nodes)) :
    at'hierarchy'edges(ac1, ac2) Implies
      Not ac1 = ac2

% proof status :-)
uppath_memory_pool : Lemma
  Forall(at : Allocation_Table(pm), p : (path?(at'hierarchy))) :
    allocation_ok?(pm)(at, s) And
    pm'states(s) Implies
      subset?(path_end(p)'memory_pool(pm)(s),
              path_start(p)'memory_pool(pm)(s))

% proof status :-)
% uppath_memory_pool_allocated_and_free_TCC1 : TCC Obligation

% proof status :-)
uppath_memory_pool_allocated_and_free : Lemma
  Forall(at : Allocation_Table(pm), p : (path?(at'hierarchy))) :
    pm'states(s) And
    allocation_ok?(pm)(at, s) And
    length(p) >= 2 Implies
      subset?(path_end(p)'memory_pool(pm)(s), path_start(p)'allocated(pm)(s))
      And
      block_is_free(path_end(p)'memory_pool(pm)(s), at'info(path_start(p)))

% proof status :-)
allocation_memory_pools : Lemma
  Forall(at : Allocation_Table(pm), ac1, ac2 : (at'hierarchy'nodes)) :
    pm'states(s) And
    allocation_ok?(pm)(at, s) Implies
      ac1 = ac2
      OR
      disjoint?(ac1'memory_pool(pm)(s), ac2'memory_pool(pm)(s))
      OR
      (subset?(ac1'memory_pool(pm)(s), ac2'allocated(pm)(s)) And
       block_is_free(ac1'memory_pool(pm)(s), at'info(ac2)))
      OR
      (subset?(ac2'memory_pool(pm)(s), ac1'allocated(pm)(s)) And

```

B PVS Theory Sources

```

    block_is_free(ac2'memory_pool(pm)(s), at'info(ac1)))

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% general allocation results
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% proof status :-)
% allocation_info_TCC1 : TCC Obligation

allocation_info(pm)(at : Allocation_Table(pm)) : Allocator_Info =
  flatten(map(at'info)(list_of_finite_set(at'hierarchy'nodes)))

% proof status :-)
member_allocation_info : Lemma
  Forall(at : Allocation_Table(pm), addr : Address, size : posnat) :
    member((addr, size), allocation_info(pm)(at)) Implies
      Exists(ac : (at'hierarchy'nodes)) : member((addr, size), at'info(ac))

% all allocation points are disjoint
% proof status :-)
allocation_consistent : Lemma
  Forall(at : Allocation_Table(pm)) :
    pm'states(s) And
    allocation_ok?(pm)(at, s) Implies
      allocator_info?(allocation_info(pm)(at))

% proof status :-)
% allocation_alloc_other_private_memory_TCC1 : TCC Obligation

% proof status :-)
% allocation_alloc_other_private_memory_TCC2 : TCC Obligation

% proof status :-)
allocation_alloc_other_private_memory : Lemma
  Forall(at : Allocation_Table(pm),
    ac1, ac2 : (at'hierarchy'nodes), size : posnat) :
    pm'states(s) And
    allocation_ok?(pm)(at, s) And
    OK?(ac1'alloc(pm)(size)(s)) And
    Not data(ac1'alloc(pm)(size)(s)) = null_address And
    Not ac1 = ac2 Implies
      stays_unchanged(pm)(s, state(ac1'alloc(pm)(size)(s)),
        ac2'private_mem(pm))

% proof status :-)
% allocation_alloc_TCC1 : TCC Obligation

% proof status :-)
allocation_alloc : Lemma

```

```

Forall(at : Allocation_Table(pm),
        ac : (at'hierarchy'nodes), size : posnat) :
  pm'states(s) And
  allocation_ok?(pm)(at, s) And
  OK?(ac'alloc(pm)(size)(s)) And
  Not data(ac'alloc(pm)(size)(s)) = null_address Implies
    allocation_ok?(pm)(
      at WITH ['info(ac) :=
        cons((data(ac'alloc(pm)(size)(s)), size), at'info(ac))],
        state(ac'alloc(pm)(size)(s)))

```

End Allocation_Table

B.3 bits.pvs

Bits : **Theory**

Begin

Importing Alignment

% MV: do we need the size limitats any more

n, i : **Var** nat

bool_to_nat(b : bool) : nat = **if** b **then** 1 **else** 0 **endif**

% proof status :-)

bool_to_nat_below : Judgement bool_to_nat(b : bool) HAS_TYPE below(2)

nat_to_bool(n : below(2)) : bool = n = 1

% proof status :-)

nat_bool_iso : **Lemma Forall**(b : bool) : nat_to_bool(bool_to_nat(b)) = b

% return true if the i-th bit in n is 1

cut_bit(n, i) : bool = rem(2)(ndiv(n, expt(2, i))) = 1

% proof status :-)

cut_bit_zero : **Lemma** cut_bit(0, i) = **False**

% cut k bits out of n, starting at i

cut_bits(n, i, k : nat) : nat =
rem(expt(2, k))(ndiv(n, expt(2, i)))

% proof status :-)

cut_bits_below : Judgement cut_bits(n, i, k : nat)
HAS_TYPE below(expt(2, k))

% proof status :-)

cut_bits_aligned : **Lemma Forall**(n, i, k : nat) :
i <= k **And** cut_bits(n, 0, k) = 0 **Implies** aligned?(i)(n)

% proof status :-)

```

aligned_cut_bits : Lemma Forall(n, i, j : nat, k : nat) :
  i + j <= k And aligned?(k)(n) Implies
    cut_bits(n, i, j) = 0

% proof status :-)
aligned_add_cut_bits : Lemma Forall(n, m, i, j, k : nat) :
  i + j <= k And aligned?(k)(n) Implies
    cut_bits(n + m, i, j) = cut_bits(m, i, j)

% proof status :-)
cut_bit_bits : Lemma
  cut_bit(n, i) = IF cut_bits(n, i, 1) = 1 Then true Else false Endif

% proof status :-)
cut_bits_zero : Lemma Forall(i, k : nat) : cut_bits(0, i, k) = 0

% proof status :-)
zero_cut_bits : Lemma Forall(n, i, k : nat) : k = 0 Implies
  cut_bits(n, i, k) = 0

% proof status :-)
% cut_bits_cut_bits_TCC1 : TCC Obligation

% proof status :-)
cut_bits_cut_bits : Lemma
  Forall(n, i1, k1, i2, k2 : nat) :
    cut_bits(cut_bits(n, i1, k1), i2, k2) =
      IF i2 >= k1 Then 0
      Elseif i2 + k2 > k1 Then cut_bits(n, i1 + i2, k1 - i2)
      Else cut_bits(n, i1 + i2, k2)
      Endif

% proof status :-)
cut_bit_cut_bits : Lemma
  Forall(n, i1, k1, i2 : nat) :
    cut_bit(cut_bits(n, i1, k1), i2) =
      IF i2 >= k1 Then False
      Else cut_bit(n, i1 + i2)
      Endif

shift_bits_left(n, i) : nat = n * expt(2, i)

% proof status :-)
% shift_bits_left_less_TCC1 : TCC Obligation

% proof status :-)
shift_bits_left_less : Lemma Forall(n, i, k : nat) :
  k >= i And n < expt(2, k - i) Implies
    shift_bits_left(n, i) < expt(2, k)

% proof status :-)

```

```

aligned_shift_bits_ok : Lemma Forall(n, i, j : nat) :
  i <= j Implies
    aligned?(i)(shift_bits_left(n, j))

% proof status :-)
% aligned_shift_bits_reduce_TCC1 : TCC Obligation

% proof status :-)
aligned_shift_bits_reduce : Lemma Forall(n, i, j : nat) :
  i > j And aligned?(i - j)(n) Implies
    aligned?(i)(shift_bits_left(n, j))

shift_bits_right(n, i) : nat = ndiv(n, expt(2, i))

% proof status :-)
% overwrite_bits_TCC1 : TCC Obligation

% overwrite_bits(n, m, i, j)
% replaces k bits starting from i in word n
% with k bits starting at i in word m
overwrite_bits(n, m, i, k : nat) : nat =
  n - rem(expt(2, i + k))(n)
  + shift_bits_left(cut_bits(m, i, k), i)
  + rem(expt(2, i))(n)

% proof status :-)
overwrite_bits_less : Lemma Forall(n, m, i, k, j : nat) :
  n < expt(2, j) And i + k <= j Implies
    overwrite_bits(n, m, i, k) < expt(2, j)

overwrite_bit(n, m, i : nat) : nat = overwrite_bits(n, m, i, 1)

% proof status :-)
overwrite_bit_less : Lemma Forall(n, m, i, k : nat) :
  n < expt(2, k) And i < k Implies
    overwrite_bit(n, m, i) < expt(2, k)

% overwrites bit i in n with b
overwrite_bool_bit(n : nat, b : bool, i : nat) : nat =
  overwrite_bit(n, shift_bits_left(bool_to_nat(b), i), i)

% proof status :-)
overwrite_bool_bit_less : Lemma Forall(n : nat, b : bool, i, k : nat) :
  n < expt(2, k) And i < k Implies
    overwrite_bool_bit(n, b, i) < expt(2, k)

% overwrite k bits in n at index i with the first k bits from m
overwrite_shift_bits(n, m, i : nat, k : nat) : nat =
  overwrite_bits(n, shift_bits_left(m, i), i, k)

```

B PVS Theory Sources

```

% proof status :-)
overwrite_shift_bits_less : Lemma Forall(n, m, i, k, j : nat) :
  n < expt(2, j) And i + k <= j Implies
  overwrite_shift_bits(n, m, i, k) < expt(2, j)

% proof status :-)
aligned_overwrite_shift_bits_more : Lemma
Forall(a, n, m, i : nat, k : posnat) :
  a <= i And aligned?(a)(n) Implies
  aligned?(a)(overwrite_shift_bits(n, m, i, k))

% proof status :-)
% aligned_overwrite_shift_bits_less_TCC1 : TCC Obligation

% the aligned(a)(n) is a very strong assumptions,
% closer to necessity would be
% aligned(a)(delete bits i to i + k - 1 in n)
% proof status :-)
aligned_overwrite_shift_bits_less : Lemma
Forall(a, n, m, i : nat, k : posnat) :
  i < a And aligned?(a)(n) And aligned?(a - i)(m) Implies
  aligned?(a)(overwrite_shift_bits(n, m, i, k))

% proof status :-)
% cut_bits_shift_bits_left_TCC1 : TCC Obligation

% proof status :-)
% cut_bits_shift_bits_left_TCC2 : TCC Obligation

% Don't install cut_bits_shift_bits_left as auto rewrite,
% it can be circular for ci = 0. Use the lemmas below instead.
% proof status :-)
cut_bits_shift_bits_left : Lemma Forall(n, si, ci, ck : nat) :
  cut_bits(shift_bits_left(n, si), ci, ck) =
  If ci + ck <= si Then 0
  Elsif ci < si
  Then cut_bits(shift_bits_left(n, si - ci), 0, ck)
  Else cut_bits(n, ci - si, ck)
  Endif

% proof status :-)
cut_bits_shift_bits_left_outside : Lemma
Forall(n, si, ci, ck : nat) :
  ci + ck <= si Implies
  cut_bits(shift_bits_left(n, si), ci, ck) = 0

% proof status :-)
% cut_bits_shift_bits_left_overlap_TCC1 : TCC Obligation

```

```

% proof status :-)
cut_bits_shift_bits_left_overlap : Lemma
  Forall(n, si, ci, ck : nat) :
    ci + ck > si And ci < si And ci > 0 Implies
      cut_bits(shift_bits_left(n, si), ci, ck) =
        cut_bits(shift_bits_left(n, si - ci), 0, ck)

% proof status :-)
cut_bits_shift_bits_left_in : Lemma
  Forall(n, si, ci, ck : nat) :
    ci >= si Implies
      cut_bits(shift_bits_left(n, si), ci, ck) =
        cut_bits(n, ci - si, ck)

% proof status :-)
overwrite_shift_bits_zero : Lemma
  Forall(z, m, i, k : nat) :
    z = 0 Implies
      overwrite_shift_bits(z, m, i, k) =
        shift_bits_left(cut_bits(m, 0, k), i)

% proof status :-)
zero_overwrite_shift_bits : Lemma
  Forall(n, m, i, k : nat) : k = 0 Implies
    overwrite_shift_bits(n, m, i, k) = n

% proof status :-)
% bit_split_TCC1 : TCC Obligation

% proof status :-)
bit_split : Lemma Forall(n, i, k : nat) :
  k <= i And n < expt(2, i) Implies
    n = overwrite_shift_bits(cut_bits(n, 0, k),
                             cut_bits(n, k, i - k), k, i - k)

% proof status :-)
shift_bits_left_cut_bits : Lemma Forall(n, i, k, j : nat) :
  i = j And aligned?(i)(n) And n < expt(2, i + k) Implies
    shift_bits_left(cut_bits(n, i, k), j) = n

% proof status :-)
cut_bits_0_to_below : Lemma Forall(n, k : nat) :
  n < expt(2, k) Implies cut_bits(n, 0, k) = n

% proof status :-)
cut_bits_overwrite_bits_disjoined : Lemma
  Forall(n, m, oi, ok : nat, ci, ck : nat) :
    oi + ok <= ci Or ci + ck <= oi Implies
      cut_bits(overwrite_bits(n, m, oi, ok), ci, ck) = cut_bits(n, ci, ck)

% proof status :-)

```


B PVS Theory Sources

```

cut_bits_overwrite_bool_bit_disjoined : Lemma
  Forall(n : nat, b : bool, oi, ci, ck : nat) :
    oi < ci Or ci + ck <= oi Implies
      cut_bits(overwrite_bool_bit(n, b, oi), ci, ck) = cut_bits(n, ci, ck)

% proof status :-)
cut_bits_overwrite_bits_contained : Lemma
  Forall(n, m, oi, ok, ci, ck : nat) :
    oi <= ci And ci + ck <= oi + ok Implies
      cut_bits(overwrite_bits(n, m, oi, ok), ci, ck) =
        cut_bits(m, ci, ck)

% proof status :-)
cut_bits_overwrite_shift_bits_disjoint : Lemma
  Forall(n, m, i1, k1, i2, k2 : nat) :
    i1 + k1 <= i2 Or i2 + k2 <= i1 Implies
      cut_bits(overwrite_shift_bits(n, m, i1, k1), i2, k2) =
        cut_bits(n, i2, k2)

% proof status :-)
% cut_bits_overwrite_shift_bits_contained_TCC1 : TCC Obligation

% proof status :-)
cut_bits_overwrite_shift_bits_contained : Lemma
  Forall(n, m, i1, k1, i2, k2 : nat) :
    i1 <= i2 And i2 + k2 <= i1 + k1 Implies
      cut_bits(overwrite_shift_bits(n, m, i1, k1), i2, k2) =
        cut_bits(m, i2 - i1, k2)

% proof status :-)
% cut_bit_overwrite_shift_bits_TCC1 : TCC Obligation

% proof status :-)
cut_bit_overwrite_shift_bits : Lemma
  Forall(n, m, i1, k1, i2 : nat) :
    cut_bit(overwrite_shift_bits(n, m, i1, k1), i2) =
      IF i1 <= i2 And i2 < i1 + k1
      Then cut_bit(m, i2 - i1)
      Else cut_bit(n, i2)
      Endif

% not needed any more, proof saved below
% cut_bits_overwrite_bool_bit_contained : Lemma
% Forall(n : nat, b : bool, oi : nat, ci, ck : nat) :
% ck = 1 And oi = ci Implies
% cut_bits(overwrite_bool_bit(n, b, oi), ci, ck) =
% bool_to_nat(b)

% proof status :-)
cut_bit_overwrite_bool_bit : Lemma
  Forall(n : nat, b : bool, i, j : nat) :

```

```

cut_bit(overwrite_bool_bit(n, b, i), j) =
  IF i = j THEN b
  ELSE cut_bit(n, j)
ENDIF

```

```

% proof status :-)

```

```

overwrite_shift_bits_merge : Lemma

```

```

Forall(m, ci1, ck1, si, ci2, ck2, os, ok : nat) :

```

```

  ci1 + ck1 = ci2 And

```

```

  si + ck1 = os And

```

```

  ck2 = ok

```

```

  Implies

```

```

    overwrite_shift_bits(shift_bits_left(cut_bits(m, ci1, ck1), si),
      cut_bits(m, ci2, ck2), os, ok) =

```

```

    shift_bits_left(cut_bits(m, ci1, ck1 + ck2), si)

```

```

End Bits

```

```

% Bit operations for below[2^p]

```

```

Bitops[p : posnat] : Theory

```

```

Begin

```

```

Importing Bits, More_List_Props

```

```

n : Var nat

```

```

val : Var nat

```

```

v1, v2 : Var nat

```

```

bit_set?(n, val) : bool = cut_bit(val, n)

```

```

% proof status :-)

```

```

bit_set_zero_mask : Lemma

```

```

Forall (n : nat) : Not bit_set?(n, 0)

```

```

% proof status :-)

```

```

bit_set_ndiv_expt2 : Lemma

```

```

Forall (n, val, m : nat) :

```

```

  bit_set?(n, ndiv(val, expt(2, m))) = bit_set?(n + m, val)

```

```

% proof status :-)

```

```

% bit_set_full_mask_TCC1 : TCC Obligation

```

```

% proof status :-)

```

```

bit_set_full_mask : Lemma

```

```

Forall (n : nat) : n < p Implies bit_set?(n, expt(2, p) - 1)

```

```

% proof status :-)

```

```

bits_equal : Lemma

```

```

Forall (v1, v2, p : nat) :

```

```

  (Forall (n : nat) : n < p Implies bit_set?(n, v1) = bit_set?(n, v2)) Implies
    rem(expt(2, p))(v1) = rem(expt(2, p))(v2)

```

```

% proof status :-)
bits_equal2 : Lemma
  Forall (v1, v2 : below[expt(2, p)]) :
    (Forall (n : nat) : n < p Implies bit_set?(n, v1) = bit_set?(n, v2)) IFF
      v1 = v2

% proof status :-)
bits_equal3 : Lemma
  Forall (v1, v2 : nat) :
    (Forall (n : nat) : bit_set?(n, v1) = bit_set?(n, v2)) Implies
      v1 = v2

% proof status :-)
bits_equal4 : Lemma
  Forall (v1, v2 : nat, m : nat) :
    (Forall (n : nat) : n >= m Implies bit_set?(n, v1) = bit_set?(n, v2)) Implies
      ndiv(v1, expt(2, m)) = ndiv(v2, expt(2, m))

set_bit(n, val) : nat =
  If bit_set?(n, val) Then
    val
  Else
    val + expt(2, n)
  Endif

% proof status :-)
% clear_bit_TCC1 : TCC Obligation

clear_bit(n, val) : nat =
  If bit_set?(n, val) Then
    val - expt(2, n)
  Else
    val
  Endif

change_bit(val : nat, c : [nat -> bool])(n : nat) : nat =
  If c(n) Then
    set_bit(n, val)
  Else
    clear_bit(n, val)
  Endif

% proof status :-)
change_bit_single_bit_op : Lemma
  Forall (n, m, v : nat, c : [nat -> bool]) : n /= m Implies
    bit_set?(n, v) = bit_set?(n, change_bit(v, c)(m))

% proof status :-)
change_bit_result : Lemma
  Forall (n, v : nat, c : [nat -> bool]) :
    bit_set?(n, change_bit(v, c)(n)) = c(n)

```

```

% proof status :-)
% change_bits_TCC1 : TCC Obligation

% proof status :-)
% change_bits_TCC2 : TCC Obligation

change_bits(v : nat, c : [nat -> bool])(n : nat) : Recursive nat =
  If n = 0 Then
    change_bit(v, c)(0)
  Else
    change_bit(change_bits(v, c)(n - 1), c)(n)
  Endif
Measure n

% proof status :-)
change_bits_result : Lemma
  Forall (m, n : nat, val : nat, c : [nat -> bool]) : m <= n Implies
    bit_set?(m, change_bits(val, c)(n)) = c(m)

% proof status :-)
change_bits_unchanged : Lemma
  Forall (m, n : nat, val : nat, c : [nat -> bool]) : m > n Implies
    bit_set?(m, change_bits(val, c)(n)) = bit_set?(m, val)

% proof status :-)
% change_bits_below_TCC1 : TCC Obligation

% proof status :-)
change_bits_below : Lemma
  Forall (v : below[expt(2, p)], c : [nat -> bool]) :
    change_bits(v, c)(p-1) < expt(2, p)

% proof status :-)
% binary_not_TCC1 : TCC Obligation

% binary ops
%-----
binary_not(val : below[expt(2, p)]) : below[expt(2, p)] =
  change_bits(val, Lambda (n : nat) : Not bit_set?(n, val))(p - 1)

% proof status :-)
binary_not_not : Lemma
  Forall ( val : below[expt(2, p)]) :
    binary_not(binary_not(val)) = val

% proof status :-)
% binary_and_TCC1 : TCC Obligation

binary_and(v1, v2 : below[expt(2, p)]) : below[expt(2, p)] =
  change_bits(v1, Lambda (n : nat) : bit_set?(n, v1) And bit_set?(n, v2))(p - 1)

```

B PVS Theory Sources

```
% proof status :-)
binary_and_not : Lemma
  Forall (val : below[expt(2, p)]) :
    binary_and(val, binary_not(val)) = 0

% proof status :-)
% binary_and_full_TCC1 : TCC Obligation

% proof status :-)
binary_and_full : Lemma
  Forall (val : below[expt(2, p)]) :
    binary_and(val, expt(2, p) - 1) = val

% proof status :-)
% binary_and_zero_TCC1 : TCC Obligation

% proof status :-)
binary_and_zero : Lemma
  Forall (val : below[expt(2, p)]) :
    binary_and(val, 0) = 0

% proof status :-)
% binary_or_TCC1 : TCC Obligation

binary_or(v1, v2 : below[expt(2, p)]) : below[expt(2, p)] =
  change_bits(v1, Lambda (n : nat) : bit_set?(n, v1) Or bit_set?(n, v2))(p - 1)

% proof status :-)
binary_or_not : Lemma
  Forall (val : below[expt(2, p)]) :
    binary_or(val, binary_not(val)) = expt(2, p) - 1

% proof status :-)
% binary_xor_TCC1 : TCC Obligation

binary_xor(v1, v2 : below[expt(2, p)]) : below[expt(2, p)] =
  change_bits(v1, Lambda (n : nat) : bit_set?(n, v1) /= bit_set?(n, v2))(p - 1)

% proof status :-)
binary_xor_self : Lemma
  Forall (val : below[expt(2, p)]) :
    binary_xor(val, val) = 0

% proof status :-)
binary_or_full : Lemma
  Forall (val : below[expt(2, p)]) :
    binary_or(val, expt(2, p) - 1) = expt(2, p) - 1

% proof status :-)
binary_or_zero : Lemma
  Forall (val : below[expt(2, p)]) :
    binary_or(val, 0) = val
```

End Bitops

Bitlist_ops : **Theory**

Begin

Importing General, Bitops[bits_per_byte]

% Lifting to lists

%=====

% proof status :-)

% binary_not_TCC1 : TCC Obligation

binary_not(l : list[Byte]) : list[Byte] =
 map[Byte, Byte](binary_not)(l)

% proof status :-)

% binary_and_TCC1 : TCC Obligation

binary_and(l1, l2 : list[Byte]) : list[Byte] =
 map2[Byte](binary_and)(l1, l2)

% proof status :-)

% binary_or_TCC1 : TCC Obligation

binary_or(l1, l2 : list[Byte]) : list[Byte] =
 map2[Byte](binary_or)(l1, l2)

% proof status :-)

% binary_xor_TCC1 : TCC Obligation

binary_xor(l1, l2 : list[Byte]) : list[Byte] =
 map2[Byte](binary_xor)(l1, l2)

full_mask?(size : nat)(l : list[Byte]) : bool =
 length(l) = size **And** every(**Lambda** (t : Byte) : t = max_byte - 1)(l)

zero_mask?(size : nat)(l : list[Byte]) : bool =
 length(l) = size **And** every(**Lambda** (t : Byte) : t = 0)(l)

% proof status :-)

singleton_zero_mask : **Lemma**

Forall (s : nat) :

 singleton?(zero_mask?(s))

% proof status :-)

% bit_set?_TCC1 : TCC Obligation

% proof status :-)

% bit_set?_TCC2 : TCC Obligation

B PVS Theory Sources

```
bit_set?(n : nat, l : list[Byte]) : bool =
  If n < length(l) * max_byte Then
    bit_set?(rem(max_byte)(n), nth(l, ndiv(n, max_byte)))
  Else
    false
  Endif

apply_bitop_valuefield(mask : list[Byte], bitop : [list[Byte] -> list[Byte]])(l : list[Byte]) : list[Byte] =
  binary_or(binary_and(l, binary_not(mask)), binary_and(bitop(l), mask))

End Bitlist_ops
```

B.4 challenge-linear.pvs

```
% MV: there are a few assumptions in the address in page table range stuff
%
% Current Address Space Layout (hypervisor/include/memory.h)
%
% | Global Area | CPU Local Area | AS Loc|
% |-----|
% |-----|
% | Kcode | Mem Pool | |RM|VGA|IOAPIC| Loc| |LAPIC|SP|CPULoc| IOPBM |
% |-----|
% |-----|
%
% HV: Assumptions
% - The global area is synced at the master PDIR level (fault.cpp/paging.cpp).
% - Master PDIR is in Kcode (.bss segment)
%
% Model:
% - address_in_page_table_range gives all page table entries used for the given
% address range. Page table entries outside this range may be in plain memory
% - this should at least allow to enter the fault handler without losing plain
% memory because subset(ro_PF u rw_PF, ro u rw)
% ? is this a generic result for plain memory, i.e., pm(range) => pm(sub_range)
% and if so do we have it.
```

```
Linear_Memory_Blessing[% the underlying physical memory
  Physical_memory : Type,
  (Importing Plain_Memory[Physical_memory])
  pm_phy : Plain_Memory] : Theory
```

Begin

```
Importing Linear_Memory[Physical_memory, pm_phy],
  Memory_Change, Control_Register_Datatype

oact : Var PRED[[Linear_memory -> SuperResult[Linear_memory]]]
ro_addr : Var PRED[Memory_Address_4G]
rw_addr : Var PRED[Memory_Address_4G]
states : Var PRED[Linear_memory]

linear_range : Var PRED[Memory_Address_4G]
```

```

s : Var Linear_memory

lvl : Var Level

% Sharing
%=====

% tuples of linear addresses that share the same physical address
linear_shared?(s)(a1, a2 : Memory_Address_4G) : bool =
  Exists(ac1, ac2 : Memory_access) :
    OK?(linear_resolve(a1, ac1)(s)) And
    OK?(linear_resolve(a2, ac2)(s)) And
    data(linear_resolve(a1, ac1)(s)) =
    data(linear_resolve(a2, ac2)(s))

% Address In Page Table
%=====

% proof status :-)
% PTab_Address_TCC1 : TCC Obligation

PTab_Address : Type =
  {a : Memory_Address_4G | aligned?(bits_per_level + pe_size)(offset(a))}

% proof status :-)
% pe_in_pdir_range?_TCC1 : TCC Obligation

pe_in_pdir_range?(s, linear_range)(phy_a : Memory_Address_4G) : bool =
  Let res = read_data(pm_phy, pdir_data_type)(PDBR)(s) In
  OK?(res) And
  Exists (lin_a : (linear_range)) :
    xlat_idx(pdir_lvl, data(res)'base_addr, lin_a) = phy_a

% proof status :-)
% pe_in_ptab_range?_TCC1 : TCC Obligation

% proof status :-)
% pe_in_ptab_range?_TCC2 : TCC Obligation

% proof status :-)
% pe_in_ptab_range?_TCC3 : TCC Obligation

% proof status :-)
% pe_in_ptab_range?_TCC4 : TCC Obligation

pe_in_ptab_range?(s, linear_range)(phy_a : Memory_Address_4G) : bool =
  Exists (pde : (pe_in_pdir_range?(s, linear_range))) :
  Exists (lin_a : (linear_range)) :
  Let res = read_data(pm_phy, paging_data_type(pdir_lvl))(pde)(s) In
  OK?(res) And pde_pt?(pdir(data(res))) And

```



```

xlat_idx(ptab_lvl, base(data(res)), lin_a) = phy_a

% proof status :-)
% pe_in_pt_range?_TCC1 : TCC Obligation

% proof status :-)
% pe_in_pt_range?_TCC2 : TCC Obligation

pe_in_pt_range?(s, linear_range, lvl)(phy_a : Memory_Address_4G) : bool =
  Cond
    lvl = pdir_lvl -> pe_in_pdir_range?(s, linear_range)(phy_a),
    lvl = ptab_lvl -> pe_in_ptab_range?(s, linear_range)(phy_a)
  EndCond

% proof status :-)
pe_in_pt_range_aligned : Lemma
  Forall (phy_a : Memory_Address_4G) :
    pe_in_pt_range?(s, linear_range, lvl)(phy_a) Implies
      aligned?(pe_size)(offset(phy_a))

address_in_pt_range?(s, linear_range)(phy_a : Memory_Address_4G) : bool =
  Exists (pe : Memory_Address_4G, lvl : Level) :
    pe_in_pt_range?(s, linear_range, lvl)(pe) And
      address_block(pe, expt(2, pe_size))(phy_a)

% proof status :-)
pe_in_pt_address_in_pt : Lemma
  Forall (base : Memory_Address_4G) :
    pe_in_pt_range?(s, linear_range, lvl)(base)
  Implies
    subset?(address_block(base, expt(2, pe_size)),
      address_in_pt_range?(s, linear_range))

% Physical Addresses
%=====

virt_to_phys_range(s : Linear_memory, linear_range : PRED[Memory_Address_4G],
  mem_ac : PRED[Memory_access])
  (phy_a : Memory_Address_4G) : bool =
  Exists (lin_a : (linear_range), ac : (mem_ac)) :
    Let res = linear_resolve(lin_a, ac)(s) In
      OK?(res) And data(res) = phy_a

virt_to_phys_range(s, linear_range) : [Memory_Address_4G -> bool] =
  virt_to_phys_range(s, linear_range, fullset[Memory_access])

% Linear Resolve Transformers
%=====

linear_resolve_register_transformers
  (q : [Physical_memory -> SuperResult[Physical_memory]]) : bool =
  q = expr_2_super(read_data(pm_phy, segment_reg_data_type)(CS)) Or

```

```

q = expr_2_super(read_data(pm_phy, pabr_data_type)(PDBR)) Or
Exists (a : Memory_Address_4G) :
  q = expr_2_super(write_data(pm_phy, address_data_type)(CR2, a))

% Linear Blessing
%=====

% Those page table entries that are used to translate the address
% range for which linear memory is defined are not part of the
% linear blessed range. This is because as part of the translation,
% reference bits in this area may be updated. Other PTEs not involved
% in the translation may well be part of plain memory.
%
% This lets the page fault handler work inside plain memory and update
% the active kernel page table in all areas besides the one in which the
% page-fault handler itself runs.

% make in memory explicit requirement !!!
linear_blessed?(s, ro_addr, rw_addr) : bool =
  % resolve OK?
  (Forall (a : Memory_Address_4G) : union(ro_addr, rw_addr)(a) Implies
    OK?(linear_resolve(a, Read)(s)) And OK?(linear_resolve(a, Execute)(s)))
  And
  (Forall (a : Memory_Address_4G) : rw_addr(a) Implies
    OK?(linear_resolve(a, Write)(s))) And
  % underlying blessing memory read addresses
  subset?(virt_to_phys_range(s, union(ro_addr, rw_addr),
    union singleton(Read), singleton(Execute)),
    union(pm_phy'ro_addr, pm_phy'rw_addr))

  And
  subset?(virt_to_phys_range(s, rw_addr, singleton(Write)), pm_phy'rw_addr)
  And
  subset?(address_in_pt_range?(s, union(ro_addr, rw_addr)), pm_phy'rw_addr)
  And
  % disjoint mapping of (ro_addr, rw_addr) and ptabs
  disjoint?(virt_to_phys_range(s, union(ro_addr, rw_addr)),
    address_in_pt_range?(s, union(ro_addr, rw_addr)))

  And
  % not shared
  % blessing memory read, memory_write addresses if these are disjoint
  (Forall (a1, a2 : Memory_Address_4G) :
    rw_addr(a1) And union(ro_addr, rw_addr)(a2) Implies
    Not linear_shared?(s)(a1, a2)) And
  % linear memory side effects
  True % no side effect restrictions

% proof status :-)
% is_linear_plain_memory?_TCC1 : TCC Obligation

% proof status :-)
% is_linear_plain_memory?_TCC2 : TCC Obligation

```

B PVS Theory Sources

```

% proof status :-)
% is_linear_plain_memory?_TCC3 : TCC Obligation

% proof status :-)
% is_linear_plain_memory?_TCC4 : TCC Obligation

% proof status :-)
% is_linear_plain_memory?_TCC5 : TCC Obligation

% proof status :-)
% is_linear_plain_memory?_TCC6 : TCC Obligation

is_linear_plain_memory?(pm : Plain_Memory[Linear_memory]) : bool =
  % linear pm
  pm'mem = linear_pm And
  % unmodified page-table area
  % cs
  (Forall (s : (pm'states)) :
    OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s))) And
  (Forall (s1, s2 : (pm'states)) :
    data(read_data(pm_phy, segment_reg_data_type)(CS)(s1)) =
      data(read_data(pm_phy, segment_reg_data_type)(CS)(s2))) And
  % pdir
  (Forall (s : (pm'states)) :
    OK?(read_data(pm_phy, pdir_data_type)(PDBR)(s))) And
  (Forall (s1, s2 : (pm'states)) :
    data(read_data(pm_phy, pdir_data_type)(PDBR)(s1)) =
      data(read_data(pm_phy, pdir_data_type)(PDBR)(s2))) And
  % pdir range
  (Forall (s1, s2 : (pm'states)) :
    pe_in_pdir_range?(s1, union(pm'ro_addr, pm'rw_addr)) =
      pe_in_pdir_range?(s2, union(pm'ro_addr, pm'rw_addr)) And
    pe_in_ptab_range?(s1, union(pm'ro_addr, pm'rw_addr)) =
      pe_in_ptab_range?(s2, union(pm'ro_addr, pm'rw_addr))) And
  % page tables
  (Forall (s : (pm'states), lvl : Level,
    a : (pe_in_pt_range?(s, union(pm'ro_addr, pm'rw_addr), lvl))) :
    OK?(read_data(pm_phy, paging_data_type(lvl))(a)(s))) And
  (Forall (s1, s2 : (pm'states), lvl : Level,
    a : (pe_in_pt_range?(s1, union(pm'ro_addr, pm'rw_addr), lvl))) :
    set_reference(data(read_data(pm_phy, paging_data_type(lvl))(a)(s1)),
      Write) =
      set_reference(data(read_data(pm_phy, paging_data_type(lvl))(a)(s2)),
      Write)) And
  % underlying blessed memory
  plain_memory?(pm_phy) And
  % underlying states
  pm'states = pm_phy'states And
  % underlying other actions
  subset?(pm'other_actions, pm_phy'other_actions) And
  subset?(linear_resolve_register_transformers, pm_phy'other_actions) And
  % memory address range

```

```

(Forall (a : (union(pm'ro_addr, pm'rw_addr))) :
  Mem?(type_of(a)) And 0 <= offset(a) And
  offset(a) < max_linear_offset) And
% linear_blessed_states
(Forall (s : (pm'states)) : linear_blessed?(s, pm'ro_addr, pm'rw_addr))

%%%%%%%%%% Plain Memory Structure for Linear Memory %%%%%%%%%%%

linear_plain_memory(states, oact, ro_addr, rw_addr)
  : Plain_Memory[Linear_memory] =
(#
  mem := linear_pm,
  states := states,
  ro_addr := ro_addr,
  rw_addr := rw_addr,
  other_actions := oact
#)

End Linear_Memory_Blessing

%>> MV:
% – todo: establish is_linear_plain_memory? from a single state and
% conclude that applying the state transformers yields new
% states for which is_linear_plain_memory? holds.

Linear_Memory_Properties[Physical_memory : Type,
  (Importing Plain_Memory[Physical_memory])
  pm_phy : Plain_Memory] : Theory

Begin

Importing Linear_Memory_Blessing[Physical_memory, pm_phy],
  Plain_Mem_Rewrites,
  Even_More_List_Props

s : Var Linear_memory
lvl : Var Level
pm : Var Plain_Memory[Linear_memory]

% Helpers
%=====

% rewrite memory_read(pm'mem...) = linear_read(...)
% proof status :-)
pm_read_linear : Lemma
Forall (a : Address) :
  is_linear_plain_memory?(pm) Implies
  memory_read(pm'mem)(a) = linear_read(a)

% proof status :-)
pm_write_linear : Lemma
Forall (a : Address, b : Byte) :
  is_linear_plain_memory?(pm) Implies

```

```

memory_write(pm'mem)(a, b) = linear_write(a, b)

% proof status :-)
pm_read_side_effect_linear : Lemma
  Forall (a : Address, bl : list[Byte], cp : bool) :
    is_linear_plain_memory?(pm) Implies
      memory_read_side_effect(pm'mem)(a, bl, cp) =
        linear_read_side_effect(a, bl, cp)

% proof status :-)
pm_write_side_effect_linear : Lemma
  Forall (a : Address, bl : list[Byte], cp : bool) :
    is_linear_plain_memory?(pm) Implies
      memory_write_side_effect(pm'mem)(a, bl, cp) =
        linear_write_side_effect(a, bl, cp)

% proof status :-)
pm_states : Lemma
  is_linear_plain_memory?(pm) Implies pm'states = pm_phy'states

% proof status :-)
pm_plain_phy : Lemma
  is_linear_plain_memory?(pm) Implies plain_memory?(pm_phy)

% proof status :-)
% pm_memory_addr_TCC1 : TCC Obligation

% proof status :-)
% pm_memory_addr_TCC2 : TCC Obligation

% proof status :-)
pm_memory_addr : Lemma
  Forall (a : Address) :
    is_linear_plain_memory?(pm) And union(pm'ro_addr, pm'rw_addr)(a) Implies
      in_memory(min_linear, max_linear)(a) And Mem?(type_of(a))

% proof status :-)
pm_linear_blessed : Lemma
  is_linear_plain_memory?(pm) And pm'states(s) Implies
    linear_blessed?(s, pm'ro_addr, pm'rw_addr)

% proof status :-)
% pm_linear_resolve_read_ok_TCC1 : TCC Obligation

% proof status :-)
pm_linear_resolve_read_ok : Lemma
  Forall (s : (pm'states), a : (union(pm'ro_addr, pm'rw_addr))) :
    is_linear_plain_memory?(pm) Implies OK?(linear_resolve(a, Read)(s))

% proof status :-)
% pm_linear_resolve_write_ok_TCC1 : TCC Obligation

```

```

% proof status :-)
pm_linear_resolve_write_ok : Lemma
  Forall (s : (pm'states), a : (pm'rw_addr)) :
    is_linear_plain_memory?(pm) Implies OK?(linear_resolve(a, Write)(s))

% proof status :-)
pm_linear_resolve_reg_transformers_other : Lemma
  is_linear_plain_memory?(pm)
  Implies
    subset?(linear_resolve_register_transformers, pm_phy'other_actions)

% proof status :-)
pm_unchanged_singleton_linear_resolve_reg_transformers : Lemma
  Forall (q : [Physical_memory -> SuperResult[Physical_memory]]) :
    is_linear_plain_memory?(pm) And linear_resolve_register_transformers(q)
  Implies
    unchanged_memory_invariant?(pm_phy'mem, pm_phy'states, singleton(q),
      union(pm_phy'ro_addr, pm_phy'rw_addr))

% proof status :-)
pm_address_in_pt_range_in_rw_addr : Lemma
  is_linear_plain_memory?(pm) And pm'states(s) Implies
    subset?(address_in_pt_range?(s, union(pm'ro_addr, pm'rw_addr)),
      pm_phy'rw_addr)

% proof status :-)
% address_block_in_memory_TCC1 : TCC Obligation

% proof status :-)
address_block_in_memory : Lemma
  Forall (a : Address, bl : list[Byte]) :
    is_linear_plain_memory?(pm) And
      subset?(address_block(a, length(bl)), union(pm'ro_addr, pm'rw_addr)) And
        cons?(bl)
  Implies
    a + length(bl) <= reg_size(max_linear)(type_of(a))

% proof status :-)
address_block_in_memory2 : Lemma
  Forall (a : Address, bl : list[Byte]) :
    is_linear_plain_memory?(pm) And
      subset?(address_block(a, length(bl)), union(pm'ro_addr, pm'rw_addr)) And
        cons?(bl)
  Implies
    0 <= a'offset

% proof status :-)
raise_fault_transformer_invariant : Lemma
  Forall (priv : Memory_privilege, access : Memory_access, present : bool,
    pfa : Memory_Address_4G) :
    is_linear_plain_memory?(pm) Implies
      transformer_invariant?(pm_phy'states, singleton(expr_2_super(

```

```

        raise_fault(priv, access, present, pfa))))

% proof status :-)
raise_fault_unchanged_memory_invariant : Lemma
  Forall (priv : Memory_privilege, access : Memory_access, present : bool,
    pfa : Memory_Address_4G) :
    is_linear_plain_memory?(pm) Implies
      unchanged_memory_invariant?(pm_phy'mem, pm_phy'states,
        singleton(expr_2_super(raise_fault(priv, access, present, pfa))),
        union(pm_phy'ro_addr, pm_phy'rw_addr))

% Side Effects
%=====
% proof status :-)
apply_side_effects_pm_states : Lemma
  Forall (l : list[[Memory_Address_4G, list[Byte]]],
    f : [Memory_Address_4G, list[Byte] ->
      [Linear_memory -> ExprResult[Linear_memory, list[Byte]]]]) :
    is_linear_plain_memory?(pm) And
      every(Lambda (e : [Memory_Address_4G, list[Byte]]) :
        transformer_invariant?(pm'states, singleton(expr_2_super(f(e)))))(l)
  Implies
    transformer_invariant?(pm'states, singleton(
      expr_2_super(apply_side_effects(l, f))))

% proof status :-)
apply_side_effects_ok : Lemma
  Forall (l : list[[Memory_Address_4G, list[Byte]]],
    f : [Memory_Address_4G, list[Byte] ->
      [Linear_memory -> ExprResult[Linear_memory, list[Byte]]]]) :
    is_linear_plain_memory?(pm) And
      pm'states(s) And
        (Forall (s1 : (pm'states)) :
          every(Lambda (e : [Memory_Address_4G, list[Byte]]) :
            OK?(f(e)(s1)))(l) And
            every(Lambda (e : [Memory_Address_4G, list[Byte]]) :
              transformer_invariant?(pm'states,
                singleton(expr_2_super(f(e)))))(l)
        )
  Implies
    OK?(apply_side_effects(l, f)(s))

% proof status :-)
% apply_side_effects_same_result_TCC1 : TCC Obligation

% proof status :-)
apply_side_effects_same_result : Lemma
  Forall (l : list[[Memory_Address_4G, list[Byte]]],
    f : [Memory_Address_4G, list[Byte] ->
      [Linear_memory -> ExprResult[Linear_memory, list[Byte]]]]) :
    is_linear_plain_memory?(pm) And
      pm'states(s) And
        (Forall (s1 : (pm'states)) :

```

```

    every(Lambda (e : [Memory_Address_4G, list[Byte]]) :
      OK?(f(e)(s1))(l) And
    (Forall (s1 : (pm'states)) :
      every(Lambda (e : [Memory_Address_4G, list[Byte]]) :
        OK?(f(e)(s1)) Implies data(f(e)(s1)) = e'2(l) And
      every(Lambda (e : [Memory_Address_4G, list[Byte]]) :
        transformer_invariant?(pm'states, singleton(expr_2_super(f(e))))(l)
    Implies
    data(apply_side_effects(l, f)(s)) =
    reduce(null, Lambda (e : [Address, list[Byte]], tail : list[Byte]) :
      Let head = e'2 In append(head, tail))(l)

% proof status :-)
apply_side_effects_unchanged : Lemma
Forall (l : list[[Memory_Address_4G, list[Byte]]],
  f : [Memory_Address_4G, list[Byte] ->
    [Linear_memory -> ExprResult[Linear_memory, list[Byte]]]],
  addresses : PRED[Memory_Address_4G]) :
  is_linear_plain_memory?(pm) And
  subset?(addresses, union(pm_phy'ro_addr, pm_phy'rw_addr)) And
  every(Lambda (e : [Memory_Address_4G, list[Byte]]) :
    unchanged_memory_invariant?(pm_phy'mem, pm_phy'states,
      singleton(expr_2_super(f(e))), addresses))(l)
Implies
  unchanged_memory_invariant?(pm_phy'mem, pm_phy'states,
    singleton(expr_2_super(apply_side_effects(l, f))), addresses)

% Main Resolve Results
%=====

% proof status :-)
subset_ptab_address_rw_addr : Lemma
Forall (base : Memory_Address_4G) :
  subset?(address_in_pt_range?(s,
    union(restrict[Address, Memory_Address_4G, bool](pm'ro_addr),
      restrict[Address, Memory_Address_4G, bool](pm'rw_addr))),
    restrict[Address, Memory_Address_4G, bool](pm_phy'rw_addr)) And
  subset?(address_block(base, expt(2, pe_size),
    extend[Address, Memory_Address_4G, bool, FALSE]
    (address_in_pt_range?(s,
      restrict[Address, Memory_Address_4G, bool]
      (union(pm'ro_addr, pm'rw_addr))))))

Implies
  subset?(address_block(base, expt(2, pe_size), pm_phy'rw_addr)

% proof status :-)
translate_transformer_invariant : Lemma
Forall (base : PTab_Address, lin_a : Memory_Address_4G,
  access : Memory_access, priv : Memory_privilege) :
  union(pm'ro_addr, pm'rw_addr)(lin_a) And
  is_linear_plain_memory?(pm) And

```



```

subset?(address_block(xlat_idx(lvl, base, lin_a), expt(2, pe_size)),
        pm_phy'rw_addr)
Implies
transformer_invariant?(pm'states, singleton(expr_2_super(
        translate(lvl, base, lin_a, access, priv))))

% proof status :-)
% translate_unchanged_pm_phy_except_pe_TCC1 : TCC Obligation

% proof status :-)
% translate_unchanged_pm_phy_except_pe_TCC2 : TCC Obligation

% proof status :-)
translate_unchanged_pm_phy_except_pe : Lemma
Forall (base : PTab_Address, lin_a : Memory_Address_4G,
        access : Memory_access, priv : Memory_privilege) :
union(pm'ro_addr, pm'rw_addr)(lin_a) And
is_linear_plain_memory?(pm) And
subset?(address_block(xlat_idx(lvl, base, lin_a), expt(2, pe_size)),
        pm_phy'rw_addr) And
(Forall (s : (pm'states)) :
    OK?(read_data(pm_phy, paging_data_type(lvl))
        (xlat_idx(lvl, base, lin_a))(s))) And
(Forall (s1, s2 : (pm'states)) :
    set_reference(data(read_data(pm_phy, paging_data_type(lvl))
        (xlat_idx(lvl, base, lin_a))(s1)), Write) =
    set_reference(data(read_data(pm_phy, paging_data_type(lvl))
        (xlat_idx(lvl, base, lin_a))(s2)), Write))

Implies
Let pe_addr = xlat_idx(lvl, base, lin_a) In
unchanged_memory_invariant?(pm_phy'mem, pm_phy'states,
    singleton(expr_2_super(translate(lvl, base, lin_a, access, priv))),
    difference(union(pm_phy'ro_addr, pm_phy'rw_addr),
        address_block(pe_addr, expt(2, pe_size))))

% proof status :-)
% translate_result_ok_TCC1 : TCC Obligation

% proof status :-)
translate_result_ok : Lemma
Forall (s1, s2 : (pm'states), base : PTab_Address,
        lin_a : Memory_Address_4G, access : Memory_access,
        priv : Memory_privilege) :
union(pm'ro_addr, pm'rw_addr)(lin_a) And
is_linear_plain_memory?(pm) And
pe_in_pt_range?(s1, union(pm'ro_addr, pm'rw_addr), lvl)
(xlat_idx(lvl, base, lin_a)) And
OK?(translate(lvl, base, lin_a, access, priv)(s1))

Implies
OK?(translate(lvl, base, lin_a, access, priv)(s2))

% proof status :-)

```

```

% translate_result_as_read_TCC1 : TCC Obligation

% proof status :-)
% translate_result_as_read_TCC2 : TCC Obligation

% proof status :-)
translate_result_as_read : Lemma
  Forall (base : PTab_Address, lin_a : Memory_Address_4G,
    access : Memory_access, priv : Memory_privilege) :
  union(pm'ro_addr, pm'rw_addr)(lin_a) And
  is_linear_plain_memory?(pm) And
  OK?(translate(lvl, base, lin_a, access, priv)(s))
Implies
  Let res = read_data(pm_phy, paging_data_type(lvl))
    (xlat_idx(lvl, base, lin_a))(s) In
  data(translate(lvl, base, lin_a, access, priv)(s)) =
  (is_leaf?(data(res)), base(data(res)))

% proof status :-)
translate_same_result : Lemma
  Forall (s1, s2 : (pm'states), base : PTab_Address,
    lin_a : Memory_Address_4G, ac1, ac2 : Memory_access,
    priv : Memory_privilege) :
  union(pm'ro_addr, pm'rw_addr)(lin_a) And
  is_linear_plain_memory?(pm) And
  pe_in_pt_range?(s1, union(pm'ro_addr, pm'rw_addr), lvl)
    (xlat_idx(lvl, base, lin_a)) And
  OK?(translate(lvl, base, lin_a, ac1, priv)(s1)) And
  OK?(translate(lvl, base, lin_a, ac2, priv)(s2))
Implies
  data(translate(lvl, base, lin_a, ac1, priv)(s1)) =
  data(translate(lvl, base, lin_a, ac2, priv)(s2))

% proof status :-)
% xlat_pe_in_pdir_range_TCC1 : TCC Obligation

% proof status :-)
% xlat_pe_in_pdir_range_TCC2 : TCC Obligation

% proof status :-)
% xlat_pe_in_pdir_range_TCC3 : TCC Obligation

% proof status :-)
xlat_pe_in_pdir_range : Lemma
  Forall (addr : Memory_Address_4G, s1, s2 : Linear_memory) :
  union(pm'ro_addr, pm'rw_addr)(addr) And
  is_linear_plain_memory?(pm) And
  pm'states(s1) And pm'states(s2)
Implies
  pe_in_pdir_range?(s1, union(pm'ro_addr, pm'rw_addr))
  (xlat_idx(pdir_lvl, data(read_data(pm_phy, pdir_data_type)
    (PDBR)(s2)))'base_addr, addr))

```

B PVS Theory Sources

```

% proof status :-)
% xlat_pe_in_pt_range_TCC1 : TCC Obligation

% proof status :-)
% xlat_pe_in_pt_range_TCC2 : TCC Obligation

% proof status :-)
% xlat_pe_in_pt_range_TCC3 : TCC Obligation

% proof status :-)
% xlat_pe_in_pt_range_TCC4 : TCC Obligation

% proof status :-)
% xlat_pe_in_pt_range_TCC5 : TCC Obligation

% proof status :-)
xlat_pe_in_pt_range : Lemma
  Forall (addr : Memory_Address_4G, access : Memory_access,
    priv : Memory_privilege, s1, s2 : Linear_memory) :
    union(pm'ro_addr, pm'rw_addr)(addr) And
      is_linear_plain_memory?(pm) And
        pm'states(s1) And pm'states(s2) And
          OK?(translate(pdir_lvl, data(read_data(pm_phy, pdir_data_type)
            (PDBR)(s1))'base_addr, addr, access, priv)
            (state(read_data(pm_phy, pdir_data_type)(PDBR)(s1)))) And
            Not data(translate(pdir_lvl, data(read_data(pm_phy, pdir_data_type)
              (PDBR)(s1))'base_addr, addr, access, priv)
                (state(read_data(pm_phy, pdir_data_type)(PDBR)(s1))))'1
          Implies
            pe_in_pt_range?(s2, union(pm'ro_addr, pm'rw_addr), ptab_lvl)
            (xlat_idx(ptab_lvl, data(translate(pdir_lvl,
              data(read_data(pm_phy, pdir_data_type)
                (PDBR)(s1))'base_addr, addr, access, priv)
                (state(read_data(pm_phy, pdir_data_type)(PDBR)(s1))))'2, addr))

% proof status :-)
linear_resolve_unchanged_pm_phy : Lemma
  Forall (addr : Memory_Address_4G, access : Memory_access) :
    union(pm'ro_addr, pm'rw_addr)(addr) And
      is_linear_plain_memory?(pm) And
        pm'states(s)
    Implies
      unchanged_memory_invariant?[Physical_memory](pm_phy'mem, pm'states,
        singleton(expr_2_super[Physical_memory, Address](
          linear_resolve(addr, access))),
        difference(union(pm_phy'ro_addr, pm_phy'rw_addr),
          address_in_pt_range?(s, union(pm'ro_addr, pm'rw_addr))))

% proof status :-)
linear_resolve_read_transformers_ok : Lemma
  Forall (addr : Memory_Address_4G) :

```

```

is_linear_plain_memory?(pm) And
  union(pm'ro_addr, pm'rw_addr)(addr)
Implies
  transformers_ok?(pm'states, singleton(expr_2_super(
    linear_resolve(addr, Read))))

% proof status :-)
linear_resolve_write_transformers_ok : Lemma
Forall (addr : Memory_Address_4G) :
  is_linear_plain_memory?(pm) And
  pm'rw_addr(addr)
Implies
  transformers_ok?(pm'states, singleton(expr_2_super(
    linear_resolve(addr, Write))))

% proof status :-)
linear_resolve_transformer_invariant : Lemma
Forall (addr : Memory_Address_4G, access : Memory_access) :
  is_linear_plain_memory?(pm) And
  union(pm'ro_addr, pm'rw_addr)(addr)
Implies
  transformer_invariant?(pm'states, singleton(expr_2_super(
    linear_resolve(addr, access))))

% proof status :-)
% linear_resolve_states_TCC1 : TCC Obligation

% proof status :-)
linear_resolve_states : Lemma
Forall (addr : Memory_Address_4G, access : Memory_access) :
  union(pm'ro_addr, pm'rw_addr)(addr) And
  is_linear_plain_memory?(pm) And
  pm'states(s) And
  OK?(linear_resolve(addr, access)(s))
Implies
  pm_phy'states(state(linear_resolve(addr, access)(s)))

% proof status :-)
linear_resolve_same_result : Lemma
Forall (addr : Memory_Address_4G, ac1, ac2 : Memory_access,
  s1, s2 : Linear_memory) :
  union(pm'ro_addr, pm'rw_addr)(addr) And
  is_linear_plain_memory?(pm) And
  pm'states(s1) And pm'states(s2) And
  OK?(linear_resolve(addr, ac1)(s1)) And
  OK?(linear_resolve(addr, ac2)(s2))
Implies
  data(linear_resolve(addr, ac1)(s1)) =
  data(linear_resolve(addr, ac2)(s2))

% Translation of addresses within same page
%=====

```

B PVS Theory Sources

```

% proof status :-)
delta_memory_address : Lemma
  Forall (addr : Memory_Address_4G, delta : nat) :
    rem(expt(2, min_page))(offset(addr)) + delta < expt(2, min_page)
  Implies
    offset(addr + delta) < max_linear_offset

% proof status :-)
% xlat_idx_same_page_address_TCC1 : TCC Obligation

% proof status :-)
% xlat_idx_same_page_address_TCC2 : TCC Obligation

% proof status :-)
xlat_idx_same_page_address : Lemma
  Forall (addr : Memory_Address_4G, delta : nat, lvl : Level,
    base : {a : Memory_Address_4G |
      aligned?(bits_per_level + pe_size)(offset(a))}) :
    rem(expt(2, min_page))(offset(addr)) + delta < expt(2, min_page)
  Implies
    xlat_idx(lvl, base, addr) = xlat_idx(lvl, base, addr + delta)

% proof status :-)
% xlat_ofs_same_page_address_TCC1 : TCC Obligation

% proof status :-)
% xlat_ofs_same_page_address_TCC2 : TCC Obligation

% proof status :-)
xlat_ofs_same_page_address : Lemma
  Forall (addr : Memory_Address_4G, delta : nat, lvl : Level,
    base : {a : Memory_Address_4G | aligned?(bus_width - (lvl + 1) *
      bits_per_level)(offset(a))}) :
    rem(expt(2, min_page))(offset(addr)) + delta < expt(2, min_page)
  Implies
    xlat_ofs(lvl, base, addr) + delta = xlat_ofs(lvl, base, addr + delta)

% proof status :-)
translate_same_page_address_ok : Lemma
  Forall (addr : Memory_Address_4G, delta : nat, lvl : Level,
    base : {a : Memory_Address_4G |
      aligned?(bits_per_level + pe_size)(offset(a))},
    access : Memory_access, priv : Memory_privilege) :
    rem(expt(2, min_page))(offset(addr)) + delta < expt(2, min_page) And
    OK?(translate(lvl, base, addr, access, priv)(s))
  Implies
    OK?(translate(lvl, base, addr + delta, access, priv)(s))

% proof status :-)
translate_same_page_address_result : Lemma
  Forall (addr : Memory_Address_4G, delta : nat, lvl : Level,

```

```

    base : {a : Memory_Address_4G |
            aligned?(bits_per_level + pe_size)(offset(a))},
    access : Memory_access, priv : Memory_privilege) :
rem(expt(2, min_page))(offset(addr)) + delta < expt(2, min_page) And
  OK?(translate(lvl, base, addr, access, priv)(s))
Implies
  translate(lvl, base, addr, access, priv)(s) =
  translate(lvl, base, addr + delta, access, priv)(s)

% proof status :-)
% linear_resolve_same_page_address_ok_TCC1 : TCC Obligation

% proof status :-)
linear_resolve_same_page_address_ok : Lemma
Forall (addr : Memory_Address_4G, delta : nat, ac : Memory_access) :
  rem(expt(2, min_page))(offset(addr)) + delta < expt(2, min_page) And
  OK?(linear_resolve(addr, ac)(s))
Implies
  OK?(linear_resolve(addr + delta, ac)(s))

% proof status :-)
% linear_resolve_same_page_address_TCC1 : TCC Obligation

% proof status :-)
linear_resolve_same_page_address : Lemma
Forall (addr : Memory_Address_4G, delta : nat, ac : Memory_access) :
  rem(expt(2, min_page))(offset(addr)) + delta < expt(2, min_page) And
  OK?(linear_resolve(addr, ac)(s))
Implies
  data(linear_resolve(addr, ac)(s)) + delta =
  data(linear_resolve(addr + delta, ac)(s))

% Linear Memory Side Effects
%=====

% proof status :-)
address_block_split_type : Lemma
Forall (a : Memory_Address_4G, bl : list[Byte]) :
  is_linear_plain_memory?(pm) And
  subset?(address_block(a, length(bl)), union(pm'ro_addr, pm'rw_addr))
Implies
  every(LAMBDA (t : [Address, list[Byte]]): Mem?(t'1'type_of) AND
    0 <= t'1'offset And
    t'1'offset < max_linear_offset)
    (split(min_page, a, bl))

% proof status :-)
% same_page_address_block_read_TCC1 : TCC Obligation

% proof status :-)
same_page_address_block_read : Lemma

```

```

Forall (a1, a2 : Memory_Address_4G, s1, s2 : posnat) :
  is_linear_plain_memory?(pm) And
  pm'states(s) And
  union(pm'ro_addr, pm'rw_addr)(a2) And
  a1 <= a2 And a2 + s2 <= a1 + s1 And
  rem(expt(2, min_page))(offset(a2)) + s2 <= expt(2, min_page) And
  subset?(address_block(a1, s1), union(pm'ro_addr, pm'rw_addr))
Implies
  subset?(address_block(data(linear_resolve(a2, Read)(s)), s2),
    union(pm_phy'ro_addr, pm_phy'rw_addr))

% proof status :- )
% same_page_address_block_write_TCC1 : TCC Obligation

% proof status :- )
same_page_address_block_write : Lemma
Forall (a1, a2 : Memory_Address_4G, s1, s2 : posnat) :
  is_linear_plain_memory?(pm) And
  pm'states(s) And
  pm'rw_addr(a2) And
  a1 <= a2 And a2 + s2 <= a1 + s1 And
  rem(expt(2, min_page))(offset(a2)) + s2 <= expt(2, min_page) And
  subset?(address_block(a1, s1), pm'rw_addr)
Implies
  subset?(address_block(data(linear_resolve(a2, Write)(s)), s2),
    pm_phy'rw_addr)

% proof status :- )
% split_linear_read_side_effects_states_TCC1 : TCC Obligation

% proof status :- )
split_linear_read_side_effects_states : Lemma
Forall (a : Memory_Address_4G, bl : (cons?[Byte])) :
  is_linear_plain_memory?(pm) And
  subset?(address_block(a, length(bl)), union(pm'ro_addr, pm'rw_addr))
Implies
  every(Lambda (e : [Memory_Address_4G, list[Byte]]) :
    transformer_invariant?(pm'states,
      singleton(expr_2_super(linear_read_side_effect_in_page(e))))
    (split(min_page, a, bl))

% proof status :- )
% split_linear_write_side_effects_states_TCC1 : TCC Obligation

% proof status :- )
split_linear_write_side_effects_states : Lemma
Forall (a : Memory_Address_4G, bl : (cons?[Byte])) :
  is_linear_plain_memory?(pm) And
  subset?(address_block(a, length(bl)), pm'rw_addr)
Implies
  every(Lambda (e : [Memory_Address_4G, list[Byte]]) :
    transformer_invariant?(pm'states,

```

```

        singleton(expr_2_super(linear_write_side_effect_in_page(e))))
    (split(min_page, a, bl))

% proof status :-)
split_linear_read_side_effects_ok : Lemma
Forall (a : Memory_Address_4G, bl : (cons?[Byte])) :
  is_linear_plain_memory?(pm) And
    subset?(address_block(a, length(bl)),
      union(pm'ro_addr, pm'rw_addr)) And
      pm'states(s)
Implies
  every(Lambda (e : [Memory_Address_4G, list[Byte]]) :
    OK?(linear_read_side_effect_in_page(e)(s)))
    (split(min_page, a, bl))

% proof status :-)
split_linear_write_side_effects_ok : Lemma
Forall (a : Memory_Address_4G, bl : (cons?[Byte])) :
  is_linear_plain_memory?(pm) And
    subset?(address_block(a, length(bl)), pm'rw_addr) And
      pm'states(s)
Implies
  every(Lambda (e : [Memory_Address_4G, list[Byte]]) :
    OK?(linear_write_side_effect_in_page(e)(s)))
    (split(min_page, a, bl))

% proof status :-)
split_linear_read_side_effects_data : Lemma
Forall (a : Memory_Address_4G, bl : (cons?[Byte])) :
  is_linear_plain_memory?(pm) And
    subset?(address_block(a, length(bl)),
      union(pm'ro_addr, pm'rw_addr)) And
      pm'states(s)
Implies
  every(Lambda (e : [Memory_Address_4G, list[Byte]]) :
    OK?(linear_read_side_effect_in_page(e)(s)) Implies
      data(linear_read_side_effect_in_page(e)(s)) = e'2)
    (split(min_page, a, bl))

% proof status :-)
split_linear_write_side_effects_data : Lemma
Forall (a : Memory_Address_4G, bl : (cons?[Byte])) :
  is_linear_plain_memory?(pm) And
    subset?(address_block(a, length(bl)), pm'rw_addr) And
      pm'states(s)
Implies
  every(Lambda (e : [Memory_Address_4G, list[Byte]]) :
    OK?(linear_write_side_effect_in_page(e)(s)) Implies
      data(linear_write_side_effect_in_page(e)(s)) = e'2)
    (split(min_page, a, bl))

% proof status :-)

```



```

split_linear_read_side_effects_unchanged_memory : Lemma
  Forall (a : Memory_Address_4G, bl : (cons?[Byte])) :
    is_linear_plain_memory?(pm) And
      subset?(address_block(a, length(bl)), union(pm'ro_addr, pm'rw_addr))
  Implies
    every(Lambda (e : [Memory_Address_4G, list[Byte]]) :
      unchanged_memory_invariant?(pm_phy'mem, pm'states,
        singleton(expr_2_super(linear_read_side_effect_in_page(e))),
        union(pm_phy'ro_addr, pm_phy'rw_addr)))
      (split(min_page, a, bl))

% proof status :-)
split_linear_write_side_effects_unchanged_memory : Lemma
  Forall (a : Memory_Address_4G, bl : (cons?[Byte])) :
    is_linear_plain_memory?(pm) And
      subset?(address_block(a, length(bl)), pm'rw_addr)
  Implies
    every(Lambda (e : [Memory_Address_4G, list[Byte]]) :
      unchanged_memory_invariant?(pm_phy'mem, pm'states,
        singleton(expr_2_super(linear_write_side_effect_in_page(e))),
        union(pm_phy'ro_addr, pm_phy'rw_addr)))
      (split(min_page, a, bl))

% linear_unchanged_invariant
%=====

% proof status :-)
linear_unchanged_invariant : Lemma
  Forall (transformers : PRED[[Linear_memory -> SuperResult[Linear_memory]]],
    addresses : PRED[Memory_Address_4G]) :
    is_linear_plain_memory?(pm) And
      subset?(addresses, union(pm'ro_addr, pm'rw_addr)) And
      (Forall (s : (pm'states)) :
        unchanged_memory_invariant?(pm_phy'mem, pm_phy'states,
          transformers,
          virt_to_phys_range(s, addresses, union(singleton(Read),
            singleton(Execute))))))
  Implies
    unchanged_memory_invariant?(pm'mem, pm'states, transformers, addresses)

% simple helpers for unchanged lemmas
%=====

% proof status :-)
% pm_phy_read_after_resolve_ok_TCC1 : TCC Obligation

% proof status :-)
pm_phy_read_after_resolve_ok : Lemma

```

```

Forall (a1 : Address, a2 : Memory_Address_4G, ac : Memory_access) :
  OK?(linear_resolve(a2, ac)(s)) And
  plain_memory?(pm_phy) And
  pm_phy'states(state(linear_resolve(a2, ac)(s))) And
  union(pm_phy'ro_addr, pm_phy'rw_addr)(a1)
Implies
  OK?(memory_read(pm_phy'mem)(a1)(state(linear_resolve(a2, ac)(s))))

```

```

% proof status :-)

```

```

pm_resolve_address : Lemma

```

```

Forall (a : Memory_Address_4G) :
  is_linear_plain_memory?(pm) And
  pm'states(s) And
  virt_to_phys_range(s, union(pm'ro_addr, pm'rw_addr),
    union singleton(Read), singleton(Execute)))(a)
Implies
  difference(union(pm_phy'ro_addr, pm_phy'rw_addr),
    address_in_pt_range?(s, union(pm'ro_addr, pm'rw_addr)))(a)

```

```

% proof status :-)

```

```

pm_resolve_address_write : Lemma

```

```

Forall (a : Memory_Address_4G) :
  is_linear_plain_memory?(pm) And
  pm'states(s) And
  virt_to_phys_range(s, pm'rw_addr, singleton(Write))(a)
Implies
  difference(union(pm_phy'ro_addr, pm_phy'rw_addr),
    address_in_pt_range?(s, union(pm'ro_addr, pm'rw_addr)))(a)

```

```

End Linear_Memory_Properties

```

```

Linear_Memory_Blessing_Properties

```

```

  [% the underlying physical memory
   Physical_memory : Type,
   (Importing Plain_Memory[Physical_memory])
   pm_phy : Plain_Memory
  ] : Theory

```

```

Begin

```

```

Importing Linear_Memory_Properties[Physical_memory, pm_phy]

```

```

pm : Var Plain_Memory[Linear_memory]

```

```

% Blessing results

```

```

%=====

```

```

% proof status :-)

```

```

linear_plain_transformers_ok : Lemma

```

```

  is_linear_plain_memory?(pm)

```

```

Implies

```

```

transformers_ok?(pm'states, union(union(
  memory_read_transformers(pm'mem, union(pm'ro_addr, pm'rw_addr)),
  memory_write_transformers(pm'mem, pm'rw_addr)), union(
  memory_read_side_effect_super_transformers(pm'mem,
    union(pm'ro_addr, pm'rw_addr)),
  memory_write_side_effect_super_transformers(pm'mem, pm'rw_addr))))))

% proof status :- )
linear_plain_unchanged_memory_invariant : Lemma
  is_linear_plain_memory?(pm)
Implies
  unchanged_memory_invariant?(pm'mem, pm'states,
    union(union(pm'other_actions,
      memory_read_transformers(pm'mem, union(pm'ro_addr, pm'rw_addr))),
      union(memory_read_side_effect_super_transformers(pm'mem,
        union(pm'ro_addr, pm'rw_addr)),
        memory_write_side_effect_super_transformers(pm'mem, pm'rw_addr))),
        union(pm'ro_addr, pm'rw_addr)))

% proof status :- )
linear_plain_unchanged_memory_invariant_write : Lemma
  is_linear_plain_memory?(pm)
Implies
  unchanged_memory_invariant?(pm'mem, pm'states,
    memory_write_transformers(pm'mem, pm'rw_addr), pm'ro_addr)

% proof status :- )
linear_plain_unchanged_memory_write_invariant : Lemma
  is_linear_plain_memory?(pm)
Implies
  unchanged_memory_write_invariant?(pm'mem, pm'states, pm'rw_addr)

% proof status :- )
linear_plain_transformer_invariant : Lemma
  is_linear_plain_memory?(pm)
Implies
  transformer_invariant?(pm'states, union(union(pm'other_actions,
    union(memory_read_transformers(pm'mem, union(pm'ro_addr, pm'rw_addr)),
      memory_write_transformers(pm'mem, pm'rw_addr))),
    union(memory_read_side_effect_super_transformers(pm'mem,
      union(pm'ro_addr, pm'rw_addr)),
      memory_write_side_effect_super_transformers(pm'mem, pm'rw_addr))))))

% proof status :- )
linear_plain_changed_memory_invariant : Lemma
  is_linear_plain_memory?(pm)
Implies
  changed_memory_invariant?(pm'mem, pm'states, pm'rw_addr)

% We must use expression transformers instead of super transformers
% to match the list argument of the side effect transformer in

```

```

% side_effect_content_unchanged against the transformer obtained
% from the union of memory side effect transformers. Super
% transformers give us only equality on result states.

% proof status :-)
linear_plain_read_side_effect_unchanged : Lemma
  is_linear_plain_memory?(pm)
Implies
  side_effect_content_unchanged(union(pm'ro_addr, pm'rw_addr), pm'states,
    memory_read_side_effect(pm'mem))

% proof status :-)
linear_plain_write_side_effect_unchanged : Lemma
  is_linear_plain_memory?(pm)
Implies
  side_effect_content_unchanged(pm'rw_addr, pm'states,
    memory_write_side_effect(pm'mem))

%%%%%% Final Blessing Result %%%%%%
% proof status :-)
linear_memory_plain_memory : Lemma
  is_linear_plain_memory?(pm)
  Implies plain_memory?(pm)

End Linear_Memory_Blessing_Properties

```

B.5 challenge-phymem.pvs

```

Phy_Mem_Properties[(IMPORTING IA32) min, max : Memory_Address] : Theory
% Properties of physical memory
Begin
IMPORTING Memory_Physical_Memory[min, max], Memory_Change_2
IMPORTING Physical_Memory_Properties[min, max]

addr : Var Address
s : Var Physical_memory

% proof status :-)
phy_mem_write_transformers_ok : Lemma
  transformers_ok?(fullset[Physical_memory],
    memory_write_transformers(phy_mem, in_memory(min, max)))

% proof status :-)
phy_mem_read_transformers_ok : Lemma
  transformers_ok?(fullset[Physical_memory],
    memory_read_transformers(phy_mem, in_memory(min, max)))

% proof status :-)
phy_mem_read_side_effects_transformers_ok : Lemma
  transformers_ok?(fullset[Physical_memory],
    memory_read_side_effect_super_transformers(phy_mem, in_memory(min, max)))

```

```

% proof status :-)
phy_mem_write_side_effects_transformers_ok : Lemma
  transformers_ok?(fullset[Physical_memory],
    memory_write_side_effect_super_transformers(phy_mem, in_memory(min, max)))

% proof status :-)
phy_mem_unchanged_memory_invariant_read_transformers : Lemma
  unchanged_memory_invariant?(phy_mem, fullset[Physical_memory],
    memory_read_transformers(phy_mem, in_memory(min, max)),
    in_memory(min, max))

% proof status :-)
phy_mem_unchanged_memory_invariant_read_side_effect_transformers : Lemma
  unchanged_memory_invariant?(phy_mem, fullset[Physical_memory],
    memory_read_side_effect_super_transformers(phy_mem, in_memory(min, max)),
    in_memory(min, max))

% proof status :-)
phy_mem_unchanged_memory_invariant_write_side_effect_transformers : Lemma
  unchanged_memory_invariant?(phy_mem, fullset[Physical_memory],
    memory_write_side_effect_super_transformers(phy_mem, in_memory(min, max)),
    in_memory(min, max))

% proof status :-)
phy_mem_unchanged_memory_write_invariant : Lemma
  unchanged_memory_write_invariant?(phy_mem, fullset[Physical_memory],
    in_memory(min, max))

% proof status :-)
phy_mem_changed_memory_invariant : Lemma
  changed_memory_invariant?(phy_mem, fullset[Physical_memory],
    in_memory(min, max))

```

End Phy_Mem_Properties

Phy_Mem_Blessing[(**IMPORTING** IA32) min, max : Memory_Address] : **Theory**
% all physical memory is blessed memory

Begin

Importing Phy_Mem_Properties[min, max], Plain_Memory

oact : **Var** PRED[[Physical_memory→SuperResult[Physical_memory]]]

```

phy_pm(oact) : Plain_Memory[Physical_memory] = (#
  mem := phy_mem,
  states := fullset[Physical_memory],
  ro_addr := emptyset[Address],
  rw_addr := in_memory(min, max),
  other_actions := oact
#)

```

```

% proof status :-)
phy_mem_plain_memory : Lemma
  unchanged_memory_invariant?(phy_mem,
    fullset[Physical_memory], oact, in_memory(min, max))
  Implies
    plain_memory?(phy_pm(oact))

End Phy_Mem_Blessing

Phy_Mem_Blessing_Data[Data : Type, (IMPORTING IA32) min, max : Memory_Address] : Theory
Begin

  Importing Phy_Mem_Blessing[min, max], Abstract_Read_Write

  addr : Var Address
  idt : Var (interpreted_data_type?[Data])

  % proof status :-)
  in_memory_blessed_memory : Lemma
    in_memory?(idt, addr, min, max) Implies
      in_blessed_memory?(idt, addr, in_memory(min, max))

End Phy_Mem_Blessing_Data

Phy_Mem_Challenge_Read_Same
[Data : Type, (IMPORTING IA32) min, max : Memory_Address] : Theory
%
% read after write at the same place returns the item written
Begin

  IMPORTING Phy_Mem_Blessing[min, max],
    Plain_Mem_Properties_2[Physical_memory, Data],
    Phy_Mem_Blessing_Data[Data, min, max]

  dt : Var (interpreted_data_type?[Data])
  s : Var Physical_memory
  addr : Var Address
  data : Var Data

  % proof status :-)
  read_write_ok : Lemma in_memory?(dt, addr, min, max)
    IMPLIES
      OK?((
        write_data(phy_mem,dt)(addr, data) ##
        read_data(phy_mem,dt)(addr)
      ))(s)
    )

  % proof status :-)

```

B PVS Theory Sources

```
% read_write_res_TCC1 : TCC Obligation

% proof status :-)
read_write_res : Lemma in_memory?(dt, addr, min, max)
  IMPLIES
  data((
    write_data(phy_mem,dt)(addr, data) ##
    read_data(phy_mem,dt)(addr)
  )(s))
  = data

end Phy_Mem_Challenge_Read_Same

Phy_Mem_Challenge_Read_Other
  [(IMPORTING IA32) min, max : Memory_Address, Data1, Data2 : Type] : Theory
% Phy_Mem_Challenge_Read_Other : Theory
%
% for read after write at a different place you can discard the write
%
Begin
  IMPORTING Phy_Mem_Blessing[min, max],
    Plain_Mem_Properties_3[Physical_memory, Data1, Data2],
    Phy_Mem_Blessing_Data

  dt1 : Var (interpreted_data_type?[Data1])
  dt2 : Var (interpreted_data_type?[Data2])
  s : Var Physical_memory
  addr1, addr2 : Var Address
  data1 : Var Data1

  % proof status :-)
  read_write_other_ok : Lemma
    in_memory?(dt1, addr1, min, max) AND
    in_memory?(dt2, addr2, min, max) AND
    blocks_disjoint?(addr1, size(uidt(dt1)), addr2, size(uidt(dt2))) AND
    valid_in_mem(phy_mem,dt2)(addr2)(s)
    IMPLIES
    OK?
    ((
      write_data(phy_mem,dt1)(addr1, data1) ##
      read_data(phy_mem,dt2)(addr2)
    )(s)
  )

  % proof status :-)
  % read_write_other_res_TCC1 : TCC Obligation

  % proof status :-)
  % read_write_other_res_TCC2 : TCC Obligation

  % proof status :-)
```

```

read_write_other_res : Lemma
  in_memory?(dt1, addr1, min, max) AND
  in_memory?(dt2, addr2, min, max) AND
  blocks_disjoint?(addr1, size(uidt(dt1)), addr2, size(uidt(dt2))) AND
  valid_in_mem(phy_mem,dt2)(addr2)(s)
  IMPLIES
    data((
      write_data(phy_mem,dt1)(addr1, data1) ##
      read_data(phy_mem,dt2)(addr2)
    )(s))
  =
  data(read_data(phy_mem,dt2)(addr2)(s))

```

```

% proof status :-)
read_read_ok : Lemma
  in_memory?(dt1, addr1, min, max) AND
  in_memory?(dt2, addr2, min, max) AND
  valid_in_mem(phy_mem,dt1)(addr1)(s) And
  valid_in_mem(phy_mem,dt2)(addr2)(s)
  Implies
    OK?(
      ( read_data(phy_mem, dt1)(addr1) ##
        read_data(phy_mem, dt2)(addr2)
      )(s)
    )

```

```

% proof status :-)
% read_read_TCC1 : TCC Obligation

```

```

% proof status :-)
% read_read_TCC2 : TCC Obligation

```

```

% proof status :-)
read_read : Lemma
  in_memory?(dt1, addr1, min, max) AND
  in_memory?(dt2, addr2, min, max) AND
  valid_in_mem(phy_mem,dt1)(addr1)(s) And
  valid_in_mem(phy_mem,dt2)(addr2)(s)
  Implies
    data (
      ( read_data(phy_mem, dt1)(addr1) ##
        read_data(phy_mem, dt2)(addr2)
      )(s)
    )
  = data(read_data(phy_mem, dt2)(addr2)(s))

```

```
End Phy_Mem_Challenge_Read_Other
```

B.6 challenge-ptab-sync-master.pvs

Challenge_Ptab_Sync_Master : **Theory**
Begin

Importing ptab_sync_master

% to do: specification and proofs

End Challenge_Ptab_Sync_Master

B.7 constants.pvs

% general constants and types
% (including architecture dependent ones)

General : **Theory**

% architecture independent constants and types

Begin

Importing Number_Props

bits_per_byte : posnat

min_bits_per_byte : nat = 8

bits_per_byte_minimum : **Axiom** bits_per_byte \geq min_bits_per_byte

max_byte : nat = $\text{expt}(2, \text{bits_per_byte})$

% proof status :-)

% Byte_TCC1 : TCC Obligation

Byte : **Nonempty_Type** = below(max_byte) **containing** 0

% proof status :-)

less_than_max_byte : **Lemma Forall**(n : nat) :
n < $\text{expt}(2, 8)$ **Implies** n < max_byte

End General

Register_Id : **Datatype**

Begin

Mem_ : Mem?

EAX_ : EAX?

EBX_ : EBX?

ECX_ : ECX?

EDX_ : EDX?

ESI_ : ESI?

EDI_ : EDI?

ESP_ : ESP?

EBP_ : EBP?

```

CS_ : CS?
DS_ : DS?
ES_ : ES?
FS_ : FS?
GS_ : GS?
SS_ : SS?

```

```
IP_ : IP?
```

```
EFLAGS_ : EFLAGS?
```

```

CR0_ : CR0?
CR2_ : CR2?
PDBR_ : PDBR?
CR4_ : CR4?

```

```

IDTR_ : IDTR?
GDTR_ : GDTR?
LDTR_ : LDTR?
TR_ : TR?

```

```

SYSENTER_CS_ : SYSENTER_CS?
SYSENTER_ESP_ : SYSENTER_ESP?
SYSENTER_EIP_ : SYSENTER_EIP?

```

End Register_Id

Address_Helpers : **Theory**

Begin

Importing General, Register_Id

Address : **Type** = [# type_of : Register_Id, offset : int #]

% Address Constructors

Mem(offset : int) : Address = (# type_of := Mem_, offset := offset #)

CR0 : Address = (# type_of := CR0_, offset := 0 #)

CR2 : Address = (# type_of := CR2_, offset := 0 #)

CR4 : Address = (# type_of := CR4_, offset := 0 #)

PDBR : Address = (# type_of := PDBR_, offset := 0 #)

CS : Address = (# type_of := CS_, offset := 0 #)

DS : Address = (# type_of := DS_, offset := 0 #)

ES : Address = (# type_of := ES_, offset := 0 #)

FS : Address = (# type_of := FS_, offset := 0 #)

GS : Address = (# type_of := GS_, offset := 0 #)

SS : Address = (# type_of := SS_, offset := 0 #)

Segment_Register_Address : **Type** =

{a : Address | a = CS **Or** a = DS **Or** a = ES **Or** a = FS **Or** a = GS **Or** a = SS}

B PVS Theory Sources

```
Memory_Address : Type = {a : Address | Mem?(a'type_of)}

a : Var Address
size : Var int

+(a, size) : Address = a With [(offset) := a'offset + size]

m_addr : Var Memory_Address

% proof status :-)
type_add_rewrite : Lemma
  (m_addr + size)'type_of = m_addr'type_of

% proof status :-)
offset_add_rewrite : Lemma
  (m_addr + size)'offset = m_addr'offset + size

% proof status :-)
type_add_memory : Judgement +(m_addr, size) Has_Type Memory_Address

% proof status :-)
address_add_zero : Lemma
  Forall (a : Address) : a + 0 = a

% proof status :-)
address_add_sum : Lemma
  Forall (a : Address, i, j : int) : (a + i) + j = a + (i + j)

a1, a2 : Var Address

< (a1, a2) : bool = a1'type_of = a2'type_of And a1'offset < a2'offset;

<= (a1, a2) : bool = a1'type_of = a2'type_of And a1'offset <= a2'offset

% Some register groups

GP?(id : Register_Id) : bool =
  EAX?(id) Or EBX?(id) Or ECX?(id) Or EDX?(id) Or ESI?(id) Or
  EDI?(id) Or EBP?(id) Or ESP?(id)

Segment_Reg?(id : Register_Id) : bool =
  CS?(id) Or DS?(id) Or ES?(id) Or FS?(id) Or GS?(id) Or SS?(id)

Control_Reg?(id : Register_Id) : bool =
  CR0?(id) Or CR2?(id) Or CR4?(id) Or PDBR?(id)

Table_Reg?(id : Register_Id) : bool =
  IDTR?(id) Or GDTR?(id) Or LDTR?(id) Or TR?(id)

MSR?(id : Register_Id) : bool =
  SYSENTER_CS?(id) Or SYSENTER_ESP?(id) Or SYSENTER_EIP?(id)
```

```

% Register sizes

% proof status :-)
% reg_size_TCC1 : TCC Obligation

% proof status :-)
% reg_size_TCC2 : TCC Obligation

reg_size(max : Memory_Address)(id : Register_Id) : Address =
  Cases id Of
    Mem_ : (# type_of := id, offset := offset(max) #),
    IP_ : (# type_of := id, offset := expt(2, bits_per_byte * 4) #),
    EFLAGS_ : (# type_of := id, offset := expt(2, bits_per_byte * 4) #),

    IDTR_ : (# type_of := id, offset := expt(2, bits_per_byte * 8) #),
    GDTR_ : (# type_of := id, offset := expt(2, bits_per_byte * 8) #),
    LDTR_ : (# type_of := id, offset := expt(2, bits_per_byte * 14) #),
    TR_ : (# type_of := id, offset := expt(2, bits_per_byte * 14) #),

    SYSENTER_CS_ : (# type_of := id, offset := expt(2, bits_per_byte * 4) #),
    SYSENTER_ESP_ : (# type_of := id, offset := expt(2, bits_per_byte * 4) #),
    SYSENTER_EIP_ : (# type_of := id, offset := expt(2, bits_per_byte * 4) #)
  Else
    Cond
      GP?(id) -> (# type_of := id, offset := expt(2, bits_per_byte * 4) #),
      Segment_Reg?(id) -> (# type_of := id, offset := expt(2, bits_per_byte * 14) #),
      Control_Reg?(id) -> (# type_of := id, offset := expt(2, bits_per_byte * 4) #)
    EndCond
  EndCases

reg_base(min : Memory_Address)(id : Register_Id) : Address =
  If Mem?(id) Then
    min
  Else
    (# type_of := id, offset := 0 #)
  Endif

in_memory(min, max : Memory_Address)(a : Address) : bool =
  reg_base(min)(a'type_of) <= a And a < reg_size(max)(a'type_of)

floor(b : posnat)(a : Address) : Address =
  a With [(offset) := floor(b)(offset(a))]

End Address_Helpers

IA32 : Theory
% architecture dependent constants and types
Begin
  IMPORTING Address_Helpers

```

B PVS Theory Sources

```
% type of addresses as subtype of natural numbers
bus_width : nat = 32

max_linear_offset : nat = expt(2, bus_width)
max_linear : Address = Mem(max_linear_offset)

min_linear : Address = Mem(0)

% proof status :-)
max_linear_offset_val : Lemma max_linear_offset = 4294967296

% max_linear_expt_val : Lemma offset(max_linear) = expt(2, 32)

% memory addresses below 2^32, for those who need it
Memory_Address_4G : Type = {a : Memory_Address | 0 <= a'offset And a'offset < max_linear_offset}

% proof status :-)
% mem_addr_4g_TCC1 : TCC Obligation

mem_addr_4g(R : PRED[(in_memory(min_linear, max_linear))])(a : Memory_Address_4G) : bool = R(a)

% smallest page size
min_page : posnat = 12

AUTO_REWRITE- bus_width, max_linear, max_linear_offset_val
```

End IA32

B.8 conversions.pvs

```
% $Id: conversions.pvs,v 1.11.4.2.2.1.2.3 2008/05/14 14:24:58 tews Exp $
%
% Author: Tjark Weber
% (c) 2007 Radboud University
%
% A shallow embedding of (some) C++ conversions as state transformers.

% Section numbers in comments throughout this file refer to
% "Programming languages – C++", ISO/IEC 14882:1998(E)

% Deviations from the C++ standard:
%
% This is preliminary work. Most of the standard hasn't even been considered
% yet.

% A note on PVS conversions: PVS has conversions as well. Should we declare
% the C++ conversion functions as PVS conversions, so that they are applied
% automatically, based on type information? We have decided against doing so
% for the following reasons. 1) Explicit conversions make the PVS translation
% of C++ code less readable, but the translated code is of little interest.
% What really matters is the proof state, which necessarily contains the
```

```
% conversions anyway. 2) Explicit conversions give us more control. It is not
% quite clear (to me, anyway) how the PVS type checker decides which
% conversions must be applied, and in which order. 3) Some C++ conversions
% require additional information (e.g. the lvalue-to-rvalue conversion must
% know the type of the stored value), and must therefore be made explicit
% anyway.
```

```
% Naming convention: conversions between different types are named
% <source>2<dest>
% where <source> and <dest> are the source and destination type, respectively, as
% given in types.pvs. Examples: bool2int, int2uint.
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% 4.1: Lvalue to rvalue conversion
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
LvalueToRvalueConversion[State, Data : Type] : THEORY
BEGIN
```

```
IMPORTING Abstract_Read_Write_Plain
```

```
pm : VAR Plain_Memory[State]
dt : VAR (interpreted_data_type?[Data])
```

```
s : VAR State
```

```
a_ex : VAR [State -> ExprResult[State, Address]]
```

```
% MV>> do we need l2r for non-fundamental types??
% 4.1.2: l2r(class) returns an rvalue to a copy initialised temporary
l2r(pm, dt)(a_ex) : [State -> ExprResult[State, Data]] =
(
  a_ex ## lambda (a : Address):
  read_data(pm, dt)(a) ## lambda (d : Data):
  ok_result(d)
)
```

```
%MV>>
% l2r_class(pm, copy_constructor)(a_ex) : ET[State, Data]
% with_new_stackvar()(lambda (temporary : Address) :
% copy_constructor(temporary, a_ex)
% XXX
```

```
%MV>>
% It seems like we could restrict Data in ET[.., Data] to fundamental types;
% For classes it seems the standard requires an in place creation or temporaries and the
% only thing to do with a class is to access a member.
```

```
END LvalueToRvalueConversion
```

B PVS Theory Sources

% MV:>> make promotions and conversions polymorphic over the fundamental types

%%%
%

% 4.5: Integral promotions

%

%%%

IntegralPromotions[State : Type] : THEORY
BEGIN

IMPORTING Cpp_Types, Ok_Result_Rewrite

% 4.5(1)

% "An rvalue of type char [...] can be converted to an rvalue of type int if
% int can represent all the values of the source type; otherwise, the source
% rvalue can be converted to an rvalue of type unsigned int."

% We take this to mean that in the former case, the value is not changed.

*% TODO: I have no idea how to model this correctly, for the *type* of the*
% conversion would depend on the condition. Also, I'm not sure what
% should happen to source rvalues when int cannot represent all the
% values of the source type, and unsigned int cannot represent the
% particular rvalue. Below, only the first alternative (conversion to
% int) is modeled. However, 4.7(3) specifies the behavior of conversions
*% to int when those conversions are *not* available as integral*
*% promotions. To sum up, we can *always* convert to int; the behavior is*
% specified either by 4.5 or by 4.7. Likewise, we can always convert to
% unsigned int; this is modeled in theory IntegralConversions below.

% char2int
% schar2int
% uchar2int
% short2int
% ushort2int

typ : **Var** Cpp_Subtype(cv({t : Cpp_Type | char?(t) **Or** schar?(t) **Or** uchar?(t) **Or**
short?(t) **Or** ushort?(t)}))

% proof status : -)
% int_conversion_TCC1 : TCC Obligation

% proof status : -)
% int_conversion_TCC2 : TCC Obligation

% proof status : -)
% int_conversion_TCC3 : TCC Obligation

int_conversion(typ) : [(range(typ)) -> (range(int))]

```

% proof status :-)
% int_conversion_spec_TCC1 : TCC Obligation

int_conversion_spec : AXIOM
  forall (n : (range(typ))) :
    % 4.5(1) / 4.7(3)
    range(int)(n) Implies int_conversion(typ)(n) = n
    % 4.7(3): otherwise conversion is allowed, but the value is
    % implementation-defined

int_conversion(typ)(n_ex : ET[State, (range(typ))]) : ET[State, (range(int))] =
(
  n_ex ## lambda (n : (range(typ))) :
    ok_result(int_conversion(typ)(n))
)

i_typ : Var Cpp_Subtype(cv(non_bool_integral_enum?))

% range to PVS int
range_to_int(i_typ)(n_ex : ET[State, (range(i_typ))]) : ET[State, int] =
(
  n_ex ## lambda (n : (range(i_typ))) :
    ok_result(n)
)

% PVS int to range
int_to_range(i_typ)(n_ex : ET[State, int]) : ET[State, (range(i_typ))] =
(
  n_ex ## lambda (n : int) :
    If range(i_typ)(n) Then
      ok_result(n)
    Else
      fatal_result
    Endif
)

% 4.5(2): TODO (wchar_t, enumeration types)

% 4.5(3): TODO (bit-fields)

% 4.5(4)

bool_ex : VAR ET[State, bool]

% proof status :-)
% bool2int_TCC1 : TCC Obligation

% proof status :-)
% bool2int_TCC2 : TCC Obligation

bool2int(bool_ex) : ET[State, (range(int))] =

```


B PVS Theory Sources

```
(
  bool_ex ## lambda (b : bool) :
  ok_result(if b then 1 else 0 endif)
)
```

END IntegralPromotions

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% 4.7: Integral conversions
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

IntegralConversions[State : Type] : THEORY
BEGIN

IMPORTING Cpp_Types, Ok_Result_Rewrite

*% 4.7(1): "An rvalue of an integer type can be converted to an rvalue of
% another integer type."*

*% The integer types are bool, char, wchar_t, (signed/unsigned) char,
% (signed/unsigned) short, (signed/unsigned) int, (signed/unsigned) long
% (3.9.1(7)).*

```
% bool_ex : VAR [State -> ExprResult[State, Semantics_bool]]
% char_ex : VAR [State -> ExprResult[State, Semantics_char]]
% wchar_t_ex : VAR [State -> ExprResult[State, Semantics_wchar_t]]
% schar_ex : VAR [State -> ExprResult[State, Semantics_schar]]
% uchar_ex : VAR [State -> ExprResult[State, Semantics_uchar]]
% short_ex : VAR [State -> ExprResult[State, Semantics_short]]
% ushort_ex : VAR [State -> ExprResult[State, Semantics_ushort]]
% int_ex : VAR [State -> ExprResult[State, Semantics_int]]
% uint_ex : VAR [State -> ExprResult[State, Semantics_uint]]
% long_ex : VAR [State -> ExprResult[State, Semantics_long]]
% ulong_ex : VAR [State -> ExprResult[State, Semantics_ulong]]
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Conversions to bool: see 4.12
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Conversions to char
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
% bool2char(bool_ex) : [State -> ExprResult[State, Semantics_char]] =
% (
%   bool_ex ## lambda (b : Semantics_bool):
%   ok_result(if b then 1 else 0 endif)
% )
```

```
% TODO: wchar_t2char
% TODO: schar2char
% TODO: uchar2char
% TODO: short2char
% TODO: ushort2char
% TODO: int2char
% TODO: uint2char
% TODO: long2char
% TODO: ulong2char
```

```
%%%%%%%%%%
% Conversions to wchar_t
%%%%%%%%%%
```

```
% TODO
```

```
%%%%%%%%%%
% Conversions to schar
%%%%%%%%%%
```

```
% TODO
```

```
%%%%%%%%%%
% Conversions to uchar
%%%%%%%%%%
```

```
% TODO
```

```
%%%%%%%%%%
% Conversions to short
%%%%%%%%%%
```

```
% TODO
```

```
%%%%%%%%%%
% Conversions to ushort
%%%%%%%%%%
```

```
% TODO
```

```
%%%%%%%%%%
% Conversions to int
%%%%%%%%%%
```

```
% 4.7(5): "The conversions allowed as integral promotions are excluded from the
% set of integral conversions." Theory IntegralPromotions should therefore
% contain most of the necessary definitions already.
```

```
% TODO: long2int
% TODO: ulong2int
```

```
%%%%%%%%%%
```

B PVS Theory Sources

```
% Conversions to uint
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% bool2uint(booLex) : [State -> ExprResult[State, Semantics_uint]] =
% (
%   booLex ## lambda (b : Semantics_bool):
%     ok_result(if b then 1 else 0 endif)
% )

% TODO: wchar_t2uint
% TODO: schar2uint
% TODO: uchar2uint
% TODO: short2uint
% TODO: ushort2uint

% int2uint(int_ex) : [State -> ExprResult[State, Semantics_uint]] =
% (
%   int_ex ## lambda (i : Semantics_int):
%     ok_result(mod(i, max_value(uint))) % 4.7(2)
% )

% TODO: long2uint
% TODO: ulong2uint

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Conversions to long
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% TODO

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Conversions to ulong
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% TODO
```

END IntegralConversions

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% 4.12: Boolean conversions
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

BooleanConversions[State : **Type**] : **THEORY**
BEGIN

IMPORTING Abstract_Read_Write_Plain, Cpp_Types

s : **VAR** State

```
% TODO: there should probably be separate conversions for every arithmetic
% and pointer type
```

```
integral_to_bool(i_typ : Cpp_Subtype(cv(non_bool_integral_enum?)))
  (n_ex : ET[State, (range(i_typ))]) : ET[State, bool] =
(
  n_ex ## lambda (n : (range(i_typ))) :
  ok_result(not (n = 0))
)
```

```
% To do: floating_point_to_bool
```

```
% proof status :-)
% pointer_to_bool_TCC1 : TCC Obligation
```

```
pointer_to_bool(p_typ : Cpp_Subtype(cv(pointer?)))
  (p_ex : ET[State, (range_pointer(cv_base(p_typ)))] : ET[State, bool] =
(
  p_ex ## lambda (ptr_val : (range_pointer(cv_base(p_typ)))) :
  ok_result(not_null?(p_typ)(ptr_val))
)
```

END BooleanConversions

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% TODO: other conversions
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% Conversions: imports everything
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

Conversions[State : Type] : **THEORY**
BEGIN

```
IMPORTING LvalueToRvalueConversion,
           IntegralPromotions,
           IntegralConversions,
           BooleanConversions
```

END Conversions

B.9 cpp-examples.pvs

```
% $Id: cpp-examples.pvs,v 1.26.6.2.2.4.2.2 2008/05/14 14:24:58 tews Exp $
%
```

B PVS Theory Sources

```
% Author: Tjark Weber, Marcus Völp
% (c) 2007 Radboud University
%
% Some (rather simple) C++ programs and ((not yet) automatic) verification
% proofs.
```

```
Cpp_Examples[State : Type] : THEORY
BEGIN
```

```
IMPORTING Cpp_Verification[State]
```

```
STATES : PRED[State]
```

```
% skip
% proof status :-)
spec01 : LEMMA
  Valid(STATES,
    skip[State],
    STATES)
```

```
pm : Plain_Memory[State]
```

```
a, b, c, n, m : Address
```

```
% proof status :-)
% PRECONDITION_TCC1 : TCC Obligation
```

```
% proof status :-)
% PRECONDITION_TCC2 : TCC Obligation
```

```
% proof status :-)
% PRECONDITION_TCC3 : TCC Obligation
```

```
% proof status :-)
% PRECONDITION_TCC4 : TCC Obligation
```

```
PRECONDITION : PRED[State] =
```

```
  lambda (s : State):
```

```
    pm'states(s) and
```

```
      in_blessed_memory?(dt(int), n, pm'rw_addr) and
      in_blessed_memory?(dt(int), m, pm'rw_addr) and
      in_blessed_memory?(dt_bool, c, pm'rw_addr) and
      in_blessed_memory?(dt(char), a, pm'rw_addr) and
      in_blessed_memory?(dt(char), b, pm'rw_addr) And
      blocks_disjoint?(a, size_of(char), b, size_of(char)) And
      blocks_disjoint?(b, size_of(char), a, size_of(char)) And
      blocks_disjoint?(n, size_of(int), m, size_of(int)) And
      blocks_disjoint?(m, size_of(int), n, size_of(int))
```

```
% proof status :-)
% POSTCONDITION_TCC1 : TCC Obligation
```

```

POSTCONDITION : PRED[State] =
  lambda (s : State):
    let ss = read_data(pm, dt(int))(n)(s) in
      OK?(ss) and data(ss) = 42

% proof status :-)
% spec02_TCC1 : TCC Obligation

% a single assignment – partial correctness
% proof status :-)
spec02 : LEMMA
  plain_memory?(pm) implies
    valid(PRECONDITION,
      e2s(assign(pm, dt(int))(id(n), literal(42))),
      POSTCONDITION)

% a single assignment – total correctness
% proof status :-)
spec03 : LEMMA
  plain_memory?(pm) implies
    Valid(PRECONDITION,
      e2s(assign(pm, dt(int))(id(n), literal(42))),
      POSTCONDITION)

% No disjointness assumption should be necessary for this lemma.
% MV: Tjark: to remove the disjointness assumption we need
% sizeof(char) = 1 and something like:
% a /= b And sizeof(dt) = 1 => block_disjoint(a, b) And block_disjoint(b, a)
% Do we really need to check this automatically?

% proof status :-)
% spec04_TCC1 : TCC Obligation

% proof status :-)
% spec04_TCC2 : TCC Obligation

% two char assignments
% proof status :-)
spec04 : LEMMA
  plain_memory?(pm) implies
    valid(PRECONDITION,
      e2s(assign(pm, dt(char))(id(a), literal(42))) ##
      e2s(assign(pm, dt(char))(id(b), literal(42))),
      Lambda (s : State) : OK?(read_data(pm, dt(char))(a)(s)) And
      data(read_data(pm, dt(char))(a)(s)) = 42)

% if-else
% proof status :-)
spec05: LEMMA
  plain_memory?(pm) implies
    valid(PRECONDITION,
      e2s(assign(pm, dt_bool)(id(c), literal(true))) ##

```

B PVS Theory Sources

```
    if_else(l2r(pm, dt_bool)(id(c)),
            e2s(assign(pm, dt(int))(id(n), literal(42))),
            skip),
    POSTCONDITION)

% proof status :-)
% spec06_TCC1 : TCC Obligation

% proof status :-)
% spec06_TCC2 : TCC Obligation

% proof status :-)
% spec06_TCC3 : TCC Obligation

% proof status :-)
% spec06_TCC4 : TCC Obligation

% a switch statement (with several assignments)
% proof status :-)
spec06 : LEMMA
  plain_memory?(pm) implies
    valid(PRECONDITION,
          e2s(assign(pm, dt(char))(id(a), literal(0))) ##
            e2s(assign(pm, dt(char))(id(b), literal(1))) ##
            switch(l2r(pm, dt(char))(id(a)), (: 0, 1 :),
                  case(0) ##
                    skip ##
                  case(1) ##
                    e2s(assign(pm, dt(int))(id(n), literal(42))) ##
                    break ##
                  default ##
                    e2s(assign(pm, dt(int))(id(n), literal(0)))
            ),
          POSTCONDITION)

% a trivial infinite loop
% proof status :-)
spec07: LEMMA
  valid(PRECONDITION,
        while(literal(true),
              skip),
        lambda (s : State): false)

% a nontrivial loop

% proof status :-)
% N_TCC1 : TCC Obligation

N : {n : (range(int)) | n >= 0} = 2
```

```

% proof status :-)
% range_int_values_TCC1 : TCC Obligation

% proof status :-)
range_int_values : Lemma
  Forall (i : int) :
    -32767 <= i And i <= 32767
  Implies
    range(int)(i)

% proof status :-)
% spec08_TCC1 : TCC Obligation

% proof status :-)
spec08: LEMMA
plain_memory?(pm) implies
valid(PRECONDITION,
  e2s(assign(pm, dt(int))(id(n), literal(0))) ##
  e2s(assign(pm, dt(int))(id(m), literal(0))) ##
  while(N + 1, lt(int)(l2r(pm, dt(int))(id(n)), literal(N)),
    e2s(postinc(pm, int)(id(n))) ##
    e2s(postinc(pm, int)(id(m)))),
lambda (s : State) :
  OK?(read_data(pm, dt(int))(m)(s)) Implies
  data(read_data(pm, dt(int))(m)(s)) = N

% proof status :-)
spec09 : Lemma
plain_memory?(pm) implies
valid(PRECONDITION,
  e2s(assign(pm, dt(int))(id(m), literal(0))) ##
  for(
    N + 1,
    e2s(assign(pm, dt(int))(id(n), literal(0))),
    lt(int)(l2r(pm, dt(int))(id(n)), literal(N)),
    postinc(pm, int)(id(n)),
    e2s(postinc(pm, int)(id(m)))),
Lambda (s : State) :
  OK?(read_data(pm, dt(int))(m)(s)) Implies
  data(read_data(pm, dt(int))(m)(s)) = N

% proof status :-)
% spec10_TCC1 : TCC Obligation

%>>
% proof status :-( unfinished
spec10 : Lemma
plain_memory?(pm) implies
valid(PRECONDITION,
  for(
    e2s(assign(pm, dt(int))(id(n), literal(0))),
    lt(int)(l2r(pm, dt(int))(id(n)), literal(N)),

```



```

postinc(pm, int)(id(n)),
skip(
% P
(Lambda (s0 : State) : Lambda (res : StmtResult[State]) :
  OK?(res) And OK?(read_data(pm, dt(int))(n)(state(res))) And
  0 <= data(read_data(pm, dt(int))(n)(state(res))) And
  data(read_data(pm, dt(int))(n)(state(res))) < N
),
% Q
(Lambda (s0 : State) : Lambda (res : StmtResult[State]) :
  OK?(res) And OK?(read_data(pm, dt(int))(n)(state(res))) And
  data(read_data(pm, dt(int))(n)(state(res))) = N
),
% R
(Lambda (s0 : State) : Lambda (res : StmtResult[State]) :
  OK?(res) And OK?(read_data(pm, dt(int))(n)(state(res))) And
  data(read_data(pm, dt(int))(n)(state(res))) <= N
),
% variant
(Lambda (s0 : State) : Lambda (s : State) :
  If OK?(read_data(pm, dt(int))(n)(s)) And
  data(read_data(pm, dt(int))(n)(s)) <= N
  Then
  N - data(read_data(pm, dt(int))(n)(s))
  Else
  N
  Endif),
% order
Lambda (i, j : int) : i < j
),
Lambda (s : State) :
  OK?(read_data(pm, dt(int))(n)(s)) Implies
  data(read_data(pm, dt(int))(n)(s)) = N

```

END Cpp_Examples

```

% Rewrite: Hoare rules
% (auto-rewrite "valid" "Valid" "PRECONDITION" "POSTCONDITION")
% (smash)
% (skosimp* : preds? t)

```

```

% Required Axioms:
% size_of(typ)
% const and volatile models

```

```

% (auto-rewrite
% "address_of_spec" "address_of_spec2" "add_simplification"
% "address_add_zero" "address_add_sum"
% "add_spec" "add_spec2" "same_array_spec" "same_array_spec2"
% "mod_range" "size_of" "uid" "check_bounds_spec"
% "er_plus_ax" "er_minus_ax" "er_div_ax" "er_times_ax" "er_neg_ax"

```

```

% "extended_real_superset_of_number_fields")

% Rewrite: Expressions
% (auto-rewrite
% "literal" "id" "member" "postinc" "postdec" "deref" "arrow"
% "unary_minus" "unary_minus_unsigned" "unary_not" "unary_bitnot"
% "preinc" "predec" "UnaryExpressions.sizeof" "ptm_member" "ptm_arrow"
% "arrow" "times" "div" "div_float" "mod" "plus" "plus_ptr"
% "postinc_ptr" "preinc_ptr" "subscript" "minus" "minus_ptr"
% "postdec_ptr" "predec_ptr" "minus_ptr_ptr" "cmp" "lt" "gt" "le" "ge"
% "lt_ptr" "gt_ptr" "le_ptr" "ge_ptr"
% "cmp_pointer" "eq_bool" "eq" "not_equal" "not_equal_bool" "eq_ptr"
% "not_equal_ptr" "bitand" "bitxor" "bitor" "and_exp" "or_exp"
% "conditional_expr" "assign" "assign_times" "assign_div" "assign_mod"
% "assign_div_float" "assign_plus" "assign_plus_ptr" "assign_minus"
% "assign_minus_ptr" "assign_bitand" "assign_bitxor" "assign_bitor"
% "comma")

% Rewrites: Plain Memory
% (auto-rewrite
% "plain_memory_write_ok_single"
% "plain_memory_read_data_ok"
% "plain_memory_write_ok_q_expr"
% "plain_memory_write_ok_q_stmt"
% "in_blessed_memory_rw_ro"
% "pm_q_prop_ok"
% "pm_q_prop_single_read"
% "pm_q_prop_single_write"
% "pm_q_prop_read_write_single"
% "pm_q_prop_read_write_other_single"
% "pm_q_prop_read_read_single"
% "plain_memory_read_write_other_single_data"
% "plain_memory_read_read"
% "plain_memory_read_write_res"
% "pm_q_prop_read_ok_expr"
% "pm_q_prop_read_ok_stmt"
% "pm_q_prop_read_pm_q_prop_expr"
% "pm_q_prop_read_pm_q_prop_stmt"
% "pm_q_prop_read_write_q_expr"
% "pm_q_prop_read_write_q_stmt"
% "plain_memory_read_write_q_data_expr"
% "plain_memory_read_write_q_data_stmt"
% "pm_q_prop_read_write_other_q_expr"
% "pm_q_prop_read_write_other_q_stmt"
% "pm_q_prop_read_read_q_expr"
% "pm_q_prop_read_read_q_stmt"
% "plain_memory_read_write_other_q_data_expr"
% "plain_memory_read_write_other_q_data_stmt"
% "plain_memory_read_read_q_data_expr"
% "plain_memory_read_read_q_data_stmt"
% "pm_q_prop_ok_result_single"
% "pm_q_prop_ok_result_q_expr"

```

B PVS Theory Sources

```
% "pm_q_prop_ok_result_q_stmt"  
% "pm_q_prop_read_ok_result_single"  
% "pm_q_prop_read_ok_result_q_expr"  
% "pm_q_prop_read_ok_result_q_stmt"  
% "pm_q_prop_e2s"  
% "pm_q_prop_stmt_e2s"  
% "pm_q_prop_read_e2s"  
% "pm_q_prop_read_stmt_e2s"  
% "pm_q_prop_read_ok_pm_q_prop_read_expr"  
% "pm_q_prop_read_ok_pm_q_prop_read_stmt"  
% )
```

```
% Rewrite: Switch  
% (auto-rewrite  
% "if_else"  
% "switch"  
% "member"  
% "break_catch_break"  
% "break_break_stmt"  
% "break_break_catch_continue"  
% "break_break_catch_default"  
% "break_break_lift_stmt"  
% "break_break_case"  
% "break_break_default"  
% "break_break"  
% "stmt_break_break"  
% "break_stmt_break"  
% "break_stmt_catch_continue"  
% "break_stmt_catch_default"  
% "break_stmt_lift"  
% "break_stmt_case"  
% "break_stmt_default"  
% "continue_catch_continue"  
% "continue_break_stmt"  
% "continue_break_catch_continue"  
% "continue_break_catch_default"  
% "continue_break_lift_stmt"  
% "continue_break_case"  
% "continue_break_default"  
% "continue_continue"  
% "stmt_continue_continue"  
% "continue_stmt_continue"  
% "continue_stmt_catch_continue"  
% "continue_stmt_catch_default"  
% "continue_stmt_lift"  
% "continue_stmt_case"  
% "continue_stmt_default"  
% "switch_stmt"  
% "stmt_switch_stmt"  
% "switch_case_taken"  
% "stmt_switch_case_taken"  
% "switch_case_not_taken"
```

```

% "stmt_switch_case_not_taken"
% "switch_default"
% "stmt_switch_default"
% "switch_catch_break"
% "stmt_switch_catch_break"
% "switch_catch_continue"
% "stmt_switch_catch_continue"
% "switch_catch_default"
% "stmt_switch_catch_default"
% "default_stmt"
% "stmt_default_stmt"
% "default_case"
% "stmt_default_case"
% "default_default"
% "stmt_default_default"
% "default_catch_break"
% "stmt_default_catch_break"
% "default_catch_continue"
% "stmt_default_catch_continue"
% "default_catch_default"
% "stmt_default_catch_default"
% "return_void_result"
% "return_ex_result"
% "stmt_remove_catch_default"
% "stmt_remove_case"
% "stmt_remove_default"
% "stmt_remove_catch_continue"
% "stmt_remove_catch_break"
% "break_break_stmt_expr"
% "break_break_catch_continue_expr"
% "break_break_catch_default_expr"
% "break_break_lift_stmt_expr"
% "break_break_case_expr"
% "break_break_default_expr"
% "continue_break_stmt_expr"
% "continue_break_catch_continue_expr"
% "continue_break_catch_default_expr"
% "continue_break_lift_stmt_expr"
% "continue_break_case_expr"
% "continue_break_default_expr"
% )

% Rewrite: Skip / E2S
% (auto-rewrite
% "skip_ok"
% "skip_state"
% "skip_elimination"
% "stmt_skip_elimination"
% "stmt_ok_lift"
% "stmt_ok_lift_expr"
% "e2s_ok"
% "stmt_e2s_ok"

```

B PVS Theory Sources

```
% "e2s_state"
% "stmt_e2s_state"
% "ok_result_fexpr"
% "ok_result_fstmt"
% "e2s_merge"
% "stmt_e2s_merge"
% "e2s_expr"
% "stmt_e2s_expr"
% )

% Rewrite: Loops by unrolling
% (auto-rewrite
% "while_unroll"
% "stmt_while_unroll"
% "iterate_while_ok"
% "while_hang"
% "while_unroll_expr"
% "while_unroll_lexpr"
% "stmt_while_unroll_expr"
% "stmt_while_unroll_lexpr"
% "while_unroll_0"
% "do_while_unroll"
% "do_while_inv_unroll"
% "for_unroll"
% "for_inv_unroll"
% "while_to_while_no_cb"
% "while_invariant?!"
% "while_variant?!"
% "while_inv_rewrite_termination_result!"
% "while_inv_rewrite_data_ok!"
% "while_inv_rewrite_data_break!"
% "while_inv_rewrite_data_return!"
% "stmt_while_inv_rewrite_termination_result!"
% "stmt_while_inv_rewrite_data_ok!"
% "stmt_while_inv_rewrite_data_break!"
% "stmt_while_inv_rewrite_data_return!"
% "pm_q_prop_while!"
% "pm_q_prop_stmt_while!"
% )

% Rewrite: Composition
% (auto-rewrite
% "composition_assoc_rewrite_stmt_ok"
% "forward_s_c_s_of_s_expr"
% "comp_eval_if_ok_fstmt"
% "stmt_eval_if_ok_fstmt"
% "comp_eval_if_ok_fstmt_cstmt"
% "stmt_eval_if_ok_fstmt_cstmt"
% "composition_assoc_rewrite_1"
% "composition_assoc_rewrite_2"
```

```

% "composition_assoc_rewrite_3"
% "composition_assoc_rewrite_4"
% "composition_assoc_rewrite_5"
% "composition_assoc_rewrite_6"
% "composition_assoc_rewrite_7"
% "composition_assoc_rewrite_8"
% "composition_assoc_rewrite_expr_1"
% "composition_assoc_rewrite_expr_2"
% "composition_assoc_rewrite_expr_3"
% "composition_assoc_rewrite_expr_4"
% "composition_assoc_rewrite_expr_5"
% "composition_assoc_rewrite_expr_6"
% "composition_assoc_rewrite_stmt_ok"
% "composition_assoc_rewrite_stmt_ok_expr"
% "composition_assoc_rewrite_stmt_ok_expr_lexpr"
% "comp_eval_if_ok_fstmt"
% "comp_eval_if_ok_fstmt_expr"
% "comp_eval_if_ok_fstmt_cstmt_expr"
% "stmt_eval_if_ok_fstmt_expr"
% "stmt_eval_if_ok_fstmt_cstmt_expr"
% "comp_eval_if_ok_fexpr_expr"
% "comp_eval_if_ok_fexpr"
% "stmt_eval_if_ok_fexpr"
% "expr_eval_if_ok_fexpr"
% "composition_assoc_expression_rewrite"
% "composition_assoc_expression_rewrite_stmt"
% "stmt_eval_if_ok_fexpr_expr"
% "expr_eval_if_ok_fexpr_expr"
% "ok_result_fexpr"
% "ok_result_fstmt"
% "composition_assoc_rewrite_stmt_ok_cstmt"
% )

% Rewrite: Allocation
% (auto-rewrite
% "allocate_stack" "deallocate_stack" "with_new_stackvar"
% )

% Rewrite: Ok Result
% (auto-rewrite
% "ok_result_ok"
% "ok_result_data"
% "ok_result_q_ok"
% "ok_result_q_state"
% "ok_result_q_data"
% "ok_result_elimination_1"
% "ok_result_elimination_2"
% "ok_result_state"
% )

% Rewrite: Conversions
% (auto-rewrite

```

B PVS Theory Sources

```
% "I2r" "integral_to_bool" "pointer_to_bool" "bool2int"  
% )
```

```
% Rewrite: Precond rules  
% (auto-rewrite  
% "comp_simple_expr_simple_expr_ok"  
% "comp_simple_expr_simple_expr_data"  
% "comp_simple_expr_simple_expr_state"  
% "comp_simple_stmt_simple_expr_ok"  
% "comp_simple_stmt_simple_expr_data"  
% "comp_simple_stmt_simple_expr_state"  
% )
```

B.10 cpp-verification.pvs

```
% $Id: cpp-verification.pvs,v 1.2.6.1.4.1 2008/05/13 14:47:26 tews Exp $  
%  
% Author: Tjark Weber  
% (c) 2007 Radboud University  
%  
% Verification environment for C++. Main theory, imports all other theories  
% that are needed to verify C++ programs in PVS.
```

```
Cpp_Verification[State : Type] : THEORY  
BEGIN
```

```
    Importing Expressions, Conversions, Hoare, Statement_Rewrites
```

```
    pm : Plain_Memory[State]
```

```
END Cpp_Verification
```

B.11 datatype_model.pvs

```
% $Id: datatype_model.pvs,v 1.1.6.2.4.2 2008/05/14 14:24:58 tews Exp $  
%  
% Author: Tjark Weber  
% (c) 2007 Radboud University  
%  
% A definition of two PODs with different "type tags", where  
% read_data(pm, pod_2)(addr) after write_data(pm, pod_1)(addr) fails.  
%  
% This is merely an example.  
%  
% The contents of this file should probably be moved to "builtin_models.pvs"  
% when the latter typechecks again.  
  
% Note that there is (yet another) problem with the current POD formalization:  
% it does not guarantee that PODs can be copied via char arrays.  
  
% Two trivial PODs, both with unit semantics.
```

Datatype_Model : **THEORY**
BEGIN

IMPORTING Unit, Interpreted_Data[Unit]

% proof status :-)

% uidt_1_TCC1 : TCC Obligation

uidt_1 : (uninterpreted_data_type?) =
 (# size := 1,
 valid? := **LAMBDA** (bl: list[Byte], a : Address):
 bl = (: 0 :)
 #)

% proof status :-)

% pod_1_TCC1 : TCC Obligation

% proof status :-)

% pod_1_TCC2 : TCC Obligation

% proof status :-)

% pod_1_TCC3 : TCC Obligation

% proof status :-)

% pod_1_TCC4 : TCC Obligation

pod_1 : (pod_data_type?) =
 (# uidt := uidt_1,
 to_byte := **LAMBDA** (d: Unit, a : Address):
 (: 0 :),
 to_mask := **Lambda** (d : Unit, a : Address) :
 zero_mask?(size(uidt_1)),
 from_byte := **LAMBDA** (bl: list[Byte], a : Address):
 if bl = (: 0 :) **then** up(unit) **else** bottom **endif**
 #)

% proof status :-)

% uidt_2_TCC1 : TCC Obligation

uidt_2 : (uninterpreted_data_type?) =
 (# size := 1,
 valid? := **LAMBDA** (bl: list[Byte], a : Address):
 bl = (: 1 :)
 #)

% proof status :-)

% pod_2_TCC1 : TCC Obligation

% proof status :-)

% pod_2_TCC2 : TCC Obligation

B PVS Theory Sources

```
% proof status :-)
% pod_2_TCC3 : TCC Obligation

pod_2 : (pod_data_type?) =
  (# uidt := uidt_2,
   to_byte := LAMBDA (d: Unit, a : Address):
     (: 1 :),
   to_mask := Lambda (d : Unit, a : Address) :
     zero_mask?(size(uidt_2)),
   from_byte := LAMBDA (bl: list[Byte], a : Address):
     if bl = (: 1 :) then up(unit) else bottom endif
  #)

END Datatype_Model

Read_After_Write_Fails[State : Type] : THEORY
BEGIN

IMPORTING Datatype_Model, Statements, Plain_Mem_Properties[State]

pm : VAR Plain_Memory[State]
s : VAR State
a : VAR Address

% proof status :-)
% plain_mem_write_list_states_TCC1 : TCC Obligation

% proof status :-)
plain_mem_write_list_states : LEMMA
Forall(size : nat, bl : list[Byte]) :
  size = length(bl) and
  plain_memory?(pm) and
  pm'states(s) and
  subset?(address_block(a, size), pm'rw_addr) and
  OK?(memory_write_list(pm'mem)(a, bl)(s)) implies
  pm'states(state(memory_write_list(pm'mem)(a, bl)(s)))

% proof status :-)
plain_mem_write_list_read_list_ok : LEMMA
Forall(size : nat, bl : list[Byte]) :
  size = length(bl) and
  plain_memory?(pm) and
  pm'states(s) and
  subset?(address_block(a, size), pm'rw_addr) and
  OK?(memory_write_list(pm'mem)(a, bl)(s)) implies
  OK?(memory_read_list(pm'mem)(a, size)
    (state(memory_write_list(pm'mem) (a, bl)(s))))

% proof status :-)
Fatal_Result : LEMMA
  plain_memory?(pm) and
  pm'states(s) and
```

```

in_blessed_memory?(pod_1, a, pm'rw_addr) implies
  Fatal?( (write_data(pm, pod_1)(a, unit) ##
           read_data(pm, pod_2)(a))(s) )

```

END Read_After_Write_Fails

% The same for two perhaps more interesting PODs: bool and Byte.

Datatype_Model_2 : **THEORY**
BEGIN

IMPORTING Interpreted_Data[bool]

% proof status :-)
% uidt_bool_TCC1 : TCC Obligation

```

uidt_bool : (uninterpreted_data_type?) =
  (# size := 2,
   valid? := LAMBDA (bl: list[Byte], a : Address):
     bl = (: 0, 0 :) or bl = (: 0, 1 :)
  #)

```

% proof status :-)
% pod_bool_TCC1 : TCC Obligation

% proof status :-)
% pod_bool_TCC2 : TCC Obligation

% proof status :-)
% pod_bool_TCC3 : TCC Obligation

% proof status :-)
% pod_bool_TCC4 : TCC Obligation

% proof status :-)
% pod_bool_TCC5 : TCC Obligation

% proof status :-)
% pod_bool_TCC6 : TCC Obligation

```

pod_bool : (pod_data_type?) =
  (# uidt := uidt_bool,
   to_byte := LAMBDA (d: bool, a : Address):
     if d then (: 0, 1 :) else (: 0, 0 :) endif,
   to_mask := Lambda (d : bool, a : Address) :
     zero_mask?(size(uidt_bool)),
   from_byte := LAMBDA (bl: list[Byte], a : Address):
     if bl = (: 0, 1 :) then up(true)
     else if bl = (: 0, 0 :) then up(false)
     else bottom endif endif
  #)

```

```

IMPORTING Interpreted_Data[Byte]

% proof status :-)
% uidt_Byte_TCC1 : TCC Obligation

% proof status :-)
% uidt_Byte_TCC2 : TCC Obligation

uidt_Byte : (uninterpreted_data_type?) =
  (# size := 2,
    valid? := LAMBDA (bl: list[Byte], a : Address):
      length(bl) = 2 and car(bl) = 1
  #)

% proof status :-)
% pod_Byte_TCC1 : TCC Obligation

% proof status :-)
% pod_Byte_TCC2 : TCC Obligation

% proof status :-)
% pod_Byte_TCC3 : TCC Obligation

% proof status :-)
% pod_Byte_TCC4 : TCC Obligation

% proof status :-)
% pod_Byte_TCC5 : TCC Obligation

pod_Byte : (pod_data_type?[Byte]) =
  (# uidt := uidt_Byte,
    to_byte := LAMBDA (d: Byte, a : Address):
      (: 1, d :),
    to_mask := Lambda (d : Byte, a : Address) :
      zero_mask?(size(uidt_Byte)),
    from_byte := LAMBDA (bl: list[Byte], a : Address):
      if length(bl) = 2 and car(bl) = 1 then up(car(cdr(bl))) else bottom endif
  #)

```

END Datatype_Model_2

Read_After_Write_Fails_2[State : **Type**] : **THEORY**
BEGIN

IMPORTING Datatype_Model_2, Statements, Plain_Mem_Properties[State]

pm : **VAR** Plain_Memory[State]
s : **VAR** State
a : **VAR** Address

```

% proof status :-)
plain_mem_write_list_states : LEMMA
  Forall(size : nat, bl : list[Byte]) :
    size = length(bl) and
    plain_memory?(pm) and
    pm'states(s) and
    subset?(address_block(a, size), pm'rw_addr) and
    OK?(memory_write_list(pm'mem)(a, bl)(s)) implies
      pm'states(state(memory_write_list(pm'mem)(a, bl)(s)))

% proof status :-)
plain_mem_write_list_read_list_ok : LEMMA
  Forall(size : nat, bl : list[Byte]) :
    size = length(bl) and
    plain_memory?(pm) and
    pm'states(s) and
    subset?(address_block(a, size), pm'rw_addr) and
    OK?(memory_write_list(pm'mem)(a, bl)(s)) implies
      OK?(memory_read_list(pm'mem)(a, size)
        (state(memory_write_list(pm'mem) (a, bl)(s))))

% proof status :-)
Fatal_Result : LEMMA
  plain_memory?(pm) and
  pm'states(s) and
  in_blessed_memory?(pod_bool, a, pm'rw_addr) implies
    Fatal?( (write_data(pm, pod_bool)(a, true) ##
      read_data(pm, pod_Byte)(a))(s) )

END Read_After_Write_Fails_2

```

B.12 device_memory.pvs

```

Expand_State[State, T : Type] : Theory
Begin

```

```

  Importing State_Transformer

```

```

Expand_state : Type = [#
  state : State,
  expansion : T
#]

```

```

em_lift(stf : [State -> SuperResult[State]]) :
  [Expand_state -> SuperResult[Expand_state]] =
  Lambda (s : Expand_state) :
    Let res = stf(s'state) In
    Cases res Of
      OK(state) : OK(s With [(state) := state]),
      Abnormal(state) : Abnormal(s With [(state) := state]),
      Bottom : Bottom

```

EndCases

```

em_lift(P : PRED[State]) : PRED[Expand_state] =
  Lambda (em : Expand_state) : P(em'state)

em_lift(P : PRED[[State -> SuperResult[State]]])
  (q : [Expand_state -> SuperResult[Expand_state]]) : bool =
  Exists (p : (P)) : q = em_lift(p)

states : Var PRED[State]
transformers : Var PRED[[State -> SuperResult[State]]]

% proof status :-)
em_transformers_ok : Lemma
  transformers_ok?(states, transformers) Implies
    transformers_ok?(em_lift(states), em_lift(transformers))

% proof status :-)
em_transformer_invariant : Lemma
  transformer_invariant?(states, transformers) Implies
    transformer_invariant?(em_lift(states), em_lift(transformers))

% proof status :-)
em_lift_singleton : Lemma
  Forall (q : [State -> SuperResult[State]]) :
    singleton(em_lift(q)) = em_lift(singleton(q))

End Expand_State

Expand_State2[State, T, Data : Type] : Theory
Begin

  Importing Expand_State[State, T]

  em_lift(stf : [State -> ExprResult[State, Data]]) :
    [Expand_state -> ExprResult[Expand_state, Data]] =
    Lambda (s : Expand_state) :
      Let res = stf(s'state) In
        Cases res Of
          OK(state, data) : OK(s With [(state) := state], data),
          Exception(ex_type, state) : Exception(ex_type, s With [(state) := state]),
          Fatal : Fatal,
          Hang : Hang
        EndCases

  % proof status :-)
  em_lift_expr_2_super : Lemma
    Forall (stf : [State -> ExprResult[State, Data]]) :
      expr_2_super(em_lift(stf)) = em_lift(expr_2_super(stf))

End Expand_State2

```

```
Expand_State3[State, T : Type] : Theory
Begin
```

```
Importing Expand_State2, Plain_Memory
```

```
pm : Var Memory_struct[State]
em_pm : Var Memory_struct[Expand_state[State, T]]
```

```
states : Var PRED[State]
addresses : Var PRED[Address]
transformers : Var PRED[[State -> SuperResult[State]]]
```

```
% proof status :-)
em_memory_read_transformers : Lemma
  (em_pm'memory_read =
    Lambda (a : Address) : em_lift[State, T, Byte](pm'memory_read(a)))
```

```
Implies
```

```
memory_read_transformers(em_pm, addresses) =
  em_lift(memory_read_transformers(pm, addresses))
```

```
% proof status :-)
em_memory_write_transformers : Lemma
  (em_pm'memory_write =
    Lambda (a : Address, b : Byte) :
      em_lift[State, T, Unit](pm'memory_write(a, b)))
```

```
Implies
```

```
memory_write_transformers(em_pm, addresses) =
  em_lift(memory_write_transformers(pm, addresses))
```

```
% proof status :-)
em_unchanged_memory_invariant : Lemma
  (em_pm'memory_read = Lambda (a : Address) : em_lift[State, T, Byte](pm'memory_read(a))) And
  unchanged_memory_invariant?(pm, states, transformers, addresses)
```

```
Implies
```

```
unchanged_memory_invariant?(em_pm, em_lift(states),
  em_lift(transformers), addresses)
```

```
% proof status :-)
em_changed_memory_invariant : Lemma
  (em_pm'memory_read =
    Lambda (a : Address) : em_lift[State, T, Byte](pm'memory_read(a))) And
  (em_pm'memory_write =
    Lambda (a : Address, b : Byte) :
      em_lift[State, T, Unit](pm'memory_write(a, b))) And
  changed_memory_invariant?(pm, states, addresses)
```

```
Implies
```

```
changed_memory_invariant?(em_pm, em_lift(states), addresses)
```

```
% proof status :-)
% em_const_unchanged_memory_invariant_TCC1 : TCC Obligation
```

```
% proof status :-)
```

```

em_const_unchanged_memory_invariant : Lemma
  Forall (q : [Expand_state[State, T] -> SuperResult[Expand_state[State, T]]) :
    (em_pm'memory_read = Lambda (a : Address) : em_lift[State, T, Byte](pm'memory_read(a))) And
    (Forall (s : Expand_state[State, T]) : has_next_state(q(s)) Implies state(q(s))'state = s'state)
  Implies
    unchanged_memory_invariant?(em_pm, em_lift(states), singleton(q), addresses)

```

```

se_transformer : Var
  [Address, list[Byte], bool -> [State -> ExprResult[State, list[Byte]]]]

```

% proof status :-)

```

em_side_effect_content_unchanged : Lemma
  side_effect_content_unchanged(addresses, states, se_transformer)
Implies
  side_effect_content_unchanged(addresses, em_lift(states),
    Lambda (a : Address, bl : list[Byte], cp : bool) :
      em_lift[State, T, list[Byte]](se_transformer(a, bl, cp)))

```

End Expand_State3

```

Device_Memory [Physical_memory : Type,
  (Importing Plain_Memory[Physical_memory])
  pm : Plain_Memory,
  Device_state : Type] : Theory

```

Begin

Importing Expand_State3

Device_memory : **Type** = Expand_state[Physical_memory, Device_state]

```

s : Var Device_memory
a : Var Address
b : Var Byte
size : Var nat

```

```

device_read_side_effect : Var
  [Address, list[Byte], bool ->
    [Device_memory -> ExprResult[Device_memory, list[Byte]]]]
device_write_side_effect : Var
  [Address, list[Byte], bool ->
    [Device_memory -> ExprResult[Device_memory, list[Byte]]]]

```

```

device_pm(device_read_side_effect,
  device_write_side_effect) : Memory_struct[Device_memory] =
(#
  memory_read := Lambda (a : Address) :
    em_lift[Physical_memory, Device_state, Byte](memory_read(pm'mem)(a)),
  memory_write := Lambda (a : Address, b : Byte) :
    em_lift[Physical_memory, Device_state, Unit](memory_write(pm'mem)(a, b)),
  memory_read_side_effect :=
    Lambda (a : Address, bl : list[Byte], cp : bool) :

```

```

em_lift[Physical_memory, Device_state, list[Byte]](
  memory_read_side_effect(pm'mem)(a, bl, cp)) ## Lambda (bl1 : list[Byte]) :
device_read_side_effect(a, bl1, cp),
memory_write_side_effect :=
  Lambda (a : Address, bl : list[Byte], cp : bool) :
  em_lift[Physical_memory, Device_state, list[Byte]](
    memory_write_side_effect(pm'mem)(a, bl, cp)) ## Lambda (bl1 : list[Byte]) :
  device_write_side_effect(a, bl1, cp)
#)

```

End Device_Memory

```

Challenge_Device_Memory [Physical_memory : Type,
  (Importing Plain_Memory[Physical_memory])
  pm_phy : Plain_Memory,
  Device_state : Type] : Theory

```

Begin

```

Importing Device_Memory[Physical_memory, pm_phy, Device_state],
  Plain_Mem_Rewrites

```

```

device_read_side_effect : Var
  [Address, list[Byte], bool ->
    [Device_memory -> ExprResult[Device_memory, list[Byte]]]]
device_write_side_effect : Var
  [Address, list[Byte], bool ->
    [Device_memory -> ExprResult[Device_memory, list[Byte]]]]

```

```

addresses : Var PRED[Address]
pm : Var Plain_Memory[Device_memory]

```

```

drse_super_transformers(addresses, device_read_side_effect) :
  PRED[[Device_memory -> SuperResult[Device_memory]]] =
  Lambda (q : [Device_memory -> SuperResult[Device_memory]]) :
  Exists (a : Address, bl : list[Byte], cp : bool) :
    subset?(address_block(a, length(bl)), addresses) And
    q = expr_2_super(device_read_side_effect(a, bl, cp))

```

```

dwse_super_transformers(addresses, device_write_side_effect) :
  PRED[[Device_memory -> SuperResult[Device_memory]]] =
  Lambda (q : [Device_memory -> SuperResult[Device_memory]]) :
  Exists (a : Address, bl : list[Byte], cp : bool) :
    subset?(address_block(a, length(bl)), addresses) And
    q = expr_2_super(device_write_side_effect(a, bl, cp))

```

```

is_device_plain_memory?(device_read_side_effect, device_write_side_effect)
  (pm) : bool =
  % plain memory
  pm'mem = device_pm(device_read_side_effect, device_write_side_effect) And
  plain_memory?(pm_phy) And

```



```

pm'states = em_lift(pm_phy'states) And
subset?(pm'other_actions, em_lift(pm_phy'other_actions)) And
subset?(pm'rw_addr, pm_phy'rw_addr) And
subset?(pm'ro_addr, union(pm_phy'ro_addr, difference(pm_phy'rw_addr, pm'rw_addr))) And
% side effect properties
transformers_ok?(pm'states, union(
  drse_super_transformers(union(pm'ro_addr, pm'rw_addr),
    device_read_side_effect),
  dwse_super_transformers(pm'rw_addr, device_write_side_effect))) And
unchanged_memory_invariant?(pm'mem, pm'states, union(
  drse_super_transformers(union(pm'ro_addr, pm'rw_addr),
    device_read_side_effect),
  dwse_super_transformers(pm'rw_addr, device_write_side_effect)),
  union(pm'ro_addr, pm'rw_addr)) And
side_effect_content_unchanged(union(pm'ro_addr, pm'rw_addr), pm'states,
  device_read_side_effect) And
side_effect_content_unchanged(pm'rw_addr, pm'states,
  device_write_side_effect)

% Helper Lemmas
%-----
% proof status :- )
pm_dev_plain : Lemma
  is_device_plain_memory?(device_read_side_effect,
    device_write_side_effect)(pm)

Implies
  plain_memory?(pm_phy)

% proof status :- )
pm_dev_states : Lemma
  is_device_plain_memory?(device_read_side_effect,
    device_write_side_effect)(pm)

Implies
  pm'states = em_lift(pm_phy'states)

% proof status :- )
pm_dev_read : Lemma
  is_device_plain_memory?(device_read_side_effect,
    device_write_side_effect)(pm)

Implies
  memory_read(pm'mem) =
    Lambda (a : Address) :
      em_lift[Physical_memory, Device_state, Byte](memory_read(pm_phy'mem)(a))

% proof status :- )
pm_dev_write : Lemma
  is_device_plain_memory?(device_read_side_effect,
    device_write_side_effect)(pm)

Implies
  memory_write(pm'mem) =
    Lambda (a : Address, b : Byte) :
      em_lift[Physical_memory, Device_state, Unit](memory_write(pm_phy'mem)(a, b))

```

```

% proof status :-)
pm_dev_read_side_effect : Lemma
  is_device_plain_memory?(device_read_side_effect,
                          device_write_side_effect)(pm)

Implies
  memory_read_side_effect(pm'mem) =
    Lambda (a : Address, bl : list[Byte], cp : bool) :
      em_lift[Physical_memory, Device_state, list[Byte]](
        memory_read_side_effect(pm_phy'mem)(a, bl, cp)) ## lambda (bl1 : list[Byte]) :
        device_read_side_effect(a, bl1, cp)

% proof status :-)
pm_dev_write_side_effect : Lemma
  is_device_plain_memory?(device_read_side_effect,
                          device_write_side_effect)(pm)

Implies
  memory_write_side_effect(pm'mem) =
    Lambda (a : Address, bl : list[Byte], cp : bool) :
      em_lift[Physical_memory, Device_state, list[Byte]](
        memory_write_side_effect(pm_phy'mem)(a, bl, cp)) ## lambda (bl1 : list[Byte]) :
        device_write_side_effect(a, bl1, cp)

% proof status :-)
pm_dev_ro_rw_addr : Lemma
  is_device_plain_memory?(device_read_side_effect,
                          device_write_side_effect)(pm)

Implies
  subset?(union(pm'ro_addr, pm'rw_addr), union(pm_phy'ro_addr, pm_phy'rw_addr))

% Plain Memory Properties
%-----

% proof status :-)
device_plain_transformers_ok : Lemma
  is_device_plain_memory?(device_read_side_effect,
                          device_write_side_effect)(pm)

Implies
  transformers_ok?(pm'states, union(
    memory_read_transformers(pm'mem, union(pm'ro_addr, pm'rw_addr)),
    memory_write_transformers(pm'mem, pm'rw_addr)))

% proof status :-)
device_plain_unchanged_memory_invariant : Lemma
  is_device_plain_memory?(device_read_side_effect,
                          device_write_side_effect)(pm)

Implies
  unchanged_memory_invariant?(pm'mem, pm'states, union(union(pm'other_actions,
    memory_read_transformers(pm'mem, union(pm'ro_addr, pm'rw_addr))),
    union(memory_read_side_effect_super_transformers(pm'mem,
      union(pm'ro_addr, pm'rw_addr)),
    memory_write_side_effect_super_transformers(pm'mem, pm'rw_addr))),

```

```

union(pm'ro_addr, pm'rw_addr))

% proof status :-)
device_plain_unchanged_memory_invariant_write : Lemma
  is_device_plain_memory?(device_read_side_effect,
                          device_write_side_effect)(pm)

Implies
  unchanged_memory_invariant?(pm'mem, pm'states,
                               memory_write_transformers(pm'mem, pm'rw_addr), pm'ro_addr)

% proof status :-)
device_plain_unchanged_memory_write_invariant : Lemma
  is_device_plain_memory?(device_read_side_effect,
                          device_write_side_effect)(pm)

Implies
  unchanged_memory_write_invariant?(pm'mem, pm'states, pm'rw_addr)

% proof status :-)
device_plain_changed_memory_invariant : Lemma
  is_device_plain_memory?(device_read_side_effect,
                          device_write_side_effect)(pm)

Implies
  changed_memory_invariant?(pm'mem, pm'states, pm'rw_addr)

% proof status :-)
device_plain_read_side_effect_unchanged : Lemma
  is_device_plain_memory?(device_read_side_effect,
                          device_write_side_effect)(pm)

Implies
  side_effect_content_unchanged(union(pm'ro_addr, pm'rw_addr), pm'states,
                                 memory_read_side_effect(pm'mem))

% proof status :-)
device_plain_write_side_effect_unchanged : Lemma
  is_device_plain_memory?(device_read_side_effect,
                          device_write_side_effect)(pm)

Implies
  side_effect_content_unchanged(pm'rw_addr, pm'states,
                               memory_write_side_effect(pm'mem))

% proof status :-)
device_plain_transformers_ok2 : Lemma
  is_device_plain_memory?(device_read_side_effect,
                          device_write_side_effect)(pm)

Implies
  transformers_ok?(pm'states, union(
    memory_read_side_effect_super_transformers(pm'mem,
                                                union(pm'ro_addr, pm'rw_addr)),
    memory_write_side_effect_super_transformers(pm'mem, pm'rw_addr)))

% Main Blessing Result
%=====

```

```

% proof status :-)
device_memory_plain_memory : Lemma
  Forall (pm) :
    is_device_plain_memory?(device_read_side_effect,
                             device_write_side_effect)(pm)
    Implies plain_memory?(pm)

```

End Challenge_Device_Memory

B.13 everything.pvs

Import_All : **Theory**

```

% This theory imports (directly or indirectly) *every* theory
% of the project. This way one can conveniently rerun all proofs
% with prove-importchain (C-c C-p i) on this theory.

```

Begin

```

% Please add all new theories here !!

```

```

Importing Phy_Mem_Challenge_Read_Same, Phy_Mem_Challenge_Read_Other,
  Linear_Memory_Blessing_Properties,
  Statement_Rewrites,
  Allocation_Table, Challenge_Random_Device, Expressions

```

```

% data type models

```

```

Importing PDBR_Data_Model, PDE_Model, PTE_Model, PF_EC_Model, Address_Model

```

```

% test theories

```

```

Importing Rewrite_Test, Search_Example, Cpp_Examples

```

```

Importing Read_After_Write_Fails, Read_After_Write_Fails_2

```

```

% intermediate stuff, that will disappear later

```

```

Importing PDBR_Datatype, PDE_Datatype, PTE_Datatype,
  Physical_Memory_Corollaries, Block_Disjoint_Rewrites,
  Posnat_induction, Set_Lift

```

End Import_All

B.14 expressions.pvs

```

% $Id: expressions.pvs,v 1.34.2.2.2.5.2.4 2008/05/14 14:24:58 tews Exp $

```

```

%

```

```

% Author: Tjark Weber (based on lecture material by Hendrik Tews)

```

```

% (c) 2007 Radboud University

```

```

%

```

```

% A shallow embedding of (some) C++ expressions as state transformers.

```

```

% Section numbers in comments throughout this file refer to

```

```

% "Programming languages – C++", ISO/IEC 14882:1998(E)

```

B PVS Theory Sources

```
% Deviations from the C++ standard:
%=====
%
% Evaluation Order
%-----
% Note that the C++ standard leaves the evaluation order of subexpressions
% mostly unspecified. The behavior is undefined unless each expression
% satisfies a (somewhat obscure) side condition (cf. 5.4). We do not model
% this here; the following definitions instead use some fixed evaluation order.
% Michael Norrish in his paper
% "Deterministic expressions in C", 8th European Symposium on Programming
% (ETAPS), Amsterdam. 1999. LNCS 1576, pp147–161
% shows that the evaluation order is actually irrelevant for expressions that
% satisfy the side condition of 5.4. At the source code level, the evaluation
% order can be made deterministic by disallowing operands with side effects,
% storing their values in temporary variables instead: e.g.
% x = f() + g();
% could be replaced by
% t1 = f(); t2 = g(); x = t1 + t2;
%
% Modulo Arithmetic
%-----
% C++ defines some types and their corresponding operations to follow
% arithmetic modulo  $2^n$  where  $n$  is the number of bits in the value
% representation. To be able detect integer overflows we do not model modulo
% arithmetic directly but instead we add the function mod which is defined to
% be the identity function for the value range of the respective datatype but
% which is otherwise undefined.
%
% Integer division "/" and remainder "%" (5.6.4) behave differently, see below.
%
% Bit-shift operators "<<" and ">>" (5.8.1) behave differently; see below.

%Open Issues:
%=====
% – Calling Convention (5.2 functions)
% I don't think the C++ standard specifies just how exactly function
% arguments are stored on the stack (or even if there is a stack, for that
% matter.) "call_f" obviously defines a certain (hopefully reasonable)
% behavior.
%
% This is defined in the ABI documents. (e.g., cdecl is defined in
% the calling convention is defined in 3.9 in the "System V ABI Intel 386
% Architecture Processor Supplement")
%
% – passing values to functions rather than addresses (5.2 functions)
% For specification/verification, it might be useful to have yet another PVS
% function for f which takes values (rather than expressions or addresses) as
% arguments. This hasn't been implemented yet.
%
% – entry points + address ranges for functions (5.2 functions)
% We should probably define for each function the range of addresses where
```

```

% the code of this function resides and the address of the entry point
%
% – should we implement exception values; some FPU ops and div 0 generate a
% well–defined hardware exception (e.g., Exception 0: divide error). Using
% Extended_Reals such an implementation could specify exception values (e.g.,
%  $x / 0 = \text{div}_0$ ), check for these values and instead of exiting with Fatal
% return with the appropriate exception.

% To do:
%=====
% – define expressions for bitfield types
% – with_new_refvar(ref, init)(Lambda (ref : (range(ref))) : xxx)
% – define << and >> operator
% – address of operator (&)
% – pointer assignment
% on assignment we have to adjust the pointer address to refer to the
% base class subobject of a derived object.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% 5.1: Primary expressions
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

PrimaryExpressions[State, Data : Type] : THEORY
BEGIN

  IMPORTING State_Transformer_Lift

  x : VAR Data

  % 5.1.2
  %>> MV: use Cpp type; define literals for int, char, float, bool // string
  literal(x) : [State -> ExprResult[State, Data]] = ok_result(x)

  % 5.1.7
  %>> MV: x is always an address (??)
  id(x) : [State -> ExprResult[State, Data]] = ok_result(x)

END PrimaryExpressions

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% 5.2: Postfix expressions
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% A C++ function "ret_data f(arg_data a)" currently corresponds to two PVS
% functions
%
% f(a : Address) : [State -> StmtResult[State, Semantics_<ret_data>]] =

```

```

% body
%
% (where each branch in the body must be terminated by a "return" statement)
% and
%
% call_f(a : [State -> ExprResult[State, Semantics_<arg_data>]]) :
% [State -> ExprResult[State, Semantics_<ret_data>]] =
% with_new_stackvar( lambda (a_addr : Address):
% assign(a_addr, a) ##
% f(a_addr) ##
% catch_return

```

PostfixExpressions[State : Type] : THEORY
BEGIN

IMPORTING Abstract_Read_Write_Plain, Cpp_Types

pm : **VAR** Plain_Memory[State]

a_ex : **VAR** ET[State, Address]

```

% 5.2.5: Class member access
member(a_ex, (offset : nat)) : [State -> ExprResult[State, Address]] =
(
  a_ex ## lambda (a : Address):
  ok_result(a + offset)
)

```

```

% 5.2.1: Subscripting
% "The expression E1[E2] is identical (by definition to *((E1) + (E2))
% see subscript in arithmetic expressions

```

```

% 5.2.5 pt 3: -> operator
% "the expression E1->E2 is converted to the equivalent from *((E1)).E2
% see arrow in unary expressions

```

```

% proof status :-)
% postinc_TCC1 : TCC Obligation

```

```

% 5.2.6 pt 1: C++'s .++ operator
postinc(pm, (typ : Cpp_Subtype(v(non_bool_integral?)))(a_ex) :
  ET[State, (range(typ))] =
(
  a_ex ## lambda (a : Address):
  read_data(pm, dt(typ))(a) ## lambda (n : (range(typ))):
  If range(typ)(mod(typ)(n + 1)) Then
    write_data(pm, dt(typ))(a, mod(typ)(n + 1)) ##
    ok_result(n)
  Else
    fatal_result
  Endif
)

```

```

)

% proof status :-)
% postinc_float_TCC1 : TCC Obligation

% proof status :-)
% postinc_float_TCC2 : TCC Obligation

% proof status :-)
% postinc_float_TCC3 : TCC Obligation

%>> example of floating point operation
postinc_float(pm, (typ : Cpp_Subtype(v(floating_point?)))(a_ex) :
  ET[State, (range_floating_point(cv_base(typ)))] =
(
  a_ex ## lambda (a : Address):
  read_data(pm, dt_cv_float(typ))(a) ##
  lambda (n : (range_floating_point(cv_base(typ)))):
  If range_floating_point(cv_base(typ))(mod_float(cv_base(typ))(er_plus(n, 1))) Then
    write_data(pm, dt_cv_float(typ))(a, mod_float(cv_base(typ))(er_plus(n, 1))) ##
    ok_result(n)
  Else
    fatal_result
  Endif
)

% 5.2.6 pt 1: C++'s ++ operator for pointers to complete object
% deferred to arithmetic expressions

% 5.2.6 pt 1
% .++ for argument type bool (deprecated)

% 5.2.6 pt 2: C++'s .-- operator
% .-- is not defined for argument type bool
postdec(pm, (typ : Cpp_Subtype(v(non_bool_integral?)))(a_ex) :
  ET[State, (range(typ))] =
(
  a_ex ## lambda (a : Address):
  read_data(pm, dt(typ))(a) ## lambda (n : (range(typ))):
  If range(typ)(mod(typ)(n - 1)) Then
    write_data(pm, dt(typ))(a, mod(typ)(n - 1)) ##
    ok_result(n)
  Else
    fatal_result
  Endif
)

% To do: postdec_float

% 5.2.6 pt 2: C++'s .-- operator for pointers to complete object
% deferred to arithmetic expressions

```


B PVS Theory Sources

```
% 5.2.7 dynamic_cast
% TODO: definition of dynamic_cast (5.2.7)

% 5.2.8 typeid
% TODO: definition of typeid (5.2.8)

% 5.2.9 static_cast
% TODO: definition of static_cast (5.2.9)
% - static_cast<reference_type>(v) = lvalue otherwise = rvalue 5.2.9.1
% - static_cast can only increase qualifiedness (+volatile / +const)

% 5.2.10 reinterpret_cast
% TODO: define reinterpret_cast (5.2.10)
% - reinterpret_cast<reference_type>(v) = lvalue otherwise = rvalue 5.2.10.1
% - reinterpret_cast<>(const v) = const 5.2.10.2
% - mapping is implementation defined

% MV: we cannot define reinterpret cast in a generic way using
% from_byte(T2)(to_byte(T1)); (5.2.10.3) the representation may or may
% not change !!!

% 5.2.11 const_cast
% TODO: define const_cast (5.2.11)
```

END PostfixExpressions

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% 5.3: Unary expressions
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

UnaryExpressions[State : **Type**] : **Theory**
BEGIN

IMPORTING PostfixExpressions

pm : **VAR** Plain_Memory[State]

a_ex : **Var** ET[State, Address]

% proof status :-)
% deref_TCC1 : TCC Obligation

```
% 5.3.1: C++'s * operator
deref(pm, (p_typ : Cpp_Subtype(cv(pointer?)))
  (p_ex : ET[State, (range_pointer(cv_base(p_typ))])) :
  ET[State, Memory_Address] =
(
  p_ex ## lambda (ptr_val : (range_pointer(cv_base(p_typ)))) :
  If not_null?(p_typ)(ptr_val) And
  check_bounds(p_typ)(ptr_val) And
```

```

    up?(address_of(p_typ)(ptr_val))
Then
    ok_result(down(address_of(p_typ)(ptr_val)))
Else
    fatal_result
Endif
)

% To do: * for pointer to function

% proof status :-)
% arrow_TCC1 : TCC Obligation

% proof status :-)
% arrow_TCC2 : TCC Obligation

% 5.2.5 pt 3: -> operator
% 'the expression E1->E2 is converted to the equivalent from (*(E1)).E2
arrow(pm, (p_typ : Cpp_Subtype(cv(Lambda (t : Cpp_Type) : pointer?(t) And
(class?(typ(t)) Or union?(typ(t))))))
  (p_ex : ET[State, (range_pointer(cv_base(p_typ)))]), offset : nat) :
  ET[State, Address] =
(
  deref(pm, p_typ)(p_ex) ## Lambda (a : Address) :
  member(ok_result(a), offset)
)

% 5.3.1 pt 2 C++'s & operator
% not needed for variables; = member() for class members

% 5.3.1 pt 6 unary + operator
% (not needed as it returns the value of the argument)

% proof status :-)
% unary_minus_TCC1 : TCC Obligation

% proof status :-)
% unary_minus_TCC2 : TCC Obligation

% 5.3.1 pt 7 unary - operator
unary_minus(typ : Cpp_Subtype(cv({t : Cpp_Type |
(char?(t) Or wchar_t?(t) Or signed_integer?(t)})))
  (n_ex : ET[State, (range(typ))]) : ET[State, (range(typ))] =
(
  n_ex ## lambda (n : (range(typ))) :
If range(typ)(mod(typ)(-n)) Then
  ok_result(mod(typ)(-n))
Else
  fatal_result
Endif
)

```

B PVS Theory Sources

```

% To do: unary_minus_float

% proof status :-)
% unary_minus_unsigned_TCC1 : TCC Obligation

% proof status :-)
% unary_minus_unsigned_TCC2 : TCC Obligation

% 5.3.1 pt 7 "the negative of an unsigned quantity is computed by
% subtracting its value from 2^n"
unary_minus_unsigned(typ : Cpp_Subtype(cv(unsigned_integer?)))
  (n_ex : ET[State, (range(typ))]) : ET[State, (range(typ))] =
(
  n_ex ## lambda (n : (range(typ))) :
  If range(typ)(mod(typ)(2^max_value_bits(typ) - n)) Then
    ok_result(mod(typ)(2^max_value_bits(typ) - n))
  Else
    fatal_result
  Endif
)

% 5.3.1 pt 8: C++'s ! operator
unary_not(typ : Cpp_Subtype(cv(bool?)))
  (b_ex : ET[State, bool]) : ET[State, bool] =
(
  b_ex ## lambda (b : bool) :
  ok_result(Not b)
)

% To do
% 5.3.1 pt 9 C++'s ~ operator
% "The result is the one's complement of its operand."
%
%>> MV and Tjark think (hope) that this means we apply one's complement
% to the binary representation and see whether a valid value
% comes out of it.
unary_bitnot(typ : Cpp_Subtype(cv(non_bool_integral_enum?)))
  (n_ex : ET[State, (range(typ))]) : ET[State, (range(typ))] =
(
  n_ex ## lambda (n : (range(typ))) :
  % Because dt(typ) is a pod for all of these types we can take any address:
  % Mem(0)
  Let a = Mem(0),
    bitnot_value =
      from_byte(dt(typ))(
        apply_bitop_valuefield(value_bitmask(typ), binary_not)
          (to_byte(dt(typ))(n, a)), a)
  In
  If up?(bitnot_value) And range(typ)(down(bitnot_value)) Then
    ok_result(down(bitnot_value))
  Else
    fatal_result
)

```

```

Endif
)

% proof status :-)
% preinc_TCC1 : TCC Obligation

% 5.3.2 pt 1: C++'s ++. operator
% Note that the prefix increment/decrement operators return an lvalue (i.e.
% an address).
preinc(pm, (typ : Cpp_Subtype(v(non_bool_integral?)))(a_ex) :
  ET[State, Address] =
(
  a_ex ## lambda (a : Address):
  read_data(pm, dt(typ))(a) ## lambda (n : (range(typ))):
  If range(typ)(mod(typ)(n + 1)) Then
    write_data(pm, dt(typ))(a, mod(typ)(n + 1)) ##
    ok_result(a)
  Else
    fatal_result
  Endif
)

% To do: preinc_float

% 5.3.2 pt 1: C++'s ++. operator for pointers
% deferred to arithmetic expressions

% 5.3.2 pt 2: C++'s --. operator
% --. is not defined for argument type bool
predec(pm, (typ : Cpp_Subtype(v(non_bool_integral?)))(a_ex) :
  ET[State, Address] =
(
  a_ex ## lambda (a : Address):
  read_data(pm, dt(typ))(a) ## lambda (n : (range(typ))):
  If range(typ)(mod(typ)(n - 1)) Then
    write_data(pm, dt(typ))(a, mod(typ)(n - 1)) ##
    ok_result(a)
  Else
    fatal_result
  Endif
)

% To do: predec_float

% 5.3.2 pt 1: C++'s --. operator for pointers
% deferred to arithmetic expressions

% proof status :-)
% sizeof_TCC1 : TCC Obligation

% 5.3.3 sizeof operator (only the type parameter form is implemented)

```

B PVS Theory Sources

```
% TODO: define the type std::size_t and return this type in sizeof
% this is elsa's size_of_type expression
%
% 7.17 pt 2 C – Standard: size_t is an unsigned integer type
sizeof(typ : Cpp_Type) : ET[State, (range(uint))] =
  If range(uint)(size_of(typ)) Then
    ok_result(size_of(typ))
  Else
    fatal_result
  Endif
```

```
% 5.3.4 C++ new operator
% TODO
```

```
% 5.3.5 C++ delete operator
% TODO
```

END UnaryExpressions

```
% 5.4 definition of cast (T)exp
% TODO
```

PointerToMemberExpressions[State : **Type**] : **THEORY**
BEGIN

Importing UnaryExpressions

pm : **VAR** Plain_Memory[State]

a_ex : **Var** ET[State, Address]

```
% 10 pt 5 "A base class subobject might have a layout different from the
% layout of the most derived object of the same type."
```

```
% 9.2 pt 17 "A pointer to a pod-struct object, suitably converted using a
% reinterpret_cast, points to its initial member [...] and vice
% versa."
```

```
% GCC seems to adjust the addresses of a pointer upon pointer assignment. For
% example, in the assignment B* bp = &a where class A : C, B {}; bp will be
% set to point to the B subobject in A. This goes as far as that a vtable is
% allocated for B. However, the vtable value is not identical to a
% freestanding B.
```

```
%>> Können wir zeigen, das ein anderes Vorgehen Typinformation in Pointern oder
% in Pointer auf Klassenmember braucht?
```

```
%>> Ich würde daher pointer to member expressions wie folgt formalisieren:
% We assume that members of a subobject are not reordered with members of
% other subobjects or with members of the derived object. We can therefore
% adjust the pointer to point to a base class. This base class may possibly
% have a layout different from a standalone object.
```

```

%
% However, to detect this, type information must be stored with the object.
% (??)
%
% We thus require the pointer to which pointer to member expressions are
% bound to have the same class as the pointer to member.

```

```

% proof status :-)
% ptm_member_TCC1 : TCC Obligation

```

```

% 5.5 pt 2 C++'s .* operator for pointer to member
% To do:
% 5.5 pt 4 "If the dynamic type of the object does not contain the member
% to which the pointer refers, the behaviour is undefined."
% range(typ) only contains offsets that are valid for the possibly derived
% class typ(typ).

```

```

ptm_member(typ : Cpp_Subtype(cv(pointer_to_member?)))
  (a_ex, (ptm_ex : ET[State, (range_ptm(cv_base(typ))])) :
    ET[State, Address] =
  (
    ptm_ex ## Lambda (ofs : (range_ptm(cv_base(typ)))) :
      % the offset of the member
      member(a_ex, ofs)
  )

```

```

% proof status :-)
% ptm_arrow_TCC1 : TCC Obligation

```

```

% 5.5 pt 3 C++'s ->* operator for pointer to member
% To do:
% check typ(ptr_typ) is a unambiguous accessible base class of typ(ptm_typ).

```

```

ptm_arrow(pm, (ptm_typ : Cpp_Subtype(cv(pointer_to_member?)))
  (p_ex : ET[State, (range_pointer(cv_base(
    pointer(typ(cv_base(ptm_typ)))]))],
    ptm_ex : ET[State, (range_ptm(cv_base(ptm_typ)))] :
    ET[State, Address] =
  (
    ptm_ex ## Lambda (ofs : (range_ptm(cv_base(ptm_typ)))) :
      arrow(pm, pointer(typ(cv_base(ptm_typ)))(p_ex, ofs)
  )

```

END PointerToMemberExpressions

```

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% 5.6: Multiplicative operators
% 5.7: Additive operators
% 5.8: Shift operators
% 5.9: Relational operators
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

```

ArithmeticExpressions[State : Type] : THEORY
BEGIN

IMPORTING UnaryExpressions

pm : **Var** Plain_Memory[State]

a_ex : **Var** ET[State, Address]

i_typ : **Var** Cpp_Subtype(cv(non_bool_integral_enum?))

*% The standard defines that bool values are subject to integral promotions:
 % 5 pt 9 "This pattern is called the 'usual arithmetic conversions' ...
 % Otherwise, the integral promotion (4.5) shall be performed."*

% Multiplicative Operators

%=====

*% 5.6: C++'s * operator (multiplication)*

times(i_typ)(n1_ex, n2_ex : ET[State, (range(i_typ))]) :

ET[State, (range(i_typ))] =
 (
 n1_ex ## **lambda** (n1 : (range(i_typ))) :
 n2_ex ## **lambda** (n2 : (range(i_typ))) :
If range(i_typ)(mod(i_typ)(n1 * n2)) **Then**
 ok_result(mod(i_typ)(n1 * n2))
Else
 fatal_result
Endif
)

% To do: times_float

*% 5.6.4: "If the second operand of / or % is zero the behavior is undefined;
 % otherwise (a/b)*b + a%b is equal to a. If both operands are nonnegative
 % then the remainder is nonnegative; if not, the sign of the remainder is
 % implementation-defined."*

% C99 requires truncation towards zero

% proof status :-)

% mod_impl_TCC1 : TCC Obligation

% 5.6: C++'s % operator (i % j resp. i / j)

mod_impl(i : int, j : {k : int | k /= 0}) : {r : int | abs(r) < abs(j)}

div_impl(i : int, j : {k : int | k /= 0}) : int

mod_impl_pos_range : **Axiom**

Forall (i, j : nat) : j /= 0 **Implies**

mod_impl(i, j) = mod(i, j)

% proof status :-)

```

% div_impl_pos_range_TCC1 : TCC Obligation

div_impl_pos_range : Axiom
  Forall (i, j : nat) : j /= 0 Implies
    div_impl(i, j) = ndiv(i, j)

mod_impl_sign : Axiom
  Forall (i : int, j : int) : j /= 0 And (i < 0 Or j < 0) Implies
    mod_impl(i, j) <= 0 Or mod_impl(i, j) >= 0

div_impl_mod_impl_result : Axiom
  Forall (i : int, j : int) : j /= 0 And (i < 0 Or j < 0) Implies
    div_impl(i, j) * j + mod_impl(i, j) = i

mod(i_typ)(n1_ex, n2_ex : ET[State, (range(i_typ))]) :
  ET[State, (range(i_typ))] =
  (
    n1_ex ## lambda (n1 : (range(i_typ))):
    n2_ex ## lambda (n2 : (range(i_typ))):
    if n2 /= 0 and range(i_typ)(mod(i_typ)(mod_impl(n1, n2))) then
      ok_result(mod(i_typ)(mod_impl(n1, n2)))
    else
      fatal_result
    endif
  )

% 5.6: C++'s / operator
div(i_typ)
  (n1_ex, n2_ex : ET[State, (range(i_typ))]) : ET[State, (range(i_typ))] =
  (
    n1_ex ## lambda (n1 : (range(i_typ))):
    n2_ex ## lambda (n2 : (range(i_typ))):
    if n2 /= 0 And range(i_typ)(mod(i_typ)(div_impl(n1, n2))) then
      ok_result(mod(i_typ)(div_impl(n1, n2)))
    else
      fatal_result
    endif
  )

% Floating point division must be handled separately.
% We use extended reals, after checking for x / 0

% To do:
% div_float(typ : Cpp_Subtype(cv(floating_point?)))
% (n1_ex, n2_ex : ET[State, (range(typ))]) : ET[State, (range(typ))] =
% (
%   n1_ex ## lambda (n1 : (range(typ))):
%   n2_ex ## lambda (n2 : (range(typ))):
%   If n2 /= 0 And range(typ)(mod(typ)(n1 / n2)) Then
%     ok_result(mod(typ)(n1 / n2))
%   Else
%     fatal_result

```


B PVS Theory Sources

```

% Endif
% )

% Additive operators
%=====

% 5.7
plus(i_typ)(n1_ex, n2_ex : ET[State, (range(i_typ))]) :
  ET[State, (range(i_typ))] =
  (
    n1_ex ## lambda (n1 : (range(i_typ))):
    n2_ex ## lambda (n2 : (range(i_typ))):
    If range(i_typ)(mod(i_typ)(n1 + n2)) Then
      ok_result(mod(i_typ)(n1 + n2))
    Else
      fatal_result
    Endif
  );

% To do: plus_float

% proof status :-)
% plus_ptr_TCC1 : TCC Obligation

% proof status :-)
% plus_ptr_TCC2 : TCC Obligation

% proof status :-)
% plus_ptr_TCC3 : TCC Obligation

% proof status :-)
% plus_ptr_TCC4 : TCC Obligation

% proof status :-)
% plus_ptr_TCC5 : TCC Obligation

% proof status :-)
% plus_ptr_TCC6 : TCC Obligation

% 5.7 pt 4 ff C++'s + operator for pointers
plus_ptr(typ : Cpp_Subtype(cv(pointer?)), i_typ)
  (p_ex : ET[State, (range_pointer(cv_base(typ)))]),
  n_ex : ET[State, (range(i_typ))]) :
  ET[State, (range_pointer(cv_base(typ)))] =
  (
    p_ex ## lambda (ptr_val : (range_pointer(cv_base(typ)))) :
    n_ex ## lambda (n : (range(i_typ))) :
    If not_null?(typ)(ptr_val) And
      range_pointer(cv_base(typ))
      (add(typ)(ptr_val, n * size_of(typ(cv_base(typ)))))) And
      check_bounds(typ)
      (add(typ)(ptr_val, n * size_of(typ(cv_base(typ))))))
  )

```

```

Then
  ok_result(add(typ)(ptr_val, n * sizeof(typ(cv_base(typ))))))
Else
  fatal_result
Endif
)

% proof status :-)
% postinc_ptr_TCC1 : TCC Obligation

% proof status :-)
% postinc_ptr_TCC2 : TCC Obligation

% proof status :-)
% postinc_ptr_TCC3 : TCC Obligation

% proof status :-)
% postinc_ptr_TCC4 : TCC Obligation

% 5.2.6 pt 1: C++'s ++ operator for pointers to complete object
% (equivalent to  $p = p + 1$ )
postinc_ptr(pm, (typ : Cpp_Subtype(v(pointer?))))(a_ex) :
  ET[State, (range_pointer(cv_base(typ)))] =
(
  a_ex ## lambda (a : Address):
  plus_ptr(typ, uchar)(read_data(pm, dt_cv_pointer(typ))(a), ok_result(1)) ##
  lambda (ptr_val : (range_pointer(cv_base(typ)))):
  write_data(pm, dt_cv_pointer(typ))(a, ptr_val) ##
  ok_result(ptr_val)
)

% 5.3.2 pt 1: C++'s ++. operator for pointers
% deferred to arithmetic expressions
preinc_ptr(pm, (typ : Cpp_Subtype(v(pointer?))))(a_ex) : ET[State, Address] =
(
  a_ex ## lambda (a : Address) :
  plus_ptr(typ, uchar)(
    read_data(pm, dt_cv_pointer(typ))(a),
    ok_result(1)) ##
  lambda (ptr_val : (range_pointer(cv_base(typ)))) :
  write_data(pm, dt_cv_pointer(typ))(a, ptr_val) ##
  ok_result(a)
)

% proof status :-)
% subscript_TCC1 : TCC Obligation

% proof status :-)
% subscript_TCC2 : TCC Obligation

% proof status :-)
% subscript_TCC3 : TCC Obligation

```

```

% proof status :-)
% subscript_TCC4 : TCC Obligation

% 5.2.1: Subscripting
% "The expression E1[E2] is identical (by definition to *((E1) + (E2))
% The check_bounds function in plus and minus enforces the conditions of
% clause 5.7 pt 5
% provided the bounds-checking pointer model is implemented
subscript(pm, (typ : Cpp_Subtype(array?)), i_typ)
  (a_ex, (n_ex : ET[State, (range(i_typ))])) : ET[State, Memory_Address] =
  (
    a_ex ## Lambda (a : Address) :
    deref(pm, pointer(typ(typ)))(
      plus_ptr(pointer(typ(typ)), i_typ)(
        read_data(pm, dt_cv_pointer(pointer(typ(typ))))(a),
        n_ex)
      )
    )
  )

% 5.7 C++'s - operator
minus(i_typ)(n1_ex, n2_ex : ET[State, (range(i_typ))]) :
  ET[State, (range(i_typ))] =
  (
    n1_ex ## lambda (n1 : (range(i_typ))):
    n2_ex ## lambda (n2 : (range(i_typ))):
    If range(i_typ)(mod(i_typ)(n1 - n2)) Then
      ok_result(mod(i_typ)(n1 - n2))
    Else
      fatal_result
    Endif
  )

% To do: minus_float

% 5.7 pt 4 ff C++'s - operator for pointer - integral
minus_ptr(typ : Cpp_Subtype(cv(pointer?)), i_typ)
  (p_ex : ET[State, (range_pointer(cv_base(typ)))]),
  n_ex : ET[State, (range(i_typ))] :
  ET[State, (range_pointer(cv_base(typ)))] =
  (
    p_ex ## lambda (ptr_val : (range_pointer(cv_base(typ)))) :
    n_ex ## lambda (n : (range(i_typ))) :
    If not_null?(typ)(ptr_val) And
      range_pointer(cv_base(typ))
      (add(typ)(ptr_val, -n * size_of(typ(cv_base(typ))))) And
      check_bounds(typ)
      (add(typ)(ptr_val, -n * size_of(typ(cv_base(typ)))))
    Then
      ok_result(add(typ)(ptr_val, -n * size_of(typ(cv_base(typ)))))
    Else
      fatal_result
  )

```

```

Endif
)

% 5.2.6 pt 2: C++'s .-- operator for pointers to complete object
postdec_ptr(pm, (typ : Cpp_Subtype(v(pointer?)))(a_ex) :
  ET[State, (range_pointer(cv_base(typ)))] =
(
  a_ex ## lambda (a : Address):
  minus_ptr(typ, uchar)(read_data(pm, dt_cv_pointer(typ))(a),
    ok_result(1)) ##
    lambda (ptr_val : (range_pointer(cv_base(typ)))):
  write_data(pm, dt_cv_pointer(typ))(a, ptr_val) ##
  ok_result(ptr_val)
)

% 5.3.2 pt 1: C++'s --. operator for pointers
predec_ptr(pm, (typ : Cpp_Subtype(v(pointer?)))(a_ex) :
  ET[State, Address] =
(
  a_ex ## lambda (a : Address) :
  minus_ptr(typ, uchar)(read_data(pm, dt_cv_pointer(typ))(a),
    ok_result(1)) ##
    lambda (ptr_val : (range_pointer(cv_base(typ)))):
  write_data(pm, dt_cv_pointer(typ))(a, ptr_val) ##
  ok_result(a)
)

% proof status :-)
% ptr_to_nonzero_object_TCC1 : TCC Obligation

% proof status :-)
% ptr_to_nonzero_object_TCC2 : TCC Obligation

% proof status :-)
% ptr_to_nonzero_object_TCC3 : TCC Obligation

% proof status :-)
ptr_to_nonzero_object : Lemma
Forall (typ : Cpp_Subtype(Lambda (p : (pointer?)) : Not void?(typ(p)) And
Not function?(typ(p)))) :
  (array?(typ(cv_base(typ))) Implies size_of(typ(cv_base(typ))) > 0)
Implies
  size_of(typ(cv_base(typ))) > 0

% 5.7 pt 6 ff C++'s - operator for pointer - pointer
% "When two pointers to elements of the same array object are subtracted,
% ..."
% Thus, the pointers have to have the same type.

% proof status :-)
% minus_ptr_typ_is_Cpp_Type_TCC1 : TCC Obligation

```

```

% proof status :-)
minus_ptr_typ_is_Cpp_Type : Lemma
  Forall (typ : Cpp_Subtype(cv(
    Lambda (p : (pointer?)) : Not void?(typ(p)) And
    Not function?(typ(p)))) :
    Cpp_Type?(typ(cv_base(typ)))

% proof status :-)
% minus_ptr_ptr_TCC1 : TCC Obligation

% proof status :-)
% minus_ptr_ptr_TCC2 : TCC Obligation

% proof status :-)
% minus_ptr_ptr_TCC3 : TCC Obligation

% proof status :-)
% minus_ptr_ptr_TCC4 : TCC Obligation

% proof status :-)
% minus_ptr_ptr_TCC5 : TCC Obligation

% proof status :-)
% minus_ptr_ptr_TCC6 : TCC Obligation

% proof status :-)
% minus_ptr_ptr_TCC7 : TCC Obligation

% proof status :-)
% minus_ptr_ptr_TCC8 : TCC Obligation

% proof status :-)
% minus_ptr_ptr_TCC9 : TCC Obligation

% proof status :-)
% minus_ptr_ptr_TCC10 : TCC Obligation

minus_ptr_ptr(typ : Cpp_Subtype(cv(
  Lambda (p : (pointer?)) : Not void?(typ(p)) And
  Not function?(typ(p))))
  (p_ex1, p_ex2 : ET[State, (range_pointer(cv_base(typ)))] :
    ET[State, (range(int))] =
  (
    p_ex1 ## lambda (ptr_val1 : (range_pointer(cv_base(typ)))) :
    p_ex2 ## lambda (ptr_val2 : (range_pointer(cv_base(typ)))) :
    If (array?(typ(cv_base(typ))) Implies size_of(typ(cv_base(typ))) > 0) And
      % otherwise we subtract pointer to incomplete arrays
      not_null?(typ)(ptr_val1) And
      not_null?(typ)(ptr_val2) And
      same_array(typ)(ptr_val1, ptr_val2) And
      up?(address_of(typ)(ptr_val1)) And
      up?(address_of(typ)(ptr_val2))
  )

```

```

Then
  Let difference = ndiv(offset(down(address_of(typ)(ptr_val1))) -
                        offset(down(address_of(typ)(ptr_val2))),
                        size_of(typ(cv_base(typ)))) In
    If range(int)(difference) Then
      ok_result(difference)
    Else
      fatal_result
    Endif
  Else
    fatal_result
  Endif
)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Shift Operators
%=====

% 5.8 pt 1: "The behavior is undefined if the right operand is negative, or
% greater than or equal to the length in bits of the promoted left operand."
% We do not (yet) model the latter.
%
% We do not model that the result is implementation-defined if the first
% operand has a signed type (for <<). The result of >> is
% implementation-defined if the first operand has a signed type and a
% negative value; below we require the first operand to be nonnegative.

% 5.8 pt 1
% type of result = type of promoted left operand (i.e., int, uint, long,
% ulong, longlong, ulonglong)
% undefined if E2 < 0 or E2 > length in bits (max_value_bits)

% 5.8 pt 2
% E1 unsigned : mod(typ(E1))(E1 << n) otherwise from_byte(to_byte(E1) << n)

% 5.8 pt 3
% E1 unsigned : mod(typ(E1))(E1 >> n) if E1 >= 0 ; E1 < 0 => implementation
% defined value

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Rational operators
%=====

cmp(i_typ, (R : PRED[[real, real]])
  (n1_ex, n2_ex : ET[State, (range(i_typ))]) : ET[State, bool] =
(
  n1_ex ## lambda (n1 : (range(i_typ))):
  n2_ex ## lambda (n2 : (range(i_typ))):

```

```

    ok_result(R(n1, n2))
  )

% To do: cmp ops for float

% 5.9 <
lt(i_typ)(n1_ex, n2_ex : ET[State, (range(i_typ))]) :
  ET[State, bool] =
    cmp(i_typ, <)(n1_ex, n2_ex)

% 5.9 >
gt(i_typ)(n1_ex, n2_ex : ET[State, (range(i_typ))]) :
  ET[State, bool] =
    cmp(i_typ, >)(n1_ex, n2_ex)

% 5.9 <=
le(i_typ)(n1_ex, n2_ex : ET[State, (range(i_typ))]) :
  ET[State, bool] =
    cmp(i_typ, <=)(n1_ex, n2_ex)

% 5.9 >=
ge(i_typ)(n1_ex, n2_ex : ET[State, (range(i_typ))]) :
  ET[State, bool] =
    cmp(i_typ, >=)(n1_ex, n2_ex)

% proof status :-)
% cmp_pointer_TCC1 : TCC Obligation

% proof status :-)
% cmp_pointer_TCC2 : TCC Obligation

% Rational operators for pointers
%-----
% 5.9
cmp_pointer(p_typ : Cpp_Subtype(cv(pointer?)), R : PRED[[real, real]],
  cmp_null : bool)
  (p1_ex, p2_ex : ET[State, (range_pointer(cv_base(p_typ)))] :
    ET[State, bool] =
  (
    p1_ex ## lambda (ptr_val1 : (range_pointer(cv_base(p_typ)))) :
    p2_ex ## lambda (ptr_val2 : (range_pointer(cv_base(p_typ)))) :
      If not_null?(p_typ)(ptr_val1) And
        not_null?(p_typ)(ptr_val2) Then
          If same_array(p_typ)(ptr_val1, ptr_val2) And
            up?(address_of(p_typ)(ptr_val1)) And
              up?(address_of(p_typ)(ptr_val2))
            Then
              ok_result(R(offset(down(address_of(p_typ)(ptr_val1))),
                offset(down(address_of(p_typ)(ptr_val2)))))
            Else
              fatal_result
          Endif
  )

```

```

    Else
      ok_result(cmp_null)
    Endif
  )

lt_ptr(p_typ : Cpp_Subtype(cv(pointer?)))
  (p1_ex, p2_ex : ET[State, (range_pointer(cv_base(p_typ)))] :
    ET[State, bool] =
    cmp_pointer(p_typ, <, false)(p1_ex, p2_ex)

gt_ptr(p_typ : Cpp_Subtype(cv(pointer?)))
  (p1_ex, p2_ex : ET[State, (range_pointer(cv_base(p_typ)))] :
    ET[State, bool] =
    cmp_pointer(p_typ, >, false)(p1_ex, p2_ex)

le_ptr(p_typ : Cpp_Subtype(cv(pointer?)))
  (p1_ex, p2_ex : ET[State, (range_pointer(cv_base(p_typ)))] :
    ET[State, bool] =
    cmp_pointer(p_typ, <=, true)(p1_ex, p2_ex)

ge_ptr(p_typ : Cpp_Subtype(cv(pointer?)))
  (p1_ex, p2_ex : ET[State, (range_pointer(cv_base(p_typ)))] :
    ET[State, bool] =
    cmp_pointer(p_typ, >=, true)(p1_ex, p2_ex)

% Rational operators for pointers to members
%-----
% 5.9 pt 2
% "If two pointers point to nonstatic data members of the same
% object, or to subobjects or array elements of such members,
% recursively, the pointer to the later declared member compares
% greater provided the two members are not separated by an
% access-specifier label and provided their class is not a union."

END ArithmeticExpressions

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% 5.10: Equality operators
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

EqualityExpressions[State, Data : Type] : THEORY
BEGIN

  IMPORTING ArithmeticExpressions

  % C++'s == operator
  eq(Lex, r_ex : ET[State, Data]) :
    ET[State, bool] =
  (
    Lex ## lambda (l : Data) :

```



```

    r_ex ## lambda (r : Data) :
    ok_result((l = r))
);

% C++'s != operator
not_equal(l_ex, r_ex : ET[State, Data]) :
  ET[State, bool] =
  (
    l_ex ## lambda (l : Data) :
    r_ex ## lambda (r : Data) :
    ok_result((l /= r))
  );

% to do: pointers to the same function
% to do: 5.10 pt 2 pointers to members

```

END EqualityExpressions

```

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% 5.11: Bitwise AND operator
% 5.12: Bitwise exclusive OR operator
% 5.13: Bitwise inclusive OR operator
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

```

BitwiseExpressions[State : Type] : **THEORY**

BEGIN

Importing Ok_Result_Rewrite, Cpp_Types

typ : **Var** Cpp_Subtype(cv(non_bool_integral_enum?))

% 5.11. bit and &

bitand(typ)(n1_ex, n2_ex : ET[State, (range(typ))]) :
 ET[State, (range(typ))] =

```

  (
    n1_ex ## lambda (n1 : (range(typ))) :
    n2_ex ## lambda (n2 : (range(typ))) :
    % Because dt(typ) is a pod for all of these types we can take any address:
    % Mem(0)
    Let a = Mem(0),
        bitand_value =
          from_byte(dt(typ))(
            apply_bitop_valuefield(value_bitmask(typ), Lambda (l : list[Byte]):
              binary_and(l, to_byte(dt(typ))(n2, a)))
              (to_byte(dt(typ))(n1, a)), a)

```

In

```

  If up?(bitand_value) And range(typ)(down(bitand_value)) Then
    ok_result(down(bitand_value))
  Else

```

```

        fatal_result
    Endif
)

% 5.12. bit xor ^
bitxor(typ)(n1_ex, n2_ex : ET[State, (range(typ))]) :
    ET[State, (range(typ))] =
(
    n1_ex ## lambda (n1 : (range(typ))) :
    n2_ex ## lambda (n2 : (range(typ))) :
    % Because dt(typ) is a pod for all of these types we can take any address:
    % Mem(0)
    Let a = Mem(0),
        bitxor_value =
            from_byte(dt(typ))(
                apply_bitop_valuefield(value_bitmask(typ), Lambda (l : list[Byte]):
                    binary_xor(l, to_byte(dt(typ))(n2, a)))
                    (to_byte(dt(typ))(n1, a)), a)
            )
    In
        If up?(bitxor_value) And range(typ)(down(bitxor_value)) Then
            ok_result(down(bitxor_value))
        Else
            fatal_result
        Endif
)

% 5.13. bit or |
bitor(typ)(n1_ex, n2_ex : ET[State, (range(typ))]) :
    ET[State, (range(typ))] =
(
    n1_ex ## lambda (n1 : (range(typ))) :
    n2_ex ## lambda (n2 : (range(typ))) :
    % Because dt(typ) is a pod for all of these types we can take any address:
    % Mem(0)
    Let a = Mem(0),
        bitor_value =
            from_byte(dt(typ))(
                apply_bitop_valuefield(value_bitmask(typ), Lambda (l : list[Byte]):
                    binary_or(l, to_byte(dt(typ))(n2, a)))
                    (to_byte(dt(typ))(n1, a)), a)
            )
    In
        If up?(bitor_value) And range(typ)(down(bitor_value)) Then
            ok_result(down(bitor_value))
        Else
            fatal_result
        Endif
)

END BitwiseExpressions

```

B PVS Theory Sources

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% 5.14: Logical AND operator
% 5.15: Logical OR operator
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% !! left-to-right evaluation is guaranteed only for base types !!

LogicalExpressions[State : Type] : THEORY
BEGIN

  IMPORTING Ok_Result_Rewrite, Cpp_Types

  Lex, r_ex, ex : VAR [State -> ExprResult[State, bool]]

  % 5.14
  % C++'s && operator
  and_exp(l_ex, r_ex) : [State -> ExprResult[State, bool]] =
  (
    Lex ## lambda (l : bool):
    if l then
      r_ex ## lambda (r : bool):
      ok_result(l and r)
    else
      % 5.14.1: "Unlike &, && guarantees left-to-right evaluation: the second
      % operand is not evaluated if the first operand is false."
      ok_result(false)
    endif
  )

  % 5.15
  % C++'s || operator
  or_exp(l_ex, r_ex) : [State -> ExprResult[State, bool]] =
  (
    Lex ## lambda (l : bool):
    if l then
      % 5.15.1: "Unlike |, || guarantees left-to-right evaluation; moreover,
      % the second operand is not evaluated if the first operand evaluates to
      % true."
      ok_result(true)
    else
      r_ex ## lambda (r : bool):
      ok_result(l or r)
    endif
  )

END LogicalExpressions

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
```

% 5.16: Conditional operator

*%
 %%*

ConditionalExpression[State, Data : **Type**] : **THEORY**
BEGIN

IMPORTING Abstract_Read_Write_Plain, Cpp_Types

pm : **VAR** Plain_Memory[State]

d_ex1 : **VAR** [State → ExprResult[State, Data]]

d_ex2 : **VAR** [State → ExprResult[State, Data]]

b_ex : **VAR** [State → ExprResult[State, bool]]

% C++'s b_ex ? d_ex1 : d_ex2

conditional_expr(b_ex, d_ex1, d_ex2) : [State → ExprResult[State, Data]] =

(
 b_ex ## **lambda** (b : bool) :
If b **Then**
 d_ex1
Else
 d_ex2
Endif
);

% Not modelled:

% 5.16 pt 2: one but not both expressions is void and is a throw expression

% 5.16 pt 2 – pt 6: restrictions on conversions and restrictions on these.

End ConditionalExpression

%%

% 5.17: Assignment operators

*%
 %%*

AssignmentExpressions[State : **Type**] : **THEORY**
BEGIN

IMPORTING ArithmeticExpressions, BitwiseExpressions

pm : **VAR** Plain_Memory[State]

AET : **Type** = [State → ExprResult[State, Address]]

a_ex : **Var** AET

p_typ : **Var** Cpp_Subtype(v(pointer?))

i_typ : **Var** Cpp_Subtype(v(non_bool_integral_enum?))

ci_typ : **Var** Cpp_Subtype(cv(non_bool_integral_enum?))

B PVS Theory Sources

```

% 5.17.7: "The behavior of an expression of the form E1 op= E2 is equivalent
% to E1 = E1 op E2 except that E1 is evaluated only once."

% the result of the second expression can be of const(a_typ)

% proof status :-)
% assign_times_TCC1 : TCC Obligation

% C++'s *= operator
assign_times(pm, i_typ)(a_ex : AET, n_ex : ET[State, (range(i_typ))]) :
  ET[State, Address] =
  (
    a_ex ## lambda (a : Address):
      times(i_typ)(read_data(pm, dt(i_typ))(a), n_ex) ##
        lambda (n : (range(i_typ))):
          write_data(pm, dt(i_typ))(a, n) ##
            ok_result(a)
  )

% To do: assign_times_float

% C++'s /= operator
assign_div(pm, i_typ)
  (a_ex : AET, n_ex : ET[State, (range(i_typ))]) : ET[State, Address] =
  (
    a_ex ## lambda (a : Address):
      div(i_typ)
        (read_data(pm, dt(i_typ))(a), n_ex) ## lambda (n : (range(i_typ))) :
          write_data(pm, dt(i_typ))(a, n) ##
            ok_result(a)
  )

% To do:
% assign_div_float(pm, f_typ)
% (a_ex : AET, n_ex : ET[State, (range(f_typ))]) : ET[State, Address] =
% (
% a_ex ## lambda (a : Address):
% div_float(f_typ)
% (read_data(pm, dt(f_typ))(a), n_ex) ## lambda (n : (range(f_typ))) :
% write_data(pm, dt(f_typ))(a, n) ##
% ok_result(a)
% )

% C++'s %= operator
assign_mod(pm, i_typ)(a_ex : AET, n_ex : ET[State, (range(i_typ))]) :
  ET[State, Address] =
  (
    a_ex ## lambda (a : Address):
      mod(i_typ)(read_data(pm, dt(i_typ))(a), n_ex) ##
        lambda (n : (range(i_typ))):
          write_data(pm, dt(i_typ))(a, n) ##
  )

```

```

    ok_result(a)
  )

% C++'s += operator
assign_plus(pm, i_typ)(a_ex : AET, n_ex : ET[State, (range(i_typ))]) :
  ET[State, Address] =
  (
    a_ex ## lambda (a : Address):
    plus(i_typ)(read_data(pm, dt(i_typ))(a), n_ex) ##
      lambda (n : (range(i_typ))):
    write_data(pm, dt(i_typ))(a, n) ##
    ok_result(a)
  )

% To do: assign_plus_float

% proof status :-)
% assign_plus_ptr_TCC1 : TCC Obligation

% proof status :-)
% assign_plus_ptr_TCC2 : TCC Obligation

assign_plus_ptr(pm, p_typ, ci_typ)(a_ex : AET,
  n_ex : ET[State, (range(ci_typ))]) : ET[State, Address] =
  (
    a_ex ##
      lambda (a : Address):
    plus_ptr(p_typ, ci_typ)(read_data(pm, dt_cv_pointer(p_typ))(a), n_ex) ##
      lambda (n : (range_pointer(cv_base(p_typ)))):
    write_data(pm, dt_cv_pointer(p_typ))(a, n) ##
    ok_result(a)
  )

% C++'s -= operator
assign_minus(pm, i_typ)(a_ex : AET, n_ex : ET[State, (range(i_typ))]) :
  ET[State, Address] =
  (
    a_ex ## lambda (a : Address):
    minus(i_typ)(read_data(pm, dt(i_typ))(a), n_ex) ##
      lambda (n : (range(i_typ))):
    write_data(pm, dt(i_typ))(a, n) ##
    ok_result(a)
  )

% To do: assign_minus_float

assign_minus_ptr(pm, p_typ, ci_typ)(a_ex : AET,
  n_ex : ET[State, (range(ci_typ))]) : ET[State, Address] =
  (
    a_ex ## lambda (a : Address):
    minus_ptr(p_typ, ci_typ)(read_data(pm, dt_cv_pointer(p_typ))(a), n_ex) ##
      lambda (n : (range_pointer(cv_base(p_typ)))) :

```

B PVS Theory Sources

```
    write_data(pm, dt_cv_pointer(p_ttyp))(a, n) ##
    ok_result(a)
)

% C++'s &= operator
assign_bitand(pm, i_ttyp)(a_ex : AET, n_ex : ET[State, (range(i_ttyp))]) :
  ET[State, Address] =
(
  a_ex ## lambda (a : Address):
  bitand(i_ttyp)(read_data(pm, dt(i_ttyp))(a), n_ex) ##
  lambda (n : (range(i_ttyp))) :
  write_data(pm, dt(i_ttyp))(a, n) ##
  ok_result(a)
)

% C++'s ^= operator
assign_bitxor(pm, i_ttyp)(a_ex : AET, n_ex : ET[State, (range(i_ttyp))]) :
  ET[State, Address] =
(
  a_ex ## lambda (a : Address):
  bitxor(i_ttyp)(read_data(pm, dt(i_ttyp))(a), n_ex) ##
  lambda (n : (range(i_ttyp))) :
  write_data(pm, dt(i_ttyp))(a, n) ##
  ok_result(a)
)

% C++'s |= operator
assign_bitor(pm, i_ttyp)(a_ex : AET, n_ex : ET[State, (range(i_ttyp))]) :
  ET[State, Address] =
(
  a_ex ## lambda (a : Address):
  bitor(i_ttyp)(read_data(pm, dt(i_ttyp))(a), n_ex) ##
  lambda (n : (range(i_ttyp))) :
  write_data(pm, dt(i_ttyp))(a, n) ##
  ok_result(a)
)
```

END AssignmentExpressions

AssignmentExpressions2[State, Data : **Type**] : **Theory**

Begin

Importing AssignmentExpressions

% Note that the assignment operators return an lvalue (i.e. an address).

*% We define assignment operators only for non class types.
% 5.17 pt 4 "Assignment to objects of a class is defined by the copy
% assignment operator".*

% Not modelled: 5.17 pt 8 restrictions on overlapping objects

```

pm : VAR Plain_Memory[State]

AET : Type = [State -> ExprResult[State, Address]]
a_ex : Var AET

% C++'s = operator
assign(pm, (idt : (interpreted_data_type?[Data])))
  (a_ex : AET, d_ex : ET[State, Data]) : ET[State, Address] =
(
  a_ex ## lambda (a : Address):
  d_ex ## lambda (d : Data) :
  write_data(pm, idt)(a, d) ##
  ok_result(a)
)

```

END AssignmentExpressions2

```

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% 5.18: Comma operator
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

```

CommaExpressions[State, Data1, Data2 : Type] : THEORY
BEGIN

IMPORTING Ok_Result_Rewrite

```

L_ex : VAR [State -> ExprResult[State, Data1]]
r_ex : VAR [State -> ExprResult[State, Data2]]

% C++'s , operator
comma(L_ex, r_ex) : [State -> ExprResult[State, Data2]] =
  (L_ex ## r_ex)

```

END CommaExpressions

```

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% Expressions: imports everything
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

```

Expressions[State : Type] : THEORY
BEGIN

```

IMPORTING PrimaryExpressions,
  % PostfixExpressions,
  % UnaryExpressions,
  PointerToMemberExpressions,

```



```

% ArithmeticExpressions,
EqualityExpressions,
% BitwiseExpressions,
LogicalExpressions,
ConditionalExpression,
%AssignmentExpressions,
AssignmentExpressions2,
CommaExpressions

```

END Expressions

B.15 graph.pvs

Graph_Util[T : Type] : Theory

% various general stuff needed for graphs

Begin

Importing More_List_Props, More_Function

% proof status :-)

every_nth : **Lemma Forall**(P : PRED[T], l : list[T]) :
 every(P)(l) **IFF** (**Forall**(i : below(length(l))) : P(nth(l, i)))

% proof status :-)

% nth_append_left_TCC1 : TCC Obligation

% proof status :-)

nth_append_left : **Lemma**
Forall(l1, l2 : list[T], n : nat) :
 n < length(l1) **Implies**
 nth(append(l1, l2), n) = nth(l1, n)

% proof status :-)

% nth_append_right_TCC1 : TCC Obligation

% proof status :-)

nth_append_right : **Lemma**
Forall(l1, l2 : list[T], n : nat) :
 n >= length(l1) **And** n < length(append(l1, l2)) **Implies**
 nth(append(l1, l2), n) = nth(l2, n - length(l1))

% proof status :-)

list_pred_card : **Lemma**
Forall(l : list[T], P : PRED[T]) :
 is_finite(P) **And** every(P)(l) **And** length(l) > card(P) **Implies**
Exists(n1, n2 : below(length(l))) :
Not n1 = n2 **And**
 nth(l, n1) = nth(l, n2)

```

% A path_list is a list with at least one element. A one
% element path is an (pointed) empty path.
% For the transitive closure we have of course to take path
% with at least two elements.
path_list?(l : list[T]) : bool = length(l) >= 1

% proof status :-)
% path_start_TCC1 : TCC Obligation

path_start(l : (path_list?)) : T = car(l)

% proof status :-)
% path_end_TCC1 : TCC Obligation

path_end(l : (path_list?)) : T = nth(l, length(l) - 1)

% proof status :-)
% path_list_induction_TCC1 : TCC Obligation

% proof status :-)
% path_list_induction_TCC2 : TCC Obligation

% proof status :-)
path_list_induction: Lemma
  Forall(p: [(path_list?) -> boolean]):
    (Forall(t : T) : p(cons(t, null))) And
    (Forall(t : T, tail : (path_list?): p(tail) Implies p(cons(t, tail)))
    Implies
    Forall (l : (path_list?): p(l))

concatable?(left, right : list[T]) : bool =
  path_list?(left) And path_list?(right) And
  path_end(left) = path_start(right)

% proof status :-)
% concat_path_TCC1 : TCC Obligation

% proof status :-)
% concat_path_TCC2 : TCC Obligation

concat_path(lr : (concatable?)) : (path_list?) =
  append(Proj_1(lr), cdr(Proj_2(lr)))

% proof status :-)
length_concat_path : Lemma
  Forall(l, r : list[T]) : concatable?(l, r) Implies
  length(concat_path(l, r)) = length(l) + length(r) - 1

End Graph_Util

```

```

Subtype_stable[T : Type] : Theory
Begin
  l : Var list[T]
  P : Var PRED[T]

  % proof status :-)
  length_stable : Lemma
    every(P)(l) Implies length[(P)](l) = length[T](l)

  % proof status :-)
  % nth_stable_TCC1 : TCC Obligation

  % proof status :-)
  nth_stable : Lemma Forall(n : below(length(l))) :
    every(P)(l) Implies
      nth[(P)](l, n) = nth[T](l, n)

End Subtype_stable

Graph_Relations[T : Type] : Theory
Begin
  Importing Graph_Util

  R : Var PRED[[T, T]]

  transitive_closure(R) : PRED[[T, T]] = Lambda(x, y : T) :
    Forall(Q : PRED[[T, T]]) : subset?(R, Q) And transitive?(Q) Implies
      Q(x, y)

  % proof status :-)
  % trans_closure_char_TCC1 : TCC Obligation

  % proof status :-)
  % trans_closure_char_TCC2 : TCC Obligation

  % proof status :-)
  % trans_closure_char_TCC3 : TCC Obligation

  % proof status :-)
  trans_closure_char : Lemma
    Forall(x, y : T) :
      transitive_closure(R)(x, y) IFF
      Exists(l : list[T]) :
        length(l) >= 2 And
        path_start(l) = x And path_end(l) = y and
        Forall(i : below(length(l) - 1)) : R(nth(l, i), nth(l, i+1))

End Graph_Relations

Graph[T : Type] : Theory

```

Begin**Importing** Graph_Relations, Subtype_stable

```

graph_type : Type = [#
  nodes : PRED[T],
  edges : PRED[[nodes], (nodes)]
#]

g : Var graph_type

% proof status :-)
% path?_TCC1 : TCC Obligation

% proof status :-)
% path?_TCC2 : TCC Obligation

% proof status :-)
% path?_TCC3 : TCC Obligation

% proof status :-)
% path?_TCC4 : TCC Obligation

path?(g)(l : (path_list?[T])) : bool =
  (Forall(i : below(length(l))) : g'nodes(nth(l, i))) And
  (Forall(i : below(length(l) - 1)) : g'edges(nth(l, i), nth(l, i+1))))

% proof status :-)
nodes_path_start : Lemma
  Forall(p : (path?(g))) : g'nodes(path_start(p))

% proof status :-)
nodes_path_end : Lemma
  Forall(p : (path?(g))) : g'nodes(path_end(p))

% proof status :-)
% path_head_TCC1 : TCC Obligation

% proof status :-)
path_head : Lemma
  Forall(p : (path?(g)), n : upto(length(p))) :
    n >= 1 Implies
    path?(g)(head(p, n))

% proof status :-)
% path_tail_TCC1 : TCC Obligation

% proof status :-)
path_tail : Lemma
  Forall(p : (path?(g)), n : upto(length(p))) :
    length(p) - n >= 1 Implies
    path?(g)(tail(p, n))

```

```

% proof status :-)
% path_cdr_TCC1 : TCC Obligation

% proof status :-)
path_cdr : Lemma
  Forall(p : (path?(g))) :
    path_list?[T](cdr(p)) Implies path?(g)(cdr(p))

% proof status :-)
path_concat_path : Lemma
  Forall(p1, p2 : (path?(g))) : concatable?(p1, p2) Implies
    path?(g)(concat_path(p1, p2))

cycle_free?(g) : bool =
  Forall(x : T) : g'nodes(x) Implies
    Not transitive_closure(g'edges)(x, x)

% proof status :-)
trans_closure_char_graph : Lemma
  Forall(x, y : (g'nodes)) :
    transitive_closure(g'edges)(x,y) IFF
  Exists(l : (path?(g))) :
    length(l) >= 2 And
    path_start(l) = x And path_end(l) = y

% proof status :-)
path_cycle_free : Lemma
  cycle_free?(g) IFF
  Forall(l : (path?(g)), i1, i2 : below(length(l))) :
    nth(l, i1) = nth(l, i2) Implies i1 = i2

% proof status :-)
different_nodes_on_edge : Lemma
  Forall(x, y : (g'nodes)) :
    cycle_free?(g) And
    g'edges(x, y) Implies
  Not x = y

% set of roots of a graph
roots(g) : PRED[(g'nodes)] =
  {y : (g'nodes) | Forall(x : (g'nodes)) : Not g'edges(x,y) }

% A tree is cycle free and each node has at most one predecessor.
tree?(g) : bool =
  cycle_free?(g) And
  Forall(x, y, z : (g'nodes)) :
    g'edges(x,z) And g'edges(y,z) Implies x = y

```

```

% In a tree there is only one path going up from a certain node.
% So any two path ending in the same node only differ in their length.
% proof status :-)
tree_path : Lemma
  Forall(p1, p2 : (path?(g))) :
    tree?(g) And
    path_end(p1) = path_end(p2) And
    length(p1) = length(p2) Implies
      p1 = p2

% a graph is finite if its set of nodes is
finite?(g) : bool = is_finite(g'nodes)

finite_tree?(g) : bool = finite?(g) and tree?(g)

% proof status :-)
% root_path_TCC1 : TCC Obligation

root_path(g : graph_type, n : (g'nodes)) : PRED[(path?(g))] =
  Lambda(p : (path?(g))) :
    roots(g)(path_start(p)) And path_end(p) = n

%%%%%%%%%%%%%%
%
% singleton root_path
%
% In finite trees there is for every node precisely one
% path upwards to a root (root_path). This is proved in the following
% and then the function path_to_root is defined (via selection out of
% a singleton).
%
%%%%%%%%%%%%%%

% proof status :-)
% path_length_bound_TCC1 : TCC Obligation

% proof status :-)
path_length_bound : Lemma
  finite?(g) and cycle_free?(g) Implies
    Forall(p : (path?(g))) : length(p) <= card(g'nodes)

% proof status :-)
non_root_path : Lemma
  Forall(n : (g'nodes)) : roots(g)(n) OR
    Exists(p : (path?(g))) : length(p) = 2 And path_end(p) = n

% proof status :-)
% non_root_path_extend_TCC1 : TCC Obligation

% proof status :-)
non_root_path_extend : Lemma

```

```

Forall(p : (path?(g))) : roots(g)(path_start(p)) OR
Exists(q : (path?(g))) :
  length(q) = length(p) + 1 And
  path_end(q) = path_end(p)

% proof status :-)
tree_path_to_root : Lemma
  finite_tree?(g) Implies
    Forall(n : (g'nodes)) :
      Exists(p : (path?(g))) : root_path(g, n)(p)

% proof status :-)
tree_unique_path_to_root : Lemma
  Forall(n : (g'nodes), p1, p2 : (path?(g))) :
    tree?(g) And root_path(g, n)(p1) And root_path(g, n)(p2)
    Implies
      p1 = p2

% proof status :-)
root_path_singleton : Lemma Forall(n : (g'nodes)) :
  finite_tree?(g) Implies
    singleton?(root_path(g, n));

% proof status :-)
% path_to_root_TCC1 : TCC Obligation

path_to_root(g : (finite_tree?), n : (g'nodes)) : (root_path(g, n)) =
  the(root_path(g, n))

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% Path factoring. Every two path that start with the same node
% have the same initial sequence.
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% proof status :-)
% path_factor_TCC1 : TCC Obligation

% proof status :-)
% path_factor_TCC2 : TCC Obligation

% proof status :-)
% path_factor_TCC3 : TCC Obligation

% proof status :-)
% path_factor_TCC4 : TCC Obligation

% proof status :-)
path_factor : Lemma Forall (p1, p2 : (path?(g))) :
  path_start(p1) = path_start(p2) And

```

```
(Exists(n : nat) : n < length(p1) And n < length(p2) And
  NOT nth(p1, n) = nth(p2, n))
```

```
Implies
```

```
Exists(q, p1t, p2t : (path?(g))) :
  concatable?(q, p1t) And concatable?(q, p2t) And
  length(p1t) >= 2 And length(p2t) >= 2 And
  p1 = concat_path(q, p1t) And
  p2 = concat_path(q, p2t) And
  nth(p1t, 0) = nth(p2t, 0) And
  NOT nth(p1t, 1) = nth(p2t, 1)
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
% Node case distinction. Two nodes n1 and n2 can be related in the
% following way:
```

```
%
% - they are equal
% - n1 lays on the root path of n2
% - n2 lays on the root path of n1
% - the two root path lead to different roots
% (in this case the two root path are totally distinct)
% - the two root path meet at some node n3
% (in this case the two root path go to the same root)
```

```
%
% This distinction is complete and can be used to prove an arbitrary
% predicate for all pairs of nodes.
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
% proof status :-)
```

```
% node_pair_distinction_TCC1 : TCC Obligation
```

```
% proof status :-)
```

```
node_pair_distinction : Lemma
```

```
Forall(P : PRED[[ (g'nodes), (g'nodes) ]]) :
  finite_tree?(g) And
  (Forall(n : (g'nodes)) : P(n, n))
And
  (Forall(n1, n2 : (g'nodes), p : (path?(g))) :
    NOT n1 = n2 And
    path_start(p) = n2 And path_end(p) = n1 And length(p) >= 2 Implies
    P(n1, n2))
And
  (Forall(n1, n2 : (g'nodes), p : (path?(g))) :
    NOT n1 = n2 And
    path_start(p) = n1 And path_end(p) = n2 And length(p) >= 2 Implies
    P(n1, n2))
And
  (Forall(n1, n2 : (g'nodes), p1, p2 : (path?(g))):
    NOT n1 = n2 And root_path(g, n1)(p1) And root_path(g, n2)(p2) And
    NOT path_start(p1) = path_start(p2) Implies
```



```

    P(n1, n2))
And
(Forall(n1, n2, n3 : (g'nodes), p1, p2 : (path?(g))):
  NOT n1 = n2 And NOT n1 = n3 And NOT n2 = n3 And
  g'edges(n3, path_start(p1)) And g'edges(n3, path_start(p2)) And
  path_end(p1) = n1 And path_end(p2) = n2 And
  NOT path_start(p1) = path_start(p2) Implies
    P(n1, n2))
Implies
  Forall(n1, n2 : (g'nodes)) : P(n1, n2)

% Variant for symmetric P, which leaves out the case of
% - n2 lays on the root path of n1

% proof status :- )
% symmetric_node_pair_distinction_TCC1 : TCC Obligation

% proof status :- )
symmetric_node_pair_distinction : Lemma
Forall(P : PRED[[[g'nodes], (g'nodes)]]) :
  finite_tree?(g) And
  symmetric?(P) And
  (Forall(n : (g'nodes)) : P(n, n))
And
  (Forall(n1, n2 : (g'nodes), p : (path?(g))) :
    NOT n1 = n2 And
    path_start(p) = n2 And path_end(p) = n1 And length(p) >= 2 Implies
      P(n1, n2))
And
  (Forall(n1, n2 : (g'nodes), p1, p2 : (path?(g))):
    NOT n1 = n2 And root_path(g, n1)(p1) And root_path(g, n2)(p2) And
    NOT path_start(p1) = path_start(p2) Implies
      P(n1, n2))
And
  (Forall(n1, n2, n3 : (g'nodes), p1, p2 : (path?(g))):
    NOT n1 = n2 And NOT n1 = n3 And NOT n2 = n3 And
    g'edges(n3, path_start(p1)) And g'edges(n3, path_start(p2)) And
    path_end(p1) = n1 And path_end(p2) = n2 And
    NOT path_start(p1) = path_start(p2) Implies
      P(n1, n2))
Implies
  Forall(n1, n2 : (g'nodes)) : P(n1, n2)

```

End Graph

B.16 hoare.pvs

```

% $Id: hoare.pvs,v 1.8.8.1.2.2 2008/05/14 14:24:58 tews Exp $
%
% Author: Tjark Weber
% (c) 2007 Radboud University

```

```

%
% Hoare logics for our shallow C++ embedding.

Hoare[State: Type] : THEORY
BEGIN

  IMPORTING State_Transformer

  P : VAR PRED[State]
  c : VAR [State -> StmtResult[State]]
  C : VAR [StmtResult[State] -> StmtResult[State]]
  Q : VAR PRED[State]
  R : Var PRED[State]

  % Currently Hoare triples with lifted statements
  % {P} C {Q}
  % are not used. They may be useful to consider partial programs on
  % intermediate states (i.e. to verify programs compositionally), but
  % otherwise, since programs are simple statements operating on states
  % (rather than on StmtResults), only Hoare triples
  % {P} c {Q}
  % are needed.

  s : VAR State

  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
  %
  % Partial correctness: if the result is OK, it must satisfy the postcondition
  %
  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

  % proof status :-)
  % valid_TCC1 : TCC Obligation

  % proof status :-)
  % valid_TCC2 : TCC Obligation

  valid(P, c, Q) : bool =
    forall s: P(s) and OK?(c(s)) implies Q(state(c(s)))

  % proved – complete
  % valid_TCC1

  valid(P, C, Q) : bool =
    forall s: P(s) and OK?(C(OK(s))) implies Q(state(C(OK(s))))

  % proved – complete
  % valid_TCC2

  % proof status :-)
  valid_and : Lemma
    valid(P, c, Q) And valid(P, c, R)

```

Implies

valid(P, c, **Lambda** (s : State) : Q(s) **And** R(s))

% proof status : -)

valid_implied : **Lemma**

valid(P, c, Q) **And** valid(P, c, **Lambda** (s : State) : Q(s) **Implies** R(s))

Implies

valid(P, c, **Lambda** (s : State) : Q(s) **And** R(s))

%%
 %
 % Total correctness: result must be OK and satisfy the postcondition
 %
 %%%

Valid(P, c, Q) : bool =
 forall s: P(s) **implies** OK?(c(s)) **and** Q(state(c(s)))

Valid(P, C, Q) : bool =
 forall s: P(s) **implies** OK?(C(OK(s))) **and** Q(state(C(OK(s))))

% proof status : -)

Valid_and : **Lemma**

Valid(P, c, Q) **And** Valid(P, c, R)

Implies

Valid(P, c, **Lambda** (s : State) : Q(s) **And** R(s))

END Hoare

B.17 linear_memory.pvs

%%
 %
 % Split range into list of aligned chunks
 %
 %%%

% To do: move to prelude

List_Split : **Theory**

Begin

Importing IA32, More_List_Props, Alignment

% split

%-----

% | a a1 a2 a3 |

% | --|-----|-----|---- |

% =>

% (a, --), (a1, -----), (a2, -----), (a3, ----), null

```

% proof status :-)
% split_TCC1 : TCC Obligation

% proof status :-)
% split_TCC2 : TCC Obligation

split(size : nat, a : Address, bl : list[Byte]) :
  Recursive list[[Address, list[Byte]]] =
  Let e2size = expt(2, size) In
  If null?(bl) Then null Else
    If a + length(bl) <= floor(e2size)(a) + e2size Then
      cons( (a, bl), null)
    Else
      Let delta = rem(e2size)(offset(a)) In
      cons( (a, head(bl, e2size - delta)),
        split(size, floor(e2size)(a) + e2size, tail(bl, e2size - delta)))
    Endif
  Endif
  Measure length(bl)

% some correctness lemmas
% proof status :-)
split_pair_cross_size : Lemma
Forall (size : nat, a1 : Address, bl1 : list[Byte], n : nat) :
  Let e2size = expt(2, size) In
  n = length(bl1) Implies
  every(Lambda(a2 : Address, bl2 : list[Byte]) :
    rem(e2size)(offset(a2)) + length(bl2) <= e2size
  )(split(size, a1, bl1))

% proof status :-)
split_pair_concat : Lemma
Forall (size : nat, a : Address, bl : list[Byte], n : nat) :
  Let e2size = expt(2, size) In
  n = length(bl) Implies
  reduce(null,
    Lambda (e : [Address, list[Byte]], tail : list[Byte]) :
      Let head = e'2 In
      append(head, tail))(split(size, a, bl)) = bl

% proof status :-)
split_range : Lemma
Forall (size : nat, a : Address, bl : list[Byte], n : nat) :
  Let e2size = expt(2, size) In
  n = length(bl) Implies
  every(Lambda (a2 : Address, bl2 : list[Byte]) :
    a <= a2 And a2 + length(bl2) <= a + n)(split(size, a, bl))

% proof status :-)
split_no_null : Lemma
Forall (size : nat, a : Address, bl : list[Byte], n : nat) :

```

```

Let e2size = expt(2, size) In
  n = length(bl) And cons?(bl) Implies
  every(Lambda (a2 : Address, bl2 : list[Byte]) : cons?(bl2))
    (split(size, a, bl))

% proof status :-)
split_null : Lemma
  Forall (size : nat, a : Address) : split(size, a, null) = null

% proof status :-)
% split_type_TCC1 : TCC Obligation

% proof status :-)
% split_type_TCC2 : TCC Obligation

% proof status :-)
split_type : Lemma
  Forall (size : nat, a : Address, bl : list[Byte], n : nat) :
    reg_base(min_linear)(type_of(a)) <= a And
    a + length(bl) <= reg_size(max_linear)(type_of(a))
    Implies
    every(LAMBDA (t: [Address, list[Byte]]):
      type_of(t'1) = type_of(a) And
      reg_base(min_linear)(type_of(a)) <= t'1 And
      t'1 < reg_size(max_linear)(type_of(a)))
      (split(min_page, a, bl))

End List_Split

%%%%%%%%%%
%
% Linear Memory
%
%%%%%%%%%%

% to do: avoid paging_data_type

Linear_Memory [Physical_Memory : Type,
  (Importing Plain_Memory[Physical_Memory])
  pm : Plain_Memory] : Theory

Begin

  Importing Paging_Datatype,
    PDBR_Datatype,
    Segment_Reg_Datatype,
    Address_Datatype

  Importing Abstract_Read_Write_Plain,
    List_Split

  Linear_memory : Type = Physical_Memory

```

```

s : Var Linear_memory
a : Var Address
access : Var Memory_access
b : Var Byte
size : Var nat

% Protected Interface
%=====

bits_per_level : nat = 10

pdir_lvl : nat = 0
ptab_lvl : nat = 1

% proof status :-)
% xlat_idx_TCC1 : TCC Obligation

% proof status :-)
% xlat_idx_TCC2 : TCC Obligation

% compute pe address / offset
xlat_idx(lvl : Level, base, addr : Memory_Address_4G) : Address =
  base + shift_bits_left(cut_bits(offset(addr), bus_width - (lvl + 1) *
    bits_per_level, bits_per_level), pe_size)

% proof status :-)
% xlat_idx_pe_aligned_TCC1 : TCC Obligation

% proof status :-)
% xlat_idx_pe_aligned_TCC2 : TCC Obligation

% proof status :-)
xlat_idx_pe_aligned : Lemma
  Forall (lvl : Level, base, addr : Memory_Address_4G) :
    aligned?(bits_per_level + pe_size)(offset(base))
  Implies
    aligned?(pe_size)(offset(xlat_idx(lvl, base, addr)))

% proof status :-)
xlat_idx_memory_address : Lemma
  Forall (lvl : Level, base, addr : Memory_Address_4G) :
    aligned?(bits_per_level + pe_size)(offset(base))
  Implies
    xlat_idx(lvl, base, addr) < max_linear

% proof status :-)
xlat_idx_memory_address2 : Lemma
  Forall (lvl : Level, base, addr : Memory_Address_4G) :
    aligned?(bits_per_level + pe_size)(offset(base))
  Implies
    0 <= offset(xlat_idx(lvl, base, addr))

```

```

xlat_ofs(lvl : Level, base, addr : Memory_Address_4G) : Address =
  base + cut_bits(offset(addr), 0, bus_width - (lvl + 1) * bits_per_level)

% proof status :-)
xlat_ofs_memory_address : Lemma
  Forall (lvl : Level, base, addr : Memory_Address_4G) :
    aligned?(bus_width - (lvl + 1) * bits_per_level)(offset(base))
  Implies
    xlat_ofs(lvl, base, addr) < max_linear

% proof status :-)
xlat_ofs_memory_address2 : Lemma
  Forall (lvl : Level, base, addr : Memory_Address_4G) :
    aligned?(bus_width - (lvl + 1) * bits_per_level)(offset(base))
  Implies
    0 <= offset(xlat_ofs(lvl, base, addr))

% proof status :-)
% raise_fault_TCC1 : TCC Obligation

% raise_fault
raise_fault(priv : Memory_privilege, access : Memory_access, present : bool,
  pfa : Memory_Address_4G) :
  [Physical_memory -> ExprResult[Physical_memory, [bool, Address]]] =
  write_data(pm, address_data_type)(CR2, pfa) ##
  exception_result(Page_fault(
    (# reserved_bit_violation := false, % writing reserved bits is fatal
    privilege := priv,
    access := add(Read, access),
    present := present
    #)))

% proof status :-)
% translate_TCC1 : TCC Obligation

% single level of the translation
translate(lvl : Level, base, addr : Memory_Address_4G,
  access : Memory_access, priv : Memory_privilege)
: [Physical_memory -> ExprResult[Physical_memory, [bool, Address]]] =
Let pe_addr = xlat_idx(lvl, base, addr) In
  (read_data(pm, paging_data_type(lvl))(pe_addr) ##
  Lambda (pe : (range_pt(lvl))) :
    If paging_type?(lvl, pe) Then
      If present?(pe) Then
        If accessible?(pe, access) And privileged?(pe, priv) Then
          write_data(pm, paging_data_type(lvl))(pe_addr,
            set_reference(pe, access)) ##
            ok_result( (is_leaf?(pe), base(pe)) )
        Else
          raise_fault(priv, access, true, addr)
        Endif
      Endif
    )

```

```

    Else
      raise_fault(priv, access, false, addr)
    Endif
  Else
    fatal_result
  Endif)

% proof status :-)
translate_memory_address : Lemma
  Forall (lvl : Level, base, addr : Memory_Address_4G,
    access : Memory_access,
    priv : Memory_privilege, s : Linear_memory) :
    aligned?(bits_per_level + pe_size)(offset(base)) And
    OK?(translate(lvl, base, addr, access, priv)(s))
  Implies
    data(translate(lvl, base, addr, access, priv)(s))'2 < max_linear

% proof status :-)
translate_memory_address2 : Lemma
  Forall (lvl : Level, base, addr : Memory_Address_4G,
    access : Memory_access,
    priv : Memory_privilege, s : Linear_memory) :
    aligned?(bits_per_level + pe_size)(offset(base)) And
    OK?(translate(lvl, base, addr, access, priv)(s))
  Implies
    0 <= offset(data(translate(lvl, base, addr, access, priv)(s))'2)

% proof status :-)
% translate_result_leaf_TCC1 : TCC Obligation

% proof status :-)
% translate_result_leaf_TCC2 : TCC Obligation

% !!! translate_result_leaf_TCC1 : original long running proof is nondefault !!
% proof status :-)
translate_result_leaf : Lemma
  Forall (lvl : Level, base, addr : Memory_Address_4G,
    access : Memory_access,
    priv : Memory_privilege, s : Linear_memory) :
    aligned?(bits_per_level + pe_size)(offset(base)) And
    OK?(translate(lvl, base, addr, access, priv)(s)) And
    data(translate(lvl, base, addr, access, priv)(s))'1
  Implies
    aligned?(bus_width - bits_per_level * (lvl + 1))
      (offset(data(translate(lvl, base, addr, access, priv)(s))'2))

% proof status :-)
% translate_result_no_leaf_TCC1 : TCC Obligation

% proof status :-)
translate_result_no_leaf : Lemma

```


B PVS Theory Sources

```

Forall (lvl : Level, base, addr : Memory_Address_4G,
        access : Memory_access,
        priv : Memory_privilege, s : Linear_memory) :
  aligned?(bits_per_level + pe_size)(offset(base)) And
  OK?(translate(lvl, base, addr, access, priv)(s)) And
  Not data(translate(lvl, base, addr, access, priv)(s))'1
Implies
  aligned?(bits_per_level + pe_size)
  (offset(data(translate(lvl, base, addr, access, priv)(s))'2))

% proof status :-)
% linear_resolve_TCC1 : TCC Obligation

% proof status :-)
% linear_resolve_TCC2 : TCC Obligation

% proof status :-)
% linear_resolve_TCC3 : TCC Obligation

% proof status :-)
% linear_resolve_TCC4 : TCC Obligation

% proof status :-)
% linear_resolve_TCC5 : TCC Obligation

% proof status :-)
% linear_resolve_TCC6 : TCC Obligation

% proof status :-)
% linear_resolve_TCC7 : TCC Obligation

% proof status :-)
% linear_resolve_TCC8 : TCC Obligation

% translate
linear_resolve(addr : Memory_Address_4G, access : Memory_access)
  : [Linear_memory -> ExprResult[Linear_memory, Address]] =
  % 1. get current privilege level from cs
  (read_data(pm, segment_reg_data_type)(CS) ###
  Lambda (cs : Segment_Reg_type) :
    Let priv = segment_to_priv(cs) In
    % 2. read PDBR
    (read_data(pm, pdbr_data_type)(PDBR) ###
    Lambda (pdbr : Pdbr_type) :
      % 3. PDE lookup
      Let lvl = pdir_lvl In
      (translate(lvl, pdbr'base_addr, addr, access, priv) ###
      Lambda (leaf : bool,
              base : {b : Memory_Address_4G |
                If leaf Then
                  aligned?(bus_width - bits_per_level * (lvl + 1))(offset(b))

```

```

    Else
      aligned?(bits_per_level + pe_size)(offset(b))
    Endif} :
  If leaf Then
    ok_result(xlat_ofs(lvl, base, addr))
  Else
    Let lvl = ptab_lvl In
      (translate(lvl, base, addr, access, priv) ##
        Lambda (leaf : bool,
              base : Memory_Address_4G) :
          ok_result(xlat_ofs(lvl, base, addr)))
    Endif)))

% proof status :-)
linear_resolve_memory_address : Judgement
linear_resolve(addr : Memory_Address_4G, access : Memory_access) Has_Type
  [Linear_memory -> ExprResult[Linear_memory, Memory_Address_4G]]

% side effect helpers
linear_read_side_effect_in_page(a : Memory_Address_4G, bl : list[Byte])(s) :
  ExprResult[Linear_memory, list[Byte]] =
  (linear_resolve(a, Read) ##
    Lambda(pa : Memory_Address_4G)(ns : Linear_memory) :
      memory_read_side_effect(pm'mem)(pa, bl, true)(s))(s)

linear_write_side_effect_in_page(a : Memory_Address_4G, bl : list[Byte])(s) :
  ExprResult[Linear_memory, list[Byte]] =
  (linear_resolve(a, Write) ##
    Lambda(pa : Memory_Address_4G)(ns : Linear_memory) :
      memory_write_side_effect(pm'mem)(pa, bl, true)(s))(s)

apply_side_effects(l : list[[Memory_Address_4G, list[Byte]]],
  f : [Memory_Address_4G, list[Byte]] ->
  [Linear_memory -> ExprResult[Linear_memory, list[Byte]]])(s) :
  ExprResult[Linear_memory, list[Byte]] =
  reduce(ok_result(null),
    Lambda (e : [Memory_Address_4G, list[Byte]],
          r : [Linear_memory -> ExprResult[Linear_memory, list[Byte]]]) :
      (r ##
        Lambda (tail : list[Byte]) :
          (f(e) ##
            Lambda (head : list[Byte]) : ok_result(append(head, tail))))
      )(l)(s)
  )
% Note that the element with the highest address is evaluated first

% Public Interface
%=====

% proof status :-)
% linear_read_TCC1 : TCC Obligation

% proof status :-)

```

B PVS Theory Sources

```
% linear_read_TCC2 : TCC Obligation

% proof status :-)
% linear_read_TCC3 : TCC Obligation

% read & write
linear_read(a) : [Linear_memory -> ExprResult[Linear_memory, Byte]] =
  If in_memory(min_linear, max_linear)(a) Then
    If Mem?(type_of(a)) Then
      linear_resolve(a, Read) ##
      Lambda (pa : Address) :
        memory_read(pm'mem)(pa)
    Else
      memory_read(pm'mem)(a)
    Endif
  Else
    fatal_result
  Endif

linear_write(a, b) : [Linear_memory -> ExprResult[Linear_memory, Unit]] =
  If in_memory(min_linear, max_linear)(a) Then
    If Mem?(type_of(a)) Then
      linear_resolve(a, Write) ##
      Lambda (pa : Address) :
        memory_write(pm'mem)(pa, b)
    Else
      memory_write(pm'mem)(a, b)
    Endif
  Else
    fatal_result
  Endif

% proof status :-)
% linear_read_side_effect_TCC1 : TCC Obligation

% Linear Side Effects
linear_read_side_effect(a : Address, bl : list[Byte], cp : bool)(s)
  : ExprResult[Linear_memory, list[Byte]] =
  If null?(bl) Or
    (reg_base(min_linear)(type_of(a)) <= a And
     a + length(bl) <= reg_size(max_linear)(type_of(a))) Then
    If Mem?(type_of(a)) Then
      % split list in page contained lists
      apply_side_effects(split(min_page, a, bl),
        linear_read_side_effect_in_page)(s)
    Else
      memory_read_side_effect(pm'mem)(a, bl, cp)(s)
    Endif
  Else
    Fatal
  Endif
```

```

linear_write_side_effect(a : Address, bl : list[Byte], cp : bool)(s)
    : ExprResult[Linear_memory, list[Byte]] =
  If null?(bl) Or
    (reg_base(min_linear)(type_of(a)) <= a And
     a + length(bl) <= reg_size(max_linear)(type_of(a))) Then
  If Mem?(type_of(a)) Then
    % split list in page contained lists
    apply_side_effects(split(min_page, a, bl),
                       linear_write_side_effect_in_page)(s)
  Else
    memory_write_side_effect(pm'mem)(a, bl, cp)(s)
  Endif
Else
  Fatal
Endif

% The memory struct
linear_pm : Memory_struct[Linear_memory] = (#
  memory_read := linear_read,
  memory_write := linear_write,

  memory_read_side_effect := linear_read_side_effect,
  memory_write_side_effect := linear_write_side_effect
#)

END Linear_Memory

```

B.18 memory.pvs

% \$Id: memory.pvs,v 1.55.6.1.4.3 2008/05/14 14:24:59 tewes Exp \$

Address_Util : Theory

Begin

IMPORTING IA32

IMPORTING More_Sets_Lemmas % for disjoint_mono

Importing More_List_Props

%%%

% address blocks

%%%

% the set of addresses $addr \leq a < addr + size$

address_block(addr : Address, size : nat) : PRED[Address] =

Lambda(a : Address) :

type_of(addr) = type_of(a) And

offset(addr) <= offset(a) AND

offset(a) < offset(addr) + size

% a lemma needed in the induction below

```

% proof status :-)
address_block_subset_1 : Lemma Forall(addr : Address, size : nat) :
  subset?(address_block(addr + 1, size),
    address_block(addr, 1 + size))

% proof status :-)
address_block_subset_2 : Lemma
  Forall(addresses : PRED[Address], addr : Address, size : nat) :
  subset?(address_block(addr, size + 1), addresses) Implies
  subset?(address_block(addr + 1, size), addresses)

blocks_disjoint?(addr1 : Address, size1 : nat,
  addr2 : Address, size2 : nat) : bool =
  type_of(addr1) /= type_of(addr2) Or
  addr1 + size1 <= addr2 Or
  addr2 + size2 <= addr1

% proof status :-)
blocks_disjoint_disjoint : Lemma
  Forall (addr1, addr2 : Address, size1, size2 : nat) :
  size1 > 0 And size2 > 0 Implies
  blocks_disjoint?(addr1, size1, addr2, size2) =
  disjoint?(address_block(addr1, size1), address_block(addr2, size2))

% proof status :-)
blocks_disjoint_symmetric : Lemma
  Forall (addr1, addr2 : Address, size1, size2 : nat) :
  blocks_disjoint?(addr1, size1, addr2, size2) =
  blocks_disjoint?(addr2, size2, addr1, size1)

% proof status :-)
blocks_in_larger_set : Lemma
  Forall (addr1, addr2 : Address, size1, size2 : nat,
    addresses : PRED[Address]) :
  subset?(address_block(addr1, size1), addresses) And
  subset?(address_block(addr2, size2), addresses) And
  blocks_disjoint?(addr1, size1, addr2, size2)
  Implies
  subset?(address_block(addr2, size2),
    difference(addresses, address_block(addr1, size1)))

blocks_pairwise_disjoint(blocks : list[[Address, posnat]]) : bool =
  Forall(addr_1, addr_2 : Address, size_1, size_2 : posnat) :
  member((addr_1, size_1), blocks) And
  member((addr_2, size_2), list_remove((addr_1, size_1), blocks))
  Implies
  blocks_disjoint?(addr_1, size_1, addr_2, size_2)

% proof status :-)
blocks_pairwise_disjoint_add : Lemma

```

```

Forall(addr : Address, size : posnat, blocks : list[[Address, posnat]]) :
  blocks_pairwise_disjoint(blocks) And
  (Forall(oaddr : Address, osize : posnat) :
    member((oaddr, osize), blocks) Implies
      blocks_disjoint?(addr, size, oaddr, osize))
Implies
  blocks_pairwise_disjoint(cons((addr, size), blocks))

```

% proof status :-)

blocks_pairwise_disjoint_remove : **Lemma**

```

Forall(addr : Address, size : posnat, blocks : list[[Address, posnat]]) :
  blocks_pairwise_disjoint(blocks) Implies
  blocks_pairwise_disjoint(list_remove((addr, size), blocks))

```

```

block_is_free(block : PRED[Address],
  alloc_points : list[[Address, posnat]]) : bool =

```

```

Forall(addr : Address, size : posnat) :
  member((addr, size), alloc_points) Implies
  disjoint?(block, address_block(addr, size))

```

% proof status :-)

subset_block_is_free : **Lemma**

```

Forall(b1, b2 : PRED[Address], alloc_points : list[[Address, posnat]]) :
  subset?(b1, b2) And
  block_is_free(b2, alloc_points) Implies
  block_is_free(b1, alloc_points)

```

% proof status :-)

block_is_free_add : **Lemma**

```

Forall(block : PRED[Address], alloc_points : list[[Address, posnat]],
  addr : Address, size : posnat) :
  block_is_free(block, alloc_points) And
  disjoint?(block, address_block(addr, size))
Implies
  block_is_free(block, cons((addr, size), alloc_points))

```

End Address_Util

Memory[State : **Type**] : **Theory**

%

% memory is a common memory abstraction

% it comes with read and write functions

%

% this theory defines the notion of memory and

% some utility functions/lemmas

%

Begin

IMPORTING Unit, Address_Util, State_Transformer

%%%

```

%
% memory definition
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
Memory_struct : Type =
  [#
    memory_read : [Address ->
                    [State -> ExprResult[State, Byte]]],
    memory_write : [Address, Byte ->
                    [State -> ExprResult[State, Unit]]],

    memory_read_side_effect : [Address, list[Byte], bool ->
                               [State -> ExprResult[State, list[Byte]]]],
    memory_write_side_effect : [Address, list[Byte], bool ->
                               [State -> ExprResult[State, list[Byte]]]]
  #]

s : Var State
pm : Var Memory_struct
bl : Var list[Byte]
addr : Var Address
size : Var nat

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% write_list
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% proof status :-)
% memory_write_list_nse_TCC1 : TCC Obligation

% proof status :-)
% memory_write_list_nse_TCC2 : TCC Obligation

% side effect free list write
memory_write_list_nse(pm)(addr, bl) : Recursive
                                     [State -> ExprResult[State, Unit]] =
Cases bl of
  null : ok_result(unit),
  cons(b, rl) :
    (memory_write(pm)(addr, b) ##
     memory_write_list_nse(pm)(addr + 1, cdr(bl)))
EndCases
Measure length(bl)

% apply write side effect before register is modified
memory_write_list(pm)(addr, bl) : [State -> ExprResult[State, Unit]] =
  memory_write_side_effect(pm)(addr, bl, false) ##
  Lambda (bl1 : list[Byte]) :

```

```

memory_write_list_nse(pm)(addr, bl1)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% read_list
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% proof status :-)
% memory_read_list_nse_TCC1 : TCC Obligation

% proof status :-)
% memory_read_list_nse_TCC2 : TCC Obligation

% side effect free list read
memory_read_list_nse(pm)(addr, size) : Recursive
                                     [State -> ExprResult[State, list[Byte]]] =
If size > 0 Then
  memory_read(pm)(addr) ##
  Lambda (b : Byte) :
    (memory_read_list_nse(pm)(addr + 1, size - 1) ##
     Lambda (res_list: list[Byte]) :
       ok_result(cons(b, res_list)))
Else
  ok_result(null)
Endif
Measure size

% apply side effect on read content
memory_read_list(pm)(addr, size) : [State -> ExprResult[State, list[Byte]]] =
  memory_read_list_nse(pm)(addr, size) ##
  Lambda (bl1 : list[Byte]) :
    memory_read_side_effect(pm)(addr, bl1, false)

% proof status :-)
memory_read_list_ok_length : Lemma
  OK?(memory_read_list_nse(pm)(addr, size)(s))
IMPLIES
  length(data(memory_read_list_nse(pm)(addr, size)(s))) = size

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% memory_read transformers (sets of memory_read's)
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

addresses : Var PRED[Address]

% the set of state transformers that read at addresses

```



```

memory_read_transformers(pm, addresses) :
    PRED[[State -> SuperResult[State]]] =
Lambda(q : [State -> SuperResult[State]]) :
    Exists(a : Address) : addresses(a) AND
        q = expr_2_super(memory_read(pm)(a))

% the obvious rewrite lemma
% proof status :-)
memory_read_transformers_memory_read : Lemma Forall(addr : Address) :
    addresses(addr)
IMPLIES
    memory_read_transformers(pm, addresses)
        (expr_2_super(memory_read(pm)(addr)))

% Monotonicity
% proof status :-)
memory_read_transformers_mono : Lemma
Forall(addresses_1, addresses_2 : PRED[Address]) :
    subset?(addresses_1, addresses_2) IMPLIES
        subset?(memory_read_transformers(pm, addresses_1),
            memory_read_transformers(pm, addresses_2))

% proof status :-)
memory_read_transformers_union : Lemma
Forall(addresses_1, addresses_2 : PRED[Address]):
    union(memory_read_transformers(pm, addresses_1),
        memory_read_transformers(pm, addresses_2))
    =
        memory_read_transformers(pm, union(addresses_1, addresses_2))

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% memory_write transformers (sets of memory_write's)
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% the set of state transformers that read at addresses
memory_write_transformers(pm, addresses) :
    PRED[[State -> SuperResult[State]]] =
Lambda(q : [State -> SuperResult[State]]) :
    Exists(a : Address, b : Byte) :
        addresses(a) AND q = expr_2_super(memory_write(pm)(a,b))

% the obvious rewrite lemma
% proof status :-)
memory_write_transformers_memory_write : Lemma
Forall(addr : Address, b : Byte) :
    addresses(addr)

```

IMPLIES

```
memory_write_transformers(pm, addresses)
      (expr_2_super(memory_write(pm)(addr, b)))
```

```
% Monotonicity
```

```
% proof status :-)
```

```
memory_write_transformers_mono : Lemma
```

```
Forall(addresses_1, addresses_2 : PRED[Address]) :
  subset?(addresses_1, addresses_2) IMPLIES
  subset?(memory_write_transformers(pm, addresses_1),
    memory_write_transformers(pm, addresses_2))
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
% side effect transformers
```

```
% (sets of memory_read_side_effect and memory_write_side_effect
```

```
%
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
states : Var PRED[State]
```

```
transformers : Var PRED[[State -> ExprResult[State, list[Byte]]]]
```

```
memory_read_side_effect_super_transformers(pm, addresses) :
```

```
      PRED[[State -> SuperResult[State]]] =
```

```
Lambda(q : [State -> SuperResult[State]]) :
```

```
Exists(a : Address, bl : list[Byte], cp : bool) :
  subset?(address_block(a, length(bl)), addresses) And
  q = expr_2_super(memory_read_side_effect(pm)(a, bl, cp))
```

```
% proof status :-)
```

```
memory_read_side_effect_super_transformers_memory_read_side_effect : Lemma
```

```
Forall(a : Address, bl : list[Byte], cp : bool) :
  subset?(address_block(a, length(bl)), addresses) Implies
  memory_read_side_effect_super_transformers(pm, addresses)
  (expr_2_super(memory_read_side_effect(pm)(a, bl, cp)))
```

```
% proof status :-)
```

```
memory_read_side_effect_super_transformers_mono : Lemma
```

```
Forall(addresses_1, addresses_2 : PRED[Address]) :
  subset?(addresses_1, addresses_2) IMPLIES
  subset?(memory_read_side_effect_super_transformers(pm, addresses_1),
    memory_read_side_effect_super_transformers(pm, addresses_2))
```

```
memory_write_side_effect_super_transformers(pm, addresses) :
```

```
      PRED[[State -> SuperResult[State]]] =
```

```
Lambda(q : [State -> SuperResult[State]]) :
```

```
Exists(a : Address, bl : list[Byte], cp : bool) :
  subset?(address_block(a, length(bl)), addresses) And
  q = expr_2_super(memory_write_side_effect(pm)(a, bl, cp))
```

```

% proof status :-)
memory_write_side_effect_super_transformers_memory_write_side_effect : Lemma
  Forall(a : Address, bl : list[Byte], cp : bool) :
    subset?(address_block(a, length(bl)), addresses) Implies
      memory_write_side_effect_super_transformers(pm, addresses)
        (expr_2_super(memory_write_side_effect(pm)(a, bl, cp)))

% proof status :-)
memory_write_side_effect_super_transformers_mono : Lemma
  Forall(addresses_1, addresses_2 : PRED[Address]) :
    subset?(addresses_1, addresses_2) IMPLIES
      subset?(memory_write_side_effect_super_transformers(pm, addresses_1),
        memory_write_side_effect_super_transformers(pm, addresses_2))

se_transformer : Var
  [Address, list[Byte], bool -> [State -> ExprResult[State, list[Byte]]]]

se_transformer1 : Var
  [Address, list[Byte], bool -> [State -> ExprResult[State, list[Byte]]]]

se_transformer2 : Var
  [Address, list[Byte], bool -> [State -> ExprResult[State, list[Byte]]]]

se_super_transformers(addresses : PRED[Address],
  se_transformer : [Address, list[Byte], bool ->
    [State -> ExprResult[State, list[Byte]]]]) :
  PRED[[State -> SuperResult[State]]] =
Lambda (q : [State -> SuperResult[State]]) :
  Exists(a : Address, bl : list[Byte], cp : bool) :
    subset?(address_block(a, length(bl)), addresses) And
      q = expr_2_super(se_transformer(a, bl, cp))

% side effect transformers do not change the data read or to be written
side_effect_content_unchanged(addresses : PRED[Address], states,
  se_transformer) : bool =
Forall(s : State, a : Address, bl : list[Byte], cp : bool) :
  states(s) And
    subset?(address_block(a, length(bl)), addresses) And
      OK?(se_transformer(a, bl, cp)(s)) Implies
        data(se_transformer(a, bl, cp)(s)) = bl

% simple access lemma
% proof status :-)
side_effect_content_unchanged_content : Lemma
  Forall(addresses : PRED[Address],
    s : State, a : Address, bl : list[Byte], cp : bool) :
    side_effect_content_unchanged(addresses, states, se_transformer) And
      states(s) And
        subset?(address_block(a, length(bl)), addresses) And
          OK?(se_transformer(a, bl, cp)(s)) Implies
            data(se_transformer(a, bl, cp)(s)) = bl

```

```

% proof status :-)
side_effect_content_unchanged_mono : Lemma
  Forall (addresses_1, addresses_2 : PRED[Address]) :
    subset?(addresses_1, addresses_2) And
    side_effect_content_unchanged(addresses_2, states, se_transformer)
    Implies
    side_effect_content_unchanged(addresses_1, states, se_transformer)

% proof status :-)
side_effect_content_unchanged_composition : Lemma
  transformer_invariant?(states,
    se_super_transformers(addresses, se_transformer1)) And
  side_effect_content_unchanged(addresses, states, se_transformer1) And
  side_effect_content_unchanged(addresses, states, se_transformer2)
Implies
  side_effect_content_unchanged(addresses, states,
    Lambda(a : Address, bl : list[Byte], cp : bool) :
      se_transformer1(a, bl, cp) ## Lambda (bl1 : list[Byte]) :
        se_transformer2(a, bl1, cp))

```

end Memory

Memory_Change[State : **Type**] : **Theory**

```

%
% this theory defines the unchanged abstraction
%

```

Begin

IMPORTING Memory

```

pm : Var Memory_struct[State]
states : VAR PRED[State]
transformers : Var PRED[[State -> SuperResult[State]]]
addresses : Var PRED[Address]

```

```

% proof status :-)
% unchanged_memory_invariant?_TCC1 : TCC Obligation

```

```

% Unchanged abstraction: For
% - an arbitrary memory,
% - a set of states
% - a set of state transformers and
% - a set of addresses
% unchanged_memory_invariant? holds if
% - states form an invariant wrt. all elements of transformers
% - all transformers leave the values at all addresses unchanged
unchanged_memory_invariant?(pm, states, transformers, addresses) : bool =
  transformer_invariant?(states, transformers) AND
Forall(s : State, q : [State -> SuperResult[State]], a : Address) :
  states(s) AND transformers(q) AND addresses(a) AND
  OK?(q(s)) AND

```

B PVS Theory Sources

```
OK?(memory_read(pm)(a)(s)) AND
OK?(memory_read(pm)(a)(state(q(s))))
IMPLIES
  data(memory_read(pm)(a)(state(q(s)))) =
    data(memory_read(pm)(a)(s))

% proof status :-)
% unchanged_memory_invariant_unchanged_TCC1 : TCC Obligation

% access lemma
% proof status :-)
unchanged_memory_invariant_unchanged : Lemma
Forall(s : State, q : [State -> SuperResult[State]], a : Address) :
  unchanged_memory_invariant?(pm, states, transformers, addresses) AND
  states(s) AND transformers(q) AND addresses(a) AND
  OK?(q(s)) AND
  OK?(memory_read(pm)(a)(s)) AND
  OK?(memory_read(pm)(a)(state(q(s))))
IMPLIES
  data(memory_read(pm)(a)(state(q(s)))) =
    data(memory_read(pm)(a)(s))

% proof status :-)
unchanged_memory_invariant_invariant : Lemma
  unchanged_memory_invariant?(pm, states, transformers, addresses) Implies
  transformer_invariant?(states, transformers)

% Monotonicity lemma
% proof status :-)
unchanged_memory_invariant_mono : Lemma
Forall(transformers_1, transformers_2 : PRED[[State -> SuperResult[State]]],
  addresses_1, addresses_2 : PRED[Address]) :
  subset?(transformers_1, transformers_2) AND
  subset?(addresses_1, addresses_2) AND
  unchanged_memory_invariant?(pm, states, transformers_2, addresses_2)
IMPLIES
  unchanged_memory_invariant?(pm, states, transformers_1, addresses_1)

% proof status :-)
% unchanged_memory_invariant_next_ok_TCC1 : TCC Obligation

% invariant lemma
% proof status :-)
unchanged_memory_invariant_next_ok : LEMMA
FORALL (s: State, q: [State -> SuperResult[State]]):
  unchanged_memory_invariant?(pm, states, transformers, addresses) AND
  states(s) AND transformers(q) AND has_next_state(q(s))
IMPLIES states(state(q(s)))

% lemma for union of transformers
```

```

% proof status :-)
unchanged_memory_invariant_union_transformers : Lemma
  Forall(transformers_1, transformers_2 :
    PRED[[State -> SuperResult[State]]]) :
    unchanged_memory_invariant?(pm, states, transformers_1, addresses) AND
    unchanged_memory_invariant?(pm, states, transformers_2, addresses)
IMPLIES
    unchanged_memory_invariant?(pm, states,
      union(transformers_1, transformers_2), addresses)

```

```

% lemma for union of address sets
% proof status :-)
unchanged_memory_invariant_union_addresses : Lemma
  Forall(addresses_1, addresses_2 : PRED[Address]) :
    unchanged_memory_invariant?(pm, states, transformers, addresses_1) AND
    unchanged_memory_invariant?(pm, states, transformers, addresses_2)
IMPLIES
    unchanged_memory_invariant?(pm, states, transformers,
      union(addresses_1, addresses_2))

```

```

% An empty address set does not automatically imply an
% unchanged_memory_invariant, because states and transformers must
% form a transformer invariant (independent of any address).
% proof status :-)
unchanged_memory_invariant_empty : Lemma
  empty?(states) OR empty?(transformers) Implies
    unchanged_memory_invariant?(pm, states, transformers, addresses)

```

```

% proof status :-)
unchanged_memory_invariant_all_transformers : Lemma
  (Forall (q : [State -> SuperResult[State]]) : transformers(q) Implies
    unchanged_memory_invariant?(pm, states, singleton(q), addresses))
Implies
  unchanged_memory_invariant?(pm, states, transformers, addresses)

```

End Memory_Change

```

Memory_Change_3[State : Type, Data : Type] : Theory
% repeat a few lemmas from above for non-super transformers

```

Begin

```

Importing Memory_Change

```

```

pm : Var Memory_struct[State]
states : VAR PRED[State]
transformers : Var PRED[[State -> SuperResult[State]]]
addresses : Var PRED[Address]

```

```

% proof status :-)
% expr_unchanged_memory_invariant_unchanged_TCC1 : TCC Obligation

```

B PVS Theory Sources

```

% proof status :-)
expr_unchanged_memory_invariant_unchanged : Lemma
  Forall(s : State, q : [State -> ExprResult[State, Data]], a : Address) :
    unchanged_memory_invariant?(pm, states, transformers, addresses) AND
    states(s) AND transformers(expr_2_super(q)) AND addresses(a) AND
    OK?(q(s)) AND
    OK?(memory_read(pm)(a)(s)) AND
    OK?(memory_read(pm)(a)(state(q(s))))
  IMPLIES
    data(memory_read(pm)(a)(state(q(s)))) =
      data(memory_read(pm)(a)(s))

% proof status :-)
% expr_unchanged_memory_invariant_next_ok_TCC1 : TCC Obligation

% proof status :-)
expr_unchanged_memory_invariant_next_ok: LEMMA
  FORALL (s: State, q: [State -> ExprResult[State, Data]]):
    unchanged_memory_invariant?(pm, states, transformers, addresses) AND
    states(s) AND transformers(expr_2_super(q)) AND has_next_state(q(s))
  IMPLIES states(state(q(s)))

End Memory_Change_3

Memory_Change_4[State : Type, Data1, Data2 : Type] : Theory
Begin

  Importing Memory_Change_3

  pm : Var Memory_struct[State]
  states : Var PRED[State]
  transformers_1 : Var PRED[[State -> SuperResult[State]]]
  transformers_2 : Var PRED[[State -> SuperResult[State]]]
  addresses : Var PRED[Address]

  % proof status :-)
  expr_unchanged_memory_invariant_composition : Lemma
    Forall (q1 : [State -> ExprResult[State, Data1]],
      q2 : [State -> ExprResult[State, Data2]]) :
      transformers_1(expr_2_super(q1)) And transformers_2(expr_2_super(q2)) And
      transformers_ok?(states, memory_read_transformers(pm, addresses)) And
      unchanged_memory_invariant?(pm, states, transformers_1, addresses) And
      unchanged_memory_invariant?(pm, states, transformers_2, addresses)
    Implies
      unchanged_memory_invariant?(pm, states, singleton(expr_2_super(q1 ## q2)), addresses)

  % proof status :-)
  fexpr_unchanged_memory_invariant_composition : Lemma
    Forall (q1 : [State -> ExprResult[State, Data1]],
      q2 : [Data1 -> [State -> ExprResult[State, Data2]]],
      P : PRED[Data1]) :

```

```

transformers_1(expr_2_super(q1)) And
transformers_ok?(states, memory_read_transformers(pm, addresses)) And
unchanged_memory_invariant?(pm, states, transformers_1, addresses) And
(Forall (s : (states)) : OK?(q1(s)) Implies P(data(q1(s)))) And
(Forall (d : (P)) : unchanged_memory_invariant?(pm, states,
                                         singleton(expr_2_super(q2(d))), addresses))

```

Implies

```

unchanged_memory_invariant?(pm, states, singleton(expr_2_super(q1 ## q2)), addresses)

```

End Memory_Change_4

Memory_Change_2[State : **Type**] : **Theory**

```

%
% Here the unchanged abstraction is further specialized.
% There is also a changed abstraction (for memory writes)
% (all this cannot go in the previous theory because of
% the type parameter Data there)
%
% in the end all this is used to prove results about
% memory_read_list after memory_write_list
% and
% memory_read_list after memory_read_list
%

```

Begin

IMPORTING Memory_Change_3

pm : **Var** Memory_struct[State]

states : **VAR** PRED[State]

addresses : **Var** PRED[Address]

```

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% special unchanged for memory_write
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

```

```

% The following is a variation of
%
% unchanged_memory_invariant?(pm, states,
% memory_write_transformers(pm, addresses),
% addresses)
%
% The variation is that it excludes the obviously wrong cases:
% writing at a _changes_ address a.
unchanged_memory_write_invariant?(pm, states, addresses) : bool =
Forall(waddr : Address, b : Byte) :
  addresses(waddr) IMPLIES
    unchanged_memory_invariant?(pm, states,
      singleton(expr_2_super(memory_write(pm)(waddr,b))),

```



```

remove(waddr, addresses))

% Monotonicity
% proof status :-)
unchanged_memory_write_invariant_mono_addresses : Lemma
  Forall(addresses_1, addresses_2 : PRED[Address]) :
    subset?(addresses_1, addresses_2) AND
    unchanged_memory_write_invariant?(pm, states, addresses_2)
  IMPLIES
    unchanged_memory_write_invariant?(pm, states, addresses_1)

% connect unchanged_memory_write_invariant? with transformer_invariant?
% proof status :-)
unchanged_memory_write_invariant_transformer_invariant : Lemma
  unchanged_memory_write_invariant?(pm, states, addresses)
  IMPLIES
    transformer_invariant?(states, memory_write_transformers(pm, addresses))

% connect unchanged_memory_write_invariant with unchanged_memory_invariant?
% for a subset of unchanged addresses
% proof status :-)
unchanged_memory_write_invariant_unchanged_memory_invariant : Lemma
  Forall(all_addresses, unchanged_addresses,
    changed_addresses : PRED[Address]) :
    unchanged_memory_write_invariant?(pm, states, all_addresses) AND
    subset?(unchanged_addresses, all_addresses) AND
    subset?(changed_addresses, all_addresses) AND
    disjoint?(unchanged_addresses, changed_addresses)
  IMPLIES
    unchanged_memory_invariant?(pm, states,
      memory_write_transformers(pm, changed_addresses),
      unchanged_addresses)

% connect unchanged_memory_invariant and unchanged_memory_write_invariant
% to a larger unchanged_memory_invariant
% proof status :-)
unchanged_memory_invariant_union : Lemma
  Forall (ro_addr, rw_addr, mod_addr : PRED[Address]) :
    unchanged_memory_invariant?(pm, states,
      memory_write_transformers(pm, rw_addr), ro_addr) And
    unchanged_memory_write_invariant?(pm, states, rw_addr) And
    subset?(mod_addr, rw_addr)
  Implies
    unchanged_memory_invariant?(pm, states,
      memory_write_transformers(pm, mod_addr),
      difference(union(ro_addr, rw_addr), mod_addr))

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% some invariant lemmas for write_list

```

```

%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% wrt transformer_invariant
% proof status :-)
transformer_invariant_write_list_nse : Lemma
  Forall(waddr : Address, bl : list[Byte], s : State) :
    transformer_invariant?(states,
      memory_write_transformers(pm, address_block(waddr, length(bl))))
  AND
    states(s)
IMPLIES
  result_pred(states)
    (expr_2_super(memory_write_list_nse(pm)(waddr, bl))(s))

% proof status :-)
transformer_invariant_write_list : Lemma
  Forall(waddr : Address, bl : list[Byte], s : State) :
    transformer_invariant?(states,
      add(
        expr_2_super(memory_write_side_effect(pm)(waddr, bl, false)),
        memory_write_transformers(pm, address_block(waddr, length(bl))))))
  And
    side_effect_content_unchanged(address_block(waddr, length(bl)), states,
      memory_write_side_effect(pm))
  And
    states(s)
Implies
  result_pred(states)(expr_2_super(memory_write_list(pm)(waddr, bl))(s))

% wrt unchanged_memory_invariant? and read_memory
% see below

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% changed abstraction for memory_read/memory_write
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% proof status :-)
% changed_memory_invariant?_TCC1 : TCC Obligation

changed_memory_invariant?(pm, states, addresses) : bool =
  transformer_invariant?(states, memory_read_transformers(pm, addresses))
AND
  transformer_invariant?(states, memory_write_transformers(pm, addresses))
AND
Forall(s : State, a : Address, b : Byte) :
  states(s) AND addresses(a) AND
  OK?(memory_write(pm)(a,b)(s)) AND
  OK?(memory_read(pm)(a)(state(memory_write(pm)(a,b)(s))))

```

IMPLIES

data(memory_read(pm)(a)(state(memory_write(pm)(a,b)(s)))) = b

% proof status :-)

changed_memory_invariant_mono : **Lemma**

Forall(addresses_1, addresses_2 : PRED[Address]) :

subset?(addresses_1, addresses_2) **AND**

changed_memory_invariant?(pm, states, addresses_2)

IMPLIES

changed_memory_invariant?(pm, states, addresses_1)

%%
%

% unchanged results : memory_read_list

%

%%

% proof status :-)

memory_read_list_nse_ok : **Lemma**

Forall(raddr : Address, size : nat, s : State) :

transformer_invariant?(states,
memory_read_transformers(pm, address_block(raddr, size)))

And

transformers_ok?(states,
memory_read_transformers(pm, address_block(raddr, size)))

And

states(s)

Implies

OK?(memory_read_list_nse(pm)(raddr, size)(s))

% proof status :-)

% memory_read_list_nse_next_length_TCC1 : TCC Obligation

% proof status :-)

% memory_read_list_nse_next_length_TCC2 : TCC Obligation

% proof status :-)

memory_read_list_nse_next_length : **Lemma**

Forall(raddr : Address, size : nat, s : State) :

transformer_invariant?(states,
memory_read_transformers(pm, address_block(raddr, size)))

AND

transformers_ok?(states,
memory_read_transformers(pm, address_block(raddr, size))) **AND**

states(s)

Implies

states(state(memory_read_list_nse(pm)(raddr, size)(s)))

And

length(data(memory_read_list_nse(pm)(raddr, size)(s))) = size

```

% if read_memory is ok then read_list is ok
% proof status :-)
memory_read_list_ok : Lemma
Forall(raddr : Address, size : nat, s : State) :
  transformer_invariant?(states,
    union(
      memory_read_transformers(pm, address_block(raddr, size)),
      memory_read_side_effect_super_transformers(pm,
        address_block(raddr, size))))))

And
transformers_ok?(states,
  union(
    memory_read_transformers(pm, address_block(raddr, size)),
    memory_read_side_effect_super_transformers(pm,
      address_block(raddr, size))))))

And
states(s)
Implies
  OK?(memory_read_list(pm)(raddr, size)(s))

% proof status :-)
% memory_read_list_next_ok_TCC1 : TCC Obligation

% if read_memory is ok then next state of read_list is in states
% proof status :-)
memory_read_list_next_ok : Lemma
Forall(raddr : Address, size : nat, s : State) :
  transformer_invariant?(states,
    union(
      memory_read_transformers(pm, address_block(raddr, size)),
      memory_read_side_effect_super_transformers(pm,
        address_block(raddr, size))))))

AND
transformers_ok?(states,
  union(
    memory_read_transformers(pm, address_block(raddr, size)),
    memory_read_side_effect_super_transformers(pm,
      address_block(raddr, size))))))

And
states(s)
Implies
  states(state(memory_read_list(pm)(raddr, size)(s)))

%%%%%%%%%%
%
% memory_read_list ## q
%
%%%%%%%%%%

% proof status :-)
% unchanged_memory_read_list_nse_TCC1 : TCC Obligation

```

B PVS Theory Sources

```

% proof status :-)
% unchanged_memory_read_list_nse_TCC2 : TCC Obligation

% proof status :-)
% unchanged_memory_read_list_nse_TCC3 : TCC Obligation

% The obvious unchanged lemma for memory_read_list
% proof status :-)
unchanged_memory_read_list_nse : Lemma
  Forall(raddr : Address, size : nat, s : State,
          q : [State -> SuperResult[State]]) :
    unchanged_memory_invariant?(pm, states,
      union(memory_read_transformers(pm, address_block(raddr, size)),
        singleton(q)),
      addresses)
  AND
    transformers_ok?(states,
      memory_read_transformers(pm, address_block(raddr, size)))
  AND
    subset?(address_block(raddr, size), addresses) AND
    states(s) AND
    OK?(q(s))
  IMPLIES
    data(memory_read_list_nse(pm)(raddr, size)(state(q(s)))) =
      data(memory_read_list_nse(pm)(raddr, size)(s))

% proof status :-)
% unchanged_memory_read_list_TCC1 : TCC Obligation

% proof status :-)
% unchanged_memory_read_list_TCC2 : TCC Obligation

% proof status :-)
% unchanged_memory_read_list_TCC3 : TCC Obligation

% proof status :-)
unchanged_memory_read_list : Lemma
  Forall(raddr : Address, size : nat, s : State,
          q : [State -> SuperResult[State]]) :
    unchanged_memory_invariant?(pm, states,
      union(
        union(memory_read_transformers(pm, address_block(raddr, size)),
          memory_read_side_effect_super_transformers(pm,
            address_block(raddr, size))),
        singleton(q)),
      addresses)
  AND
    transformers_ok?(states,
      union(
        memory_read_transformers(pm, address_block(raddr, size)),
        memory_read_side_effect_super_transformers(pm,

```

```

                                address_block(raddr, size)))
AND
side_effect_content_unchanged(address_block(raddr, size), states,
                                memory_read_side_effect(pm))

And
subset?(address_block(raddr, size), addresses) AND
states(s) AND
OK?(q(s))
IMPLIES
  data(memory_read_list(pm)(raddr, size)(state(q(s)))) =
  data(memory_read_list(pm)(raddr, size)(s))

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% memory_read ## memory_read_list
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% proof status :-)
% unchanged_memory_invariant_unchanged_read_list_nse_TCC1 : TCC Obligation

% proof status :-)
% unchanged_memory_invariant_unchanged_read_list_nse_TCC2 : TCC Obligation

% proof status :-)
% unchanged_memory_invariant_unchanged_read_list_nse_TCC3 : TCC Obligation

% proof status :-)
unchanged_memory_invariant_unchanged_read_list_nse : Lemma
Forall(s : State, addr_1, addr_2 : Address, size : nat) :
  unchanged_memory_invariant?(pm, states,
    memory_read_transformers(pm, address_block(addr_2, size)),
    addresses)
AND
transformers_ok?(states, memory_read_transformers(pm, addresses))
AND
subset?(address_block(addr_2, size), addresses)
And
addresses(addr_1)
And
states(s)
IMPLIES
  data(memory_read(pm)(addr_1)(
    state(memory_read_list_nse(pm)(addr_2, size)(s)))) =
  data(memory_read(pm)(addr_1)(s))

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% normal termination of memory_write_list
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

```

```

% if write_memory is ok then write_list is ok
% proof status :-)
memory_write_list_ok_nse : Lemma
  Forall(waddr : Address, s : State, bl : list[Byte]) :
    transformer_invariant?(states,
      memory_write_transformers(pm, address_block(waddr, length(bl))))
  AND
    transformers_ok?(states,
      memory_write_transformers(pm, address_block(waddr, length(bl))))
  AND
    states(s)
  IMPLIES
    OK?(memory_write_list_nse(pm)(waddr, bl)(s))

% proof status :-)
memory_write_list_ok : Lemma
  Forall(waddr : Address, s : State, bl : list[Byte]) :
    transformer_invariant?(states,
      union(memory_write_transformers(pm, address_block(waddr, length(bl))),
        memory_write_side_effect_super_transformers(pm,
          address_block(waddr, length(bl))))))
  AND
    transformers_ok?(states,
      union(memory_write_transformers(pm, address_block(waddr, length(bl))),
        memory_write_side_effect_super_transformers(pm,
          address_block(waddr, length(bl))))))
  AND
    side_effect_content_unchanged(address_block(waddr, length(bl)), states,
      memory_write_side_effect(pm))
  And
    states(s)
  IMPLIES
    OK?(memory_write_list(pm)(waddr, bl)(s))

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% unchanged results : memory_read_list ## memory_write_list
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% proof status :-)
% unchanged_memory_invariant_unchanged_write_list_nse_TCC1 : TCC Obligation

% proof status :-)
% unchanged_memory_invariant_unchanged_write_list_nse_TCC2 : TCC Obligation

% proof status :-)
% unchanged_memory_invariant_unchanged_write_list_nse_TCC3 : TCC Obligation

```

```

% proof status :-)
unchanged_memory_invariant_unchanged_write_list_nse : Lemma
Forall(s : State, raddr, waddr : Address, bl : list[Byte]) :
  unchanged_memory_invariant?(pm, states,
    memory_write_transformers(pm, address_block(waddr, length(bl))),
    singleton(raddr))
AND
transformers_ok?(states,
  memory_write_transformers(pm, address_block(waddr, length(bl))))
AND
transformers_ok?(states, singleton(expr_2_super(memory_read(pm)(raddr))))
AND
states(s)
IMPLIES
data(memory_read(pm)(raddr)(
  state(memory_write_list_nse(pm)(waddr, bl)(s)))) =
data(memory_read(pm)(raddr)(s))

% proof status :-)
% unchanged_memory_invariant_unchanged_write_list_TCC1 : TCC Obligation

% proof status :-)
% unchanged_memory_invariant_unchanged_write_list_TCC2 : TCC Obligation

% proof status :-)
% unchanged_memory_invariant_unchanged_write_list_TCC3 : TCC Obligation

% proof status :-)
unchanged_memory_invariant_unchanged_write_list : Lemma
Forall(s : State, raddr, waddr : Address, bl : list[Byte]) :
  unchanged_memory_invariant?(pm, states,
    union(memory_write_transformers(pm, address_block(waddr, length(bl))),
      memory_write_side_effect_super_transformers(pm,
        address_block(waddr, length(bl))))),
    singleton(raddr))
AND
transformers_ok?(states,
  union(memory_write_transformers(pm, address_block(waddr, length(bl))),
    memory_write_side_effect_super_transformers(pm,
      address_block(waddr, length(bl))))))
AND
transformers_ok?(states, singleton(expr_2_super(memory_read(pm)(raddr))))
AND
side_effect_content_unchanged(address_block(waddr, length(bl)), states,
  memory_write_side_effect(pm))
AND
states(s)
IMPLIES
data(memory_read(pm)(raddr)(
  state(memory_write_list(pm)(waddr, bl)(s)))) =

```



```

    data(memory_read(pm)(raddr)(s))

% unchanged memory is an invariant of read_list
% proof status :-)
unchanged_memory_invariant_read_list : Lemma
Forall(addr : Address, size : nat) :
  unchanged_memory_invariant?(pm, states, union(
    memory_read_transformers(pm, address_block(addr, size)),
    memory_read_side_effect_super_transformers(pm,
      address_block(addr, size))),
    addresses)
AND
transformers_ok?(states,
  union(
    memory_read_transformers(pm, addresses),
    memory_read_side_effect_super_transformers(pm, addresses)))
And
subset?(address_block(addr, size), addresses)
IMPLIES
  unchanged_memory_invariant?(pm, states,
    singleton(expr_2_super(memory_read_list(pm)(addr, size))),
    addresses)

% proof status :-)
% unchanged_memory_read_list_write_list_nse_TCC1 : TCC Obligation

% proof status :-)
% unchanged_memory_read_list_write_list_nse_TCC2 : TCC Obligation

% proof status :-)
unchanged_memory_read_list_write_list_nse : Lemma
Forall(waddr : Address, bl : list[Byte], s : State) :
  unchanged_memory_invariant?(pm, states,
    memory_read_transformers(pm, address_block(waddr, length(bl))),
    address_block(waddr, length(bl)))
AND
  unchanged_memory_write_invariant?(pm, states,
    address_block(waddr, length(bl)))
AND
  changed_memory_invariant?(pm, states, address_block(waddr, length(bl)))
AND
  transformers_ok?(states,
    memory_read_transformers(pm, address_block(waddr, length(bl))))
AND
  transformers_ok?(states,
    memory_write_transformers(pm, address_block(waddr, length(bl))))
AND
  states(s)
IMPLIES
  data(memory_read_list_nse(pm)(waddr, length(bl))(
    state(memory_write_list_nse(pm)(waddr, bl)(s)))) = bl

```

```

% proof status :-)
% unchanged_memory_read_list_write_list_TCC1 : TCC Obligation

% proof status :-)
% unchanged_memory_read_list_write_list_TCC2 : TCC Obligation

% proof status :-)
unchanged_memory_read_list_write_list : Lemma
Forall(waddr : Address, bl : list[Byte], s : State) :
  unchanged_memory_invariant?(pm, states, union(
    memory_read_transformers(pm, address_block(waddr, length(bl))),
    union(
      memory_read_side_effect_super_transformers(pm,
        address_block(waddr, length(bl))),
      memory_write_side_effect_super_transformers(pm,
        address_block(waddr, length(bl))))),
    address_block(waddr, length(bl)))
AND
  unchanged_memory_write_invariant?(pm, states,
    address_block(waddr, length(bl)))
AND
  changed_memory_invariant?(pm, states, address_block(waddr, length(bl)))
AND
  transformers_ok?(states,
    union(
      memory_read_transformers(pm, address_block(waddr, length(bl))),
      memory_read_side_effect_super_transformers(pm,
        address_block(waddr, length(bl))))))
AND
  transformers_ok?(states,
    union(
      memory_write_transformers(pm, address_block(waddr, length(bl))),
      memory_write_side_effect_super_transformers(pm,
        address_block(waddr, length(bl))))))
AND
  side_effect_content_unchanged(address_block(waddr, length(bl)), states,
    memory_read_side_effect(pm))
AND
  side_effect_content_unchanged(address_block(waddr, length(bl)), states,
    memory_write_side_effect(pm))
AND
  states(s)
IMPLIES
  data(memory_read_list(pm)(waddr, length(bl)))(
    state(memory_write_list(pm)(waddr, bl)(s))) = bl

```

End Memory_Change_2

B.19 paging-data-models.pvs

B PVS Theory Sources

PDBR_Data_Model : **Theory**

Begin

Importing Bits, PDBR_Type, Interpreted_Data[Pdbr_type]

```
% 31 12 5 4 3 2 0
% +-----+
% || |P|P| |
% | page dir base | empty |C|W| emp |
% || |D|T| |
% +-----+
```

```
% proof status :-)
% pdbr_valid?_TCC1 : TCC Obligation
```

```
% proof status :-)
% pdbr_valid?_TCC2 : TCC Obligation
```

```
pdbr_valid?(data : list[Byte], a : Address) : bool =
  length(data) = 4 And
  % upper 20 bits are the base
  % next 7 bits are empty
  cut_bits(nth(data, 1), 0, 4) = 0 And
  cut_bits(nth(data, 0), 5, 3) = 0 And
  % bit 4 is PCD
  % bit 3 is PWT
  % remainder is empty
  cut_bits(nth(data, 0), 0, 3) = 0
```

```
% proof status :-)
% pdbr_to_byte_TCC1 : TCC Obligation
```

```
% proof status :-)
% pdbr_to_byte_TCC2 : TCC Obligation
```

```
% proof status :-)
% pdbr_to_byte_TCC3 : TCC Obligation
```

```
pdbr_to_byte(pdbr : Pdbr_type, a : Address) : list[Byte] =
  (: overwrite_bool_bit(
    overwrite_bool_bit(0, pdbr'pwt, 3),
    pdbr'pcd, 4),
    cut_bits(offset(pdbr'base_addr), 8, 8),
    cut_bits(offset(pdbr'base_addr), 16, 8),
    cut_bits(offset(pdbr'base_addr), 24, 8)
  :)
```

```
% proof status :-)
% pdbr_from_byte_TCC1 : TCC Obligation
```

```
% proof status :-)
% pdbr_from_byte_TCC2 : TCC Obligation
```

```

% proof status :-)
% pabr_from_byte_TCC3 : TCC Obligation

% proof status :-)
% pabr_from_byte_TCC4 : TCC Obligation

% proof status :-)
% pabr_from_byte_TCC5 : TCC Obligation

pabr_from_byte(data : list[Byte], a : Address) : lift[Pabr_type] =
  IF pabr_valid?(data, a) THEN
    up((# base_addr :=
      Mem(
        overwrite_shift_bits(
          overwrite_shift_bits(
            overwrite_shift_bits(0, nth(data, 1), 8, 8),
            nth(data, 2), 16, 8),
            nth(data, 3), 24, 8)
          ),
        pcd := cut_bit(nth(data, 0), 4),
        pwt := cut_bit(nth(data, 0), 3)
      #))
  ELSE bottom
ENDIF

% proof status :-)
% pabr_model_TCC1 : TCC Obligation

pabr_model : Interpreted_data_type[Pabr_type] = (#
  uidt := (#
    size := 4,
    valid? := pabr_valid?
  #),
  to_byte := pabr_to_byte,
  to_mask := Lambda (d : Pabr_type, a : Address) :
    (zero_mask?(4)),
  from_byte := pabr_from_byte
#)

% proof status :-)
pabr_valid_to_byte : Lemma Forall(d: Pabr_type, a: Address):
  valid?(uidt(pabr_model))(to_byte(pabr_model)(d, a), a)

% proof status :-)
pabr_is_pod : Theorem pod_data_type?(pabr_model)

End PABR_Data_Model

```

B PVS Theory Sources

PDE_Model : **Theory**

Begin

```
% PDE 4K
%
% 32 12 11 9 8 7 6 5 4 3 2 1 0
% +-----+
% ||||| R || P | P | U | R ||
% | page table address | avail | G | P | E | A | C | W | / | / | P |
% ||||| S | S || D | T | S | W ||
% +-----+
%
% PDE 4M
%
% 32 22 21 13 12 11 9 8 7 6 5 4 3 2 1 0
% +-----+
% ||| P ||||| P | P | U | R ||
% | page address | reserved | A | avail | G | P | D | A | C | W | / | / | P |
% ||| T ||| S ||| D | T | S | W ||
% +-----+
%
% G : global page (ignored for 4K PTE's)
% PS : page size (1 for superpages)
% RES : reserved (0)
% A : accessed
% PCD : cache disabled
% PWT : cache write through
% U/S : user / supervisor
% R/W : read / write
% P : present
% PAT : page table attribute index
% D : dirty
```

Importing Pde_type, Memory_privilege_util, Interpreted_Data

```
% proof status :-)
% pde_valid?_TCC1 : TCC Obligation
```

```
% proof status :-)
% pde_valid?_TCC2 : TCC Obligation
```

```
% proof status :-)
% pde_valid?_TCC3 : TCC Obligation
```

```
pde_valid?(data : list[Byte], a : Address) : bool =
  length(data) = 4 And
  IF cut_bit(nth(data, 0), 0)
  Then % present
  IF cut_bit(nth(data, 0), 7)
  Then % superpage entry
  cut_bits(nth(data, 1), 5, 3) = 0 And
  cut_bits(nth(data, 2), 0, 6) = 0
```

```

    Else % PTE entry
      cut_bits(nth(data, 0), 6, 1) = 0
    Endif
Else % not present
  True
Endif

% proof status :-)
% pde_pte_to_byte_TCC1 : TCC Obligation

% proof status :-)
% pde_pte_to_byte_TCC2 : TCC Obligation

% proof status :-)
% pde_pte_to_byte_TCC3 : TCC Obligation

pde_pte_to_byte(pde : Pde_pt_type, a : Address) : list[Byte] =
  (:
    overwrite_bool_bit(
      overwrite_bool_bit(
        overwrite_bool_bit(
          overwrite_bool_bit(
            overwrite_bool_bit(
              overwrite_bool_bit(0, true, 0),
              memory_access_to_bools(pde'access)'1, 1),
              memory_privilege_to_bool(pde'privilege), 2),
              pde'write_through, 3),
              pde'cache_disabled, 4),
              pde'accessed, 5),

    overwrite_shift_bits(
      overwrite_bool_bit(0, pde'global_page, 0),
      cut_bits(offset(pde'page_table_base), 12, 4),
      4, 4),

    cut_bits(offset(pde'page_table_base), 16, 8),
    cut_bits(offset(pde'page_table_base), 24, 8)
  :)

% proof status :-)
% pde_super_to_byte_TCC1 : TCC Obligation

% proof status :-)
% pde_super_to_byte_TCC2 : TCC Obligation

% proof status :-)
% pde_super_to_byte_TCC3 : TCC Obligation

pde_super_to_byte(spge : Pde_4m_type, a : Address) : list[Byte] =
  (:
    overwrite_bool_bit(

```

B PVS Theory Sources

```
    overwrite_bool_bit(
      overwrite_bool_bit(
        overwrite_bool_bit(
          overwrite_bool_bit(
            overwrite_bool_bit(
              overwrite_bool_bit(
                overwrite_bool_bit(0, true, 0),
                memory_access_to_bools(spage'access)'1, 1),
                memory_privilege_to_bool(spage'privilege), 2),
                spage'write_through, 3),
                spage'cache_disabled, 4),
                spage'accessed, 5),
                spage'dirty, 6),
        true, 7),

    overwrite_bool_bit(
      overwrite_bool_bit(0, spage'global_page, 0),
      spage'page_table_attribute_index, 4),

    shift_bits_left(cut_bits(offset(spage'page_base), 22, 2), 6),
    cut_bits(offset(spage'page_base), 24, 8)
  :)

% proof status :- )
% pde_to_byte_TCC1 : TCC Obligation

% proof status :- )
% pde_to_byte_TCC2 : TCC Obligation

pde_to_byte(pde : Pde_type, a : Address) : list[Byte] =
  Cases pde of
    Not_present : (: 0, 0, 0, 0 :),
    Pde_pt(pte) : pde_pte_to_byte(pte, a),
    Pde_4m(spage) : pde_super_to_byte(spage, a)
  EndCases

% proof status :- )
% pde_4m_from_byte_TCC1 : TCC Obligation

% proof status :- )
% pde_4m_from_byte_TCC2 : TCC Obligation

% proof status :- )
% pde_4m_from_byte_TCC3 : TCC Obligation

% proof status :- )
% pde_4m_from_byte_TCC4 : TCC Obligation

% proof status :- )
% pde_4m_from_byte_TCC5 : TCC Obligation
```

```

% proof status :-)
% pde_4m_from_byte_TCC6 : TCC Obligation

pde_4m_from_byte(data : list[Byte], a : Address) : lift[Pde_4m_type] =
  % The first if here seems redundant. However, we do
  % down(x_pod'from_byte...) which only works on valid data. The
  % (worse) alternative is to restrict the input to valid data-address
  % pairs.
IF pde_valid?(data, a) And
  cut_bit(nth(data, 0), 0) And
  cut_bit(nth(data, 0), 7)
Then
  up((#
    page_base :=
      Mem(
        overwrite_shift_bits(
          overwrite_shift_bits(0, cut_bits(nth(data, 2), 6, 2), 22, 2),
          nth(data, 3), 24, 8)
        ),
    page_table_attribute_index := cut_bit(nth(data, 1), 4),
    global_page := cut_bit(nth(data, 1), 0),
    dirty := cut_bit(nth(data, 0), 6),
    accessed := cut_bit(nth(data, 0), 5),
    cache_disabled := cut_bit(nth(data, 0), 4),
    write_through := cut_bit(nth(data, 0), 3),
    privilege := bool_to_memory_privilege(cut_bit(nth(data, 0), 2)),
    access := bools_to_memory_access(cut_bit(nth(data, 0), 1), true)
  #))
Else bottom
Endif

% proof status :-)
% pde_4k_from_byte_TCC1 : TCC Obligation

% proof status :-)
% pde_4k_from_byte_TCC2 : TCC Obligation

% proof status :-)
% pde_4k_from_byte_TCC3 : TCC Obligation

% proof status :-)
% pde_4k_from_byte_TCC4 : TCC Obligation

% proof status :-)
% pde_4k_from_byte_TCC5 : TCC Obligation

pde_4k_from_byte(data : list[Byte], a : Address) : lift[Pde_pt_type] =
  % Again the first if seems redundant ...
IF pde_valid?(data, a) And
  cut_bit(nth(data, 0), 0) And
  Not cut_bit(nth(data, 0), 7)
Then

```



```

up((#
  page_table_base :=
    Mem(
      overwrite_shift_bits(
        overwrite_shift_bits(
          overwrite_shift_bits(0, cut_bits(nth(data, 1), 4, 4),
            12, 4),
          nth(data, 2), 16, 8),
          nth(data, 3), 24, 8)
      ),
      global_page := cut_bit(nth(data, 1), 0),
      accessed := cut_bit(nth(data, 0), 5),
      cache_disabled := cut_bit(nth(data, 0), 4),
      write_through := cut_bit(nth(data, 0), 3),
      privilege := bool_to_memory_privilege(cut_bit(nth(data, 0), 2)),
      access := bools_to_memory_access(cut_bit(nth(data, 0), 1), true)
    #))
Else bottom
Endif

% proof status :-)
% pde_from_byte_TCC1 : TCC Obligation

% proof status :-)
% pde_from_byte_TCC2 : TCC Obligation

pde_from_byte(data : list[Byte], a : Address) : lift[Pde_type] =
IF pde_valid?(data, a) Then
  IF cut_bit(nth(data, 0), 0)
    Then % present
      IF cut_bit(nth(data, 0), 7)
        Then % super page entry
          up(Pde_4m(down(pde_4m_from_byte(data, a))))
        Else % page table entry
          up(Pde_pt(down(pde_4k_from_byte(data, a))))
        Endif
      Else % not present
        up(Not_present)
      Endif
    Else bottom
  Endif

% proof status :-)
% pde_model_TCC1 : TCC Obligation

pde_model : Interpreted_data_type[Pde_type] = (#
  uidt := (#
    size := 4,
    valid? := pde_valid?
  #),
  to_byte := pde_to_byte,
  to_mask := Lambda (d : Pde_type, a : Address) :

```

```

        (zero_mask?(4)),
    from_byte := pde_from_byte
#)

% proof status :-)
pde_valid_to_byte : Lemma Forall(d: Pde_type, a: Address):
    valid?(uidt(pde_model))(to_byte(pde_model)(d, a), a)

% proof status :-)
pde_is_pod : Theorem pod_data_type?(pde_model)

```

End PDE_Model

PTE_Model : Theory

Begin

```

% PTE
%
% 32 12 11 9 8 7 6 5 4 3 2 1 0
% +-----+-----+-----+-----+-----+-----+-----+-----+
% |||| P ||| P | P | U | R ||
% | page address | avail | G | A | D | A | C | W | / | / | P |
% |||| T ||| D | T | S | W ||
% +-----+-----+-----+-----+-----+-----+-----+-----+
% G : global page
% PAT : page table attribute index
% D : dirty
% A : accessed
% PCD : cache disabled
% PWT : cache write through
% U/S : user / supervisor
% R/W : read / write
% P : present

```

Importing Pte_type, Memory_privilege_util, Interpreted_Data

```

pte_valid?(data : list[Byte], a : Address) : bool =
    length(data) = 4

% proof status :-)
% pte_page_to_byte_TCC1 : TCC Obligation

% proof status :-)
% pte_page_to_byte_TCC2 : TCC Obligation

% proof status :-)
% pte_page_to_byte_TCC3 : TCC Obligation

pte_page_to_byte(pte : Pte_4k_type, a : Address) : list[Byte] =

```



```

% proof status :-)
% pte_page_from_byte_TCC5 : TCC Obligation

% proof status :-)
% pte_page_from_byte_TCC6 : TCC Obligation

pte_page_from_byte(data : list[Byte], a : Address) : lift[Pte_4k_type] =
  % The first if seems redundant ... see earlier comments on this
  IF pte_valid?(data, a) And cut_bit(nth(data, 0), 0)
  Then
    up((#
      page_base :=
        Mem(
          overwrite_shift_bits(
            overwrite_shift_bits(
              overwrite_shift_bits(0, cut_bits(nth(data, 1), 4, 4),
                12, 4),
              nth(data, 2), 16, 8),
              nth(data, 3), 24, 8)
            ),
          global_page := cut_bit(nth(data, 1), 0),
          page_table_attribute_index := cut_bit(nth(data, 0), 7),
          dirty := cut_bit(nth(data, 0), 6),
          accessed := cut_bit(nth(data, 0), 5),
          cache_disabled := cut_bit(nth(data, 0), 4),
          write_through := cut_bit(nth(data, 0), 3),
          privilege := bool_to_memory_privilege(cut_bit(nth(data, 0), 2)),
          access := bools_to_memory_access(cut_bit(nth(data, 0), 1), true)
        #))
  Else bottom
  Endif

% proof status :-)
% pte_from_byte_TCC1 : TCC Obligation

pte_from_byte(data : list[Byte], a : Address) : lift[Pte_type] =
  IF pte_valid?(data, a) Then
    IF cut_bit(nth(data, 0), 0)
    Then % present
      up(Pte(down(pte_page_from_byte(data, a))))
    Else % not present
      up(Not_present)
    Endif
  Else bottom
  Endif

% proof status :-)
% pte_model_TCC1 : TCC Obligation

pte_model : Interpreted_data_type[Pte_type] = (#
  uidt := (#

```

B PVS Theory Sources

```

    size := 4,
    valid? := pte_valid?
#),
to_byte := pte_to_byte,
to_mask := Lambda (d : Pte_type, a : Address) :
    (zero_mask?(4)),
from_byte := pte_from_byte
#)

% proof status :-)
pte_valid_to_byte : Lemma Forall(d : Pte_type, a : Address) :
    pte_model'uidt'valid?(pte_model'to_byte(d, a), a)

% proof status :-)
pte_is_pod : Theorem pod_data_type?(pte_model)

```

End PTE_Model

PF_EC_Model : **Theory**

Begin

Importing PF_Error_Code_Type, Memory_privilege_util

```

% Page Fault Error Code
% 31 5 4 3 2 1 0
% +-----+
% | I | R | U | W | |
% | Reserved | / | S | / | / | P |
% | | D | V | S | R | |
% +-----+

% proof status :-)
% pf_ec_valid?_TCC1 : TCC Obligation

% proof status :-)
% pf_ec_valid?_TCC2 : TCC Obligation

% proof status :-)
% pf_ec_valid?_TCC3 : TCC Obligation

% proof status :-)
% pf_ec_valid?_TCC4 : TCC Obligation

pf_ec_valid?(data : list[Byte], a : Address) : bool =
    length(data) = 4 And
    % upper 27 bits are reserved
    cut_bits(nth(data, 3), 0, 8) = 0 And
    cut_bits(nth(data, 2), 0, 8) = 0 And
    cut_bits(nth(data, 1), 0, 8) = 0 And
    cut_bits(nth(data, 0), 5, 3) = 0

```

```

% bit 4 is instruction / data
% bit 3 is reserved bit
% bit 2 is user supervisor
% bit 1 is write / read
% bit 0 is present

% proof status :-)
% pf_ec_to_byte_TCC1 : TCC Obligation

% proof status :-)
% pf_ec_to_byte_TCC2 : TCC Obligation

pf_ec_to_byte(pf_ec : PF_error_code_type, a : Address) : list[Byte] =
  (: overwrite_bool_bit(
    overwrite_bool_bit(
      overwrite_bool_bit(
        overwrite_bool_bit(
          overwrite_bool_bit(0, pf_ec'present, 0),
          member(Write, pf_ec'access), 1),
          memory_privilege_to_bool(pf_ec'privilege), 2),
          pf_ec'reserved_bit_violation, 3),
          member(Execute, pf_ec'access), 4),
    0,
    0,
    0
  :))

% proof status :-)
% pf_ec_from_byte_TCC1 : TCC Obligation

% proof status :-)
% pf_ec_from_byte_TCC2 : TCC Obligation

pf_ec_from_byte(data : list[Byte], a : Address) : lift[PF_error_code_type] =
  If pf_ec_valid?(data, a) Then
    up((# reserved_bit_violation := cut_bit(nth(data, 0), 3),
      privilege := bool_to_memory_privilege(cut_bit(nth(data, 0), 2)),
      access := booleans_to_memory_access(cut_bit(nth(data, 0), 1), cut_bit(nth(data, 0), 4)),
      present := cut_bit(nth(data, 0), 0)
    #))
  Else
    bottom
  Endif

% proof status :-)
% pf_ec_model_TCC1 : TCC Obligation

pf_ec_model : Interpreted_data_type[PF_error_code_type] = (#
  uidt := (#
    size := 4,
    valid? := pf_ec_valid?
  #),

```

B PVS Theory Sources

```
to_byte := pf_ec_to_byte,
to_mask := Lambda (d : PF_error_code_type, a : Address) :
    (zero_mask?(4)),
from_byte := pf_ec_from_byte
#)

% proof status :-)
pf_ec_valid_to_byte : Lemma Forall(d: PF_error_code_type, a: Address):
    valid?(uidt(pf_ec_model))(to_byte(pf_ec_model)(d, a), a)

% proof status :-)
pf_ec_is_pod : Theorem pod_data_type?(pf_ec_model)
```

End PF_EC_Model

Address_Model : **Theory**

Begin

Importing Address_Datatype

```
address_valid?(data : list[Byte], a : Address) : bool =
    length(data) = 4

% proof status :-)
% address_to_byte_TCC1 : TCC Obligation

% proof status :-)
% address_to_byte_TCC2 : TCC Obligation

% proof status :-)
% address_to_byte_TCC3 : TCC Obligation

address_to_byte(addr : Memory_Address_4G, a : Address) : list[Byte] =
    (: cut_bits(offset(addr), 0, 8),
    cut_bits(offset(addr), 8, 8),
    cut_bits(offset(addr), 16, 8),
    cut_bits(offset(addr), 24, 8)
    :)

% proof status :-)
% address_from_byte_TCC1 : TCC Obligation

% proof status :-)
% address_from_byte_TCC2 : TCC Obligation

% proof status :-)
% address_from_byte_TCC3 : TCC Obligation

% proof status :-)
% address_from_byte_TCC4 : TCC Obligation
```

```

% proof status :-)
% address_from_byte_TCC5 : TCC Obligation

address_from_byte(data : list[Byte], a : Address) : lift[Memory_Address_4G] =
  If address_valid?(data, a) Then
    up(Mem(overwrite_shift_bits(
      overwrite_shift_bits(
        overwrite_shift_bits(
          overwrite_shift_bits(0,
            nth(data, 0), 0, 8),
            nth(data, 1), 8, 8),
            nth(data, 2), 16, 8),
            nth(data, 3), 24, 8)))
  Else
    bottom
  Endif

% proof status :-)
% address_model_TCC1 : TCC Obligation

address_model : Interpreted_data_type[Memory_Address_4G] = (#
  uidt := (#
    size := 4,
    valid? := address_valid?
  #),
  to_byte := address_to_byte,
  to_mask := Lambda (d : Memory_Address_4G, a : Address) :
    (zero_mask?(4)),
  from_byte := address_from_byte
#)

% proof status :-)
address_valid_to_byte : Lemma Forall(d : Memory_Address_4G, a : Address):
  valid?(uidt(address_model))(to_byte(address_model))(d, a), a

% proof status :-)
address_is_pod : Theorem pod_data_type?(address_model)

```

End Address_Model

B.20 paging-data.pvs

PDBR_Type : **Theory**

Begin

Importing IA32, Alignment

% proof status :-)

% Pdbr_type_TCC1 : TCC Obligation

B PVS Theory Sources

```
Pdbr_type : Type = [#  
  base_addr : {a : Memory_Address_4G | aligned?(12)(offset(a))},  
  pcd : bool,  
  pwt : bool  
#]
```

End PDBR_Type

PDBR_Datatype : **Theory**

% axioms are discharged in PDBR_Data_Model

Begin

Importing PDBR_Type, Interpreted_Data[Pdbr_type]

pdbr_data_type_exists : **Axiom EXISTS**(x: (pod_data_type?[Pdbr_type])): **TRUE**

% proof status :-)

% pdbr_data_type_TCC1 : TCC Obligation

pdbr_data_type : (pod_data_type?[Pdbr_type])

End PDBR_Datatype

% Page_Directory_Types[T : Type] : Theory

% type dependency waits until available bits are used

Page_Directory_Types : **Theory**

Begin

Importing IA32, Memory_privilege, Memory_access_util, Alignment

% proof status :-)

% Pde_pt_type_TCC1 : TCC Obligation

```
Pde_pt_type : Type = [#  
  page_table_base : {a : Memory_Address_4G | aligned?(12)(offset(a))},  
  % pde_4k_avail : T,  
  global_page : bool,  
  accessed : bool,  
  cache_disabled : bool,  
  write_through : bool,  
  privilege : Memory_privilege,  
  access : RX_Memory_access  
#]
```

```
Pde_4m_type : Type = [#  
  page_base : {a : Memory_Address_4G | aligned?(22)(offset(a))},  
  page_table_attribute_index : bool,  
  % pde_4m_avail : T,  
  global_page : bool,  
  dirty : bool,  
  accessed : bool,  
#]
```

```

cache_disabled : bool,
write_through : bool,
privilege : Memory_privilege,
access : RX_Memory_access
#]

Pte_4k_type : Type = [#
  page_base : {a : Memory_Address_4G | aligned?(12)(offset(a))},
  % pte_avail : T,
  global_page : bool,
  page_table_attribute_index : bool,
  dirty : bool,
  accessed : bool,
  cache_disabled : bool,
  write_through : bool,
  privilege : Memory_privilege,
  access : RX_Memory_access
#]

```

End Page_Directory_Types

```

%Pde_type[X, Y : Type] : Datatype
% type dependency waits until available bits are used
Pde_type : Datatype
Begin
  Importing Page_Directory_Types

  % Not_present(data : Y) : not_present?
  Not_present : not_present?
  Pde_pt(pde_pt : Pde_pt_type) : pde_pt?
  Pde_4m(pde_4m : Pde_4m_type) : pde_4m?
End Pde_type

```

```

PDE_Datatype : Theory
Begin
  Importing Pde_type, Interpreted_Data

  pde_data_type_exists : Axiom
    Exists(x : (pod_data_type?[Pde_type])) : True

  % proof status :-)
  % pde_data_type_TCC1 : TCC Obligation

  pde_data_type : (pod_data_type?[Pde_type])

End PDE_Datatype

```

```

Pte_type : Datatype
Begin

```

B PVS Theory Sources

Importing Page_Directory_Types

Not_present : not_present?
Pte(pte : Pte_4k_type) : pte?

End Pte_type

PTE_Datatype : **Theory**

Begin

Importing Pte_type, Interpreted_Data

pte_data_type_exists : **Axiom**
Exists(x : (pod_data_type?[Pte_type])) : **True**

% proof status :-)
% pte_data_type_TCC1 : TCC Obligation

pte_data_type : (pod_data_type?[Pte_type])

End PTE_Datatype

%%
% Use this type only in linear resolve and only after verifying the type
%

Paging_type : **Datatype**

Begin

Importing Pde_type, Pte_type

Ptab(ptab : Pte_type) : ptab?
Pdir(pdir : Pde_type) : pdir?

End Paging_type

Paging_type_helpers : **Theory**

Begin

Importing Paging_type

pt : **Var** Paging_type

max_level : nat = 2
Level : **Type** = below[max_level]

% proof status :-)
% range_pt_TCC1 : TCC Obligation

range_pt(lvl : Level) : PRED[Paging_type] =

Lambda (p : Paging_type) :

Cond

lvl = 0 -> pdir?(p),

lvl = 1 -> ptab?(p)

EndCond

```

set_accessed(pt) : Paging_type =
Cases pt Of
  Ptab(ptab) :
    Cases ptab Of
      Not_present : Ptab(Not_present),
      Pte(pe) : Ptab(Pte(pe With [accessed := true]))
    EndCases,
  Pdir(pdir) :
    Cases pdir Of
      Not_present : Pdir(Not_present),
      Pde_pt(pe) : Pdir(Pde_pt(pe With [accessed := true])),
      Pde_4m(pe) : Pdir(Pde_4m(pe With [accessed := true]))
    EndCases
EndCases

set_dirty(pt) : Paging_type =
Cases pt Of
  Ptab(ptab) :
    Cases ptab Of
      Not_present : Ptab(Not_present),
      Pte(pe) : Ptab(Pte(pe With [dirty := true]))
    EndCases,
  Pdir(pdir) :
    Cases pdir Of
      Not_present : Pdir(Not_present),
      Pde_pt(pe) : Pdir(Pde_pt(pe)),
      Pde_4m(pe) : Pdir(Pde_4m(pe With [dirty := true]))
    EndCases
EndCases

set_reference(pt : Paging_type, access : Memory_access) : Paging_type =
If Write?(access) Then
  set_accessed(set_dirty(pt))
Else
  set_accessed(pt)
Endif

present?(pt) : bool =
Cases pt Of
  Ptab(ptab) : Not not_present?(ptab),
  Pdir(pdir) : Not not_present?(pdir)
EndCases

% proof status :-
% base_TCC1 : TCC Obligation

% proof status :-
% base_TCC2 : TCC Obligation

base(pt : (present?)) : Memory_Address_4G =
Cases pt Of
  Ptab(ptab) : page_base(pte(ptab)),

```

B PVS Theory Sources

```

Pdir(pdir) :
  Cases pdir Of
    Pde_pt(pe) : page_table_base(pe),
    Pde_4m(pe) : page_base(pe)
  EndCases
EndCases

accessible?(pt : (present?), access : Memory_access) : bool =
  Cases pt Of
    Ptab(ptab) : member(access, access(pte(ptab))),
    Pdir(pdir) :
      Cases pdir Of
        Pde_pt(pe) : member(access, access(pe)),
        Pde_4m(pe) : member(access, access(pe))
      EndCases
    EndCases

privileged?(pt : (present?), priv : Memory_privilege) : bool =
  Cases pt Of
    Ptab(ptab) : Not Supervisor?(priv) And User?(privilege(pte(ptab))),
    Pdir(pdir) :
      Cases pdir Of
        Pde_pt(pe) : Not Supervisor?(priv) And User?(privilege(pe)),
        Pde_4m(pe) : Not Supervisor?(priv) And User?(privilege(pe))
      EndCases
    EndCases

is_leaf?(pt : (present?)) : bool =
  Not (pdir?(pt) And pde_pt?(pdir(pt)))

pdir_entry?(pt) : bool =
  pdir?(pt)

ptab_entry?(pt) : bool =
  ptab?(pt)

paging_type?(lvl : Level, pe : Paging_type) : bool =
  Cond
    lvl = 0 -> pdir_entry?(pe),
    lvl = 1 -> ptab_entry?(pe)
  EndCond

% proof status :-
set_reference_range : Lemma Forall (lvl : Level, pt : (range_pt(lvl)), access : Memory_access) :
  range_pt(lvl)(set_reference(pt, access))

% proof status :-
set_reference_pdir_entry : Lemma Forall (pt1, pt2 : Paging_type, access : Memory_access) :
  set_reference(pt1, access) = set_reference(pt2, access) Implies
  pdir_entry?(pt1) = pdir_entry?(pt2)

% proof status :-

```

```

set_reference_ptab_entry : Lemma Forall (pt1, pt2 : Paging_type, access : Memory_access) :
  set_reference(pt1, access) = set_reference(pt2, access) Implies
  ptab_entry?(pt1) = ptab_entry?(pt2)

% proof status :-)
set_reference_paging_type : Lemma Forall (pt1, pt2 : Paging_type, access : Memory_access, lvl : Level) :
  set_reference(pt1, access) = set_reference(pt2, access) Implies
  paging_type?(lvl, pt1) = paging_type?(lvl, pt2)

% proof status :-)
set_reference_present : Lemma Forall (pt1, pt2 : Paging_type, access : Memory_access) :
  set_reference(pt1, access) = set_reference(pt2, access) Implies
  present?(pt1) = present?(pt2)

% proof status :-)
% set_reference_base_TCC1 : TCC Obligation

% proof status :-)
set_reference_base : Lemma Forall (pt1, pt2 : Paging_type, access : Memory_access) :
  set_reference(pt1, access) = set_reference(pt2, access) And present?(pt1) Implies
  base(pt1) = base(pt2)

% proof status :-)
set_reference_accessible : Lemma Forall (pt1, pt2 : Paging_type, access, ac1 : Memory_access) :
  set_reference(pt1, access) = set_reference(pt2, access) And present?(pt1) Implies
  accessible?(pt1, ac1) = accessible?(pt2, ac1)

% proof status :-)
set_reference_privileged : Lemma
Forall (pt1, pt2 : Paging_type, access : Memory_access, priv : Memory_privilege) :
  set_reference(pt1, access) = set_reference(pt2, access) And present?(pt1) Implies
  privileged?(pt1, priv) = privileged?(pt2, priv)

% proof status :-)
set_reference_leaf : Lemma Forall (pt1, pt2 : Paging_type, access : Memory_access) :
  set_reference(pt1, access) = set_reference(pt2, access) And present?(pt1) Implies
  is_leaf?(pt1) = is_leaf?(pt2)

End Paging_type_helpers

% To do: remove this theory
Paging_Datatype : Theory
Begin

Importing Paging_type_helpers,
  PTE_Datatype, PDE_Datatype, Interpreted_Data_Lift

% proof status :-)
% paging_data_type_TCC1 : TCC Obligation

% proof status :-)

```

B PVS Theory Sources

```
% paging_data_type_TCC2 : TCC Obligation

% proof status :-)
% paging_data_type_TCC3 : TCC Obligation

% proof status :-)
% paging_data_type_TCC4 : TCC Obligation

% proof status :-)
% paging_data_type_TCC5 : TCC Obligation

% proof status :-)
% paging_data_type_TCC6 : TCC Obligation

% proof status :-)
% paging_data_type_TCC7 : TCC Obligation

paging_data_type(lvl : Level) : (pod_data_type?(range_pt(lvl))) =
  Cond
    lvl = 0 -> dt_lift[Pde_type, (range_pt(lvl))](pde_data_type,
      restrict[(pdir?), (range_pt(lvl)), Pde_type](pdir), Pdir),
    lvl = 1 -> dt_lift[Pte_type, (range_pt(lvl))](pte_data_type,
      restrict[(ptab?), (range_pt(lvl)), Pte_type](ptab), Ptab)
  EndCond

pe_size : nat = 2 % log2 size of page table entries

paging_data_type_size : Axiom
  Forall (lvl : Level) :
    size(uidt(paging_data_type(lvl))) = expt(2, pe_size)

End Paging_Datatype

PF_Error_Code_Type : Theory
Begin
  Importing Interpreted_Data, Error_Code_Types

  pf_ec_type_exists : Axiom
    Exists(x : (pod_data_type?[PF_error_code_type])) : True

  % proof status :-)
  % pf_ec_data_type_TCC1 : TCC Obligation

  pf_ec_data_type : (pod_data_type?[PF_error_code_type])

End PF_Error_Code_Type

Address_Datatype : Theory
Begin
```

Importing Interpreted_Data

```
address_data_type_exists : Axiom
  Exists(x : (pod_data_type?[Memory_Address_4G])) : True
```

```
% proof status :-)
% address_data_type_TCC1 : TCC Obligation
```

```
address_data_type : (pod_data_type?[Memory_Address_4G])
```

```
End Address_Datatype
```

```
Segment_Types : Theory
```

```
Begin
```

```
Importing Memory_privilege, IA32
```

```
Segment_type : Type = {Data_RO_Accessed, % 0001
  Data_RW_Accessed, % 0011
  Code_EO_Accessed, % 1001
  Code_ER_Accessed, % 1011
  % System
  LDT, % 0010
  TSS32_Avail, % 1001
  Interrupt_Gate, % 1110
  Trap_Gate % 1111
}
```

```
Segment_Selector_type : Type = [#
  index : below[expt(2,13)],
  table_indicator : bool, % GDT = false, LDT = true
  requested_privilege_level : Memory_privilege
#]
```

```
Access_Rights_type : Type = [#
  segment_type : Segment_type,
  system : bool,
  privilege : Memory_privilege,
  present : bool,
  % bits 11:8 reserved
  avail : bool,
  long_mode : bool,
  op_size_32 : bool, % true = 32 bit, false = 16 bit
  gran : bool,
  unusable : bool
  % bits 31:7 reserved
#]
```

```
% In the VMCS, IA32 stores the elements of this record in separate fields,
% Only the selector field is architecturally visible. The remaining
% fields are shadow fields.
```


B PVS Theory Sources

```
Segment_Reg_type : Type = [#  
  selector : Segment_Selector_type,  
  base_address : Memory_Address_4G,  
  limit : below[expt(2,32)],  
  access_rights : Access_Rights_type  
#]  
  
segment_to_priv(cs : Segment_Reg_type) : Memory_privilege =  
  cs'access_rights'privilege
```

End Segment_Types

Segment_Reg_Datatype : **Theory**

Begin

Importing Interpreted_Data, Segment_Types

% Segment Selector

segment_selector_type_exists : **Axiom**

Exists (x : (pod_data_type?[Segment_Selector_type])) : **True**

% proof status :-)

% segment_selector_data_type_TCC1 : TCC Obligation

segment_selector_data_type : (pod_data_type?[Segment_Selector_type])

% Segment Register

segment_reg_data_type_exists : **Axiom**

Exists(x : (pod_data_type?[Segment_Reg_type])) : **True**

% proof status :-)

% segment_reg_data_type_TCC1 : TCC Obligation

segment_reg_data_type : (pod_data_type?[Segment_Reg_type])

End Segment_Reg_Datatype

Control_Register_Types : **Theory**

Begin

Importing IA32

CR0_type : **Type** = [#

foo : bool,

bar : bool

#]

CR2_type : **Type** = Memory_Address_4G

```
CR4_type : Type = [#
  foo : bool
#]
```

End Control_Register_Types

Control_Register_Datatype : **Theory**
Begin

Importing Control_Register_Types, Interpreted_Data

```
cr0_data_type_exists : Axiom
  Exists(x : (pod_data_type?[CR0_type])) : True
```

```
% proof status :-)
% cr0_data_type_TCC1 : TCC Obligation
```

```
cr0_data_type : (pod_data_type?[CR0_type])
```

```
cr2_data_type_exists : Axiom
  Exists(x : (pod_data_type?[CR2_type])) : True
```

```
% proof status :-)
% cr2_data_type_TCC1 : TCC Obligation
```

```
cr2_data_type : (pod_data_type?[CR2_type])
```

```
cr4_data_type_exists : Axiom
  Exists(x : (pod_data_type?[CR4_type])) : True
```

```
% proof status :-)
% cr4_data_type_TCC1 : TCC Obligation
```

```
cr4_data_type : (pod_data_type?[CR4_type])
```

End Control_Register_Datatype

B.21 *physical_memory.pvs*

% specification of physical memory

Physical_Memory [(**Importing** IA32) min, max : Memory_Address] : **Theory**
Begin

IMPORTING ExprResult, Unit

```
Physical_memory : Type = [Address -> Byte]
```

```
% read & write
```

B PVS Theory Sources

```
s : Var Physical_memory
a : Var Address
b : Var Byte
bl : Var list[Byte]
cp : Var bool % cross page access

physical_read_side_effect(id : Address, bl, cp)(s) : ExprResult[Physical_memory, list[Byte]] =
  %Cases id Of
  %Else
  OK(s, bl)
  %EndCases

physical_write_side_effect(id : Address, bl, cp)(s) : ExprResult[Physical_memory, list[Byte]] =
  %Cases id Of
  %Else
  OK(s, bl)
  %EndCases

physical_read(a)(s) : ExprResult[Physical_memory, Byte] =
  If in_memory(min, max)(a) Then
    OK(s, s(a))
  Else
    Fatal
  Endif

physical_write(a : Address, b : Byte)(s) : ExprResult[Physical_memory, Unit] =
  If in_memory(min, max)(a) Then
    OK(s With [(a) := b], unit)
  Else
    Fatal
  Endif

End Physical_Memory
```

```
Physical_Memory_Corollaries [(Importing IA32) min, max : Memory_Address ] : Theory
Begin
```

```
IMPORTING Physical_Memory[min, max]

s : Var Physical_memory
a : Var Address
b : Var Byte

%%%%%% Next states

% proof status :-)
% physical_read_ok_next_state_TCC1 : TCC Obligation

% proof status :-)
physical_read_ok_next_state : Lemma
  OK?(physical_read(a)(s)) Implies state(physical_read(a)(s)) = s
```

```
% proof status :-)
% physical_write_ok_next_state_TCC1 : TCC Obligation
```

```
% proof status :-)
physical_write_ok_next_state : Lemma
  OK?(physical_write(a, b)(s)) Implies
    state(physical_write(a, b)(s)) = s with [(a) := b]
```

```
%%%%%%%%% Returned data
```

```
% proof status :-)
physical_read_ok_get_data : Lemma
  OK?(physical_read(a)(s)) Implies
    data(physical_read(a)(s)) = s(a)
```

End Physical_Memory_Corollaries

Physical_Memory_Properties [(**Importing** IA32) min, max : Memory_Address] : **Theory**
Begin

```
IMPORTING Physical_Memory[min, max]
```

```
s : Var Physical_memory
a : Var Address
b : Var Byte
```

```
%%%%%%%%% utility lemma
```

```
% proof status :-)
physical_read_ok : Lemma in_memory(min, max)(a) Implies OK?(physical_read(a)(s))
```

```
% proof status :-)
physical_write_ok : Lemma in_memory(min, max)(a) Implies OK?(physical_write(a, b)(s))
```

End Physical_Memory_Properties

Memory_Physical_Memory[(**Importing** IA32) min, max : Memory_Address] : **Theory**
% Defines the physical memory instance of memory

Begin

```
IMPORTING Physical_Memory[min, max]
```

```
IMPORTING Memory[Physical_memory]
```

```
%% IMPORTING Transformer_Result_State
```

```
%%%%%%%%% Public Interface %%%%%%%%%%
```

```
phy_mem : Memory_struct = (#
```

B PVS Theory Sources

```
memory_read := physical_read,
memory_write := physical_write,

memory_write_side_effect := physical_write_side_effect,
memory_read_side_effect := physical_read_side_effect
#)
```

End Memory_Physical_Memory

B.22 plain_memory.pvs

Plain_Memory[State : **Type**] : **Theory**
% plain memory definition and immediate utility results
Begin

IMPORTING Memory_Change_2

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% Plain Memory Definition
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
Plain_Memory : Type = [#
  mem : Memory_struct[State],
  states : PRED[State],
  ro_addr : PRED[Address],
  rw_addr : PRED[Address],
  other_actions : PRED[[State -> SuperResult[State]]]
#]
```

pm : **Var** Plain_Memory

addr : **Var** Address
size : **Var** nat

```
% We need to add one additional set of addresses to express that
% certain device state remains unchanged despite blessed accesses.
% Thereby the memory itself needs not to be blessed.
%
% Example: writing to RAM does not change the APIC EOI register state.
%
% Read, write, other_action and side effect transformers must establish
% an invariant which states:
% - that when reading a subset of the addresses is allowed, the same
% values are read, and
% - that device accesses behave the same before and after the blessed
% memory access
% - This rules out that the memory access counter is in the above set
% which is what we desire.
% - How can we formalize this point? How do we prevent the device from
```

*% changing internal state which may only be seen after a sequence of
% future accesses?*

```

plain_memory?(pm) : bool =
  % there is an invariant hidden, see below
  unchanged_memory_invariant?(pm'mem, pm'states,
    union(
      union(pm'other_actions,
        memory_read_transformers(pm'mem, union(pm'ro_addr, pm'rw_addr))),
      union(memory_read_side_effect_super_transformers(pm'mem,
        union(pm'ro_addr, pm'rw_addr)),
        memory_write_side_effect_super_transformers(pm'mem, pm'rw_addr))),
    union(pm'ro_addr, pm'rw_addr))
  AND
  unchanged_memory_invariant?(pm'mem, pm'states,
    memory_write_transformers(pm'mem, pm'rw_addr),
    pm'ro_addr)
  AND
  unchanged_memory_write_invariant?(pm'mem, pm'states, pm'rw_addr)
  AND
  changed_memory_invariant?(pm'mem, pm'states, pm'rw_addr)
  AND
  transformers_ok?(pm'states,
    union(
      union(memory_read_transformers(pm'mem, union(pm'ro_addr, pm'rw_addr)),
        memory_write_transformers(pm'mem, pm'rw_addr)),
      union(memory_read_side_effect_super_transformers(pm'mem,
        union(pm'ro_addr, pm'rw_addr)),
        memory_write_side_effect_super_transformers(pm'mem, pm'rw_addr))))
  And
  side_effect_content_unchanged(union(pm'ro_addr, pm'rw_addr), pm'states,
    memory_read_side_effect(pm'mem))
  And
  side_effect_content_unchanged(pm'rw_addr, pm'states,
    memory_write_side_effect(pm'mem))

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% Utility results
%
% transformer invariants
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% proof status :-)
plain_memory_invariant_read_transformers_ro_addr : Lemma
  plain_memory?(pm) Implies
    transformer_invariant?(pm'states,
      memory_read_transformers(pm'mem, pm'ro_addr))

% proof status :-)

```

B PVS Theory Sources

```
plain_memory_invariant_read_transformers_rw_addr : Lemma
  plain_memory?(pm) Implies
    transformer_invariant?(pm'states,
      memory_read_transformers(pm'mem, pm'rw_addr))

% proof status :-)
plain_memory_invariant_read_transformers_ro_rw_addr : Lemma
  plain_memory?(pm) Implies
    transformer_invariant?(pm'states,
      memory_read_transformers(pm'mem, union(pm'ro_addr, pm'rw_addr)))

% proof status :-)
plain_memory_invariant_write_transformers_rw_addr : Lemma
  plain_memory?(pm) Implies
    transformer_invariant?(pm'states,
      memory_write_transformers(pm'mem, pm'rw_addr))

% proof status :-)
plain_memory_invariant : Lemma
  plain_memory?(pm) Implies
    transformer_invariant?(pm'states,
      union(
        pm'other_actions,
        union(union(memory_read_transformers(pm'mem,
          union(pm'ro_addr, pm'rw_addr)),
            memory_write_transformers(pm'mem, pm'rw_addr)),
          union(memory_read_side_effect_super_transformers(pm'mem,
            union(pm'ro_addr, pm'rw_addr)),
            memory_write_side_effect_super_transformers(pm'mem,
              pm'rw_addr))))))

% proof status :-)
plain_memory_unchanged_invariant : Lemma
  plain_memory?(pm) Implies
    unchanged_memory_invariant?(pm'mem, pm'states,
      union(
        union(pm'other_actions,
          memory_read_transformers(pm'mem, union(pm'ro_addr, pm'rw_addr))),
        union(memory_read_side_effect_super_transformers(pm'mem,
          union(pm'ro_addr, pm'rw_addr)),
          memory_write_side_effect_super_transformers(pm'mem, pm'rw_addr))),
        union(pm'ro_addr, pm'rw_addr))

% proof status :-)
plain_memory_unchanged_memory_invariant_write : Lemma
  plain_memory?(pm) Implies
    unchanged_memory_invariant?(pm'mem, pm'states,
      memory_write_transformers(pm'mem, pm'rw_addr),
      pm'ro_addr)

% proof status :-)
```

```

plain_memory_unchanged_memory_write_invariant : Lemma
  plain_memory?(pm) Implies
    unchanged_memory_write_invariant?(pm'mem, pm'states, pm'rw_addr)

% proof status :-)
plain_memory_changed_memory_invariant : Lemma
  plain_memory?(pm) Implies
    changed_memory_invariant?(pm'mem, pm'states, pm'rw_addr)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% transformer invariants on blocks
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% proof status :-)
plain_memory_invariant_read_block : Lemma
  plain_memory?(pm) And
  subset?(address_block(addr, size), union(pm'ro_addr, pm'rw_addr))
  Implies
    transformer_invariant?(pm'states, memory_read_transformers(pm'mem,
      address_block(addr, size)))

% proof status :-)
plain_memory_invariant_write_block : Lemma
  plain_memory?(pm) And
  subset?(address_block(addr, size), pm'rw_addr)
  Implies
    transformer_invariant?(pm'states,
      memory_write_transformers(pm'mem, address_block(addr, size)))

% proof status :-)
plain_memory_transformer_invariant_read_side_effects : Lemma
  plain_memory?(pm) And
  subset?(address_block(addr, size), union(pm'ro_addr, pm'rw_addr))
  Implies
    transformer_invariant?(pm'states,
      memory_read_side_effect_super_transformers(pm'mem,
        address_block(addr, size)))

% proof status :-)
plain_memory_transformer_invariant_write_side_effects : Lemma
  plain_memory?(pm) And
  subset?(address_block(addr, size), pm'rw_addr)
  Implies
    transformer_invariant?(pm'states,
      memory_write_side_effect_super_transformers(pm'mem,
        address_block(addr, size)))

% proof status :-)
plain_memory_transformer_invariant_read_list_block : Lemma

```



```

plain_memory?(pm) And
subset?(address_block(addr, size), union(pm'ro_addr, pm'rw_addr))
Implies
  transformer_invariant?(pm'states,
    union(memory_read_transformers(pm'mem, address_block(addr, size)),
      memory_read_side_effect_super_transformers(pm'mem,
        address_block(addr, size))))

% proof status :-)
plain_memory_transformer_invariant_write_list_block : Lemma
plain_memory?(pm) And
subset?(address_block(addr, size), pm'rw_addr)
Implies
  transformer_invariant?(pm'states,
    union(memory_write_transformers(pm'mem, address_block(addr, size)),
      memory_write_side_effect_super_transformers(pm'mem,
        address_block(addr, size))))

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% transformers_ok?
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% proof status :-)
plain_memory_transformers_ok_read_ro_rw : Lemma
plain_memory?(pm)
Implies
  transformers_ok?(pm'states,
    memory_read_transformers(pm'mem, union(pm'ro_addr, pm'rw_addr)))

% proof status :-)
plain_memory_transformers_ok_read_block : Lemma
plain_memory?(pm) And
subset?(address_block(addr, size), union(pm'ro_addr, pm'rw_addr))
Implies
  transformers_ok?(pm'states, memory_read_transformers(pm'mem,
    address_block(addr, size)))

% proof status :-)
plain_memory_transformers_ok_write_rw : Lemma
plain_memory?(pm)
Implies
  transformers_ok?(pm'states,
    memory_write_transformers(pm'mem, pm'rw_addr))

% proof status :-)
plain_memory_transformers_ok_write_block : Lemma
plain_memory?(pm) And

```

```

subset?(address_block(addr, size), pm'rw_addr)
Implies
  transformers_ok?(pm'states,
    memory_write_transformers(pm'mem, address_block(addr, size)))

% proof status :-)
plain_memory_transformers_ok_read_side_effects_ro_rw : Lemma
plain_memory?(pm)
Implies
  transformers_ok?(pm'states,
    memory_read_side_effect_super_transformers(pm'mem,
      union(pm'ro_addr, pm'rw_addr)))

% proof status :-)
plain_memory_transformers_ok_read_side_effects_block : Lemma
plain_memory?(pm) And
subset?(address_block(addr, size), union(pm'ro_addr, pm'rw_addr))
Implies
  transformers_ok?(pm'states,
    memory_read_side_effect_super_transformers(pm'mem,
      address_block(addr, size)))

% proof status :-)
plain_memory_transformers_ok_write_side_effects_ro_rw : Lemma
plain_memory?(pm)
Implies
  transformers_ok?(pm'states,
    memory_write_side_effect_super_transformers(pm'mem, pm'rw_addr))

% proof status :-)
plain_memory_transformers_ok_write_side_effects_block : Lemma
plain_memory?(pm) And
subset?(address_block(addr, size), pm'rw_addr)
Implies
  transformers_ok?(pm'states,
    memory_write_side_effect_super_transformers(pm'mem,
      address_block(addr, size)))

% proof status :-)
plain_memory_transformers_ok_read_list_block : Lemma
plain_memory?(pm) And
subset?(address_block(addr, size), union(pm'ro_addr, pm'rw_addr))
Implies
  transformers_ok?(pm'states,
    union(memory_read_transformers(pm'mem, address_block(addr, size)),
      memory_read_side_effect_super_transformers(pm'mem,
        address_block(addr, size))))

% proof status :-)

```

B PVS Theory Sources

```

plain_memory_transformers_ok_write_list_block : Lemma
  plain_memory?(pm) And
  subset?(address_block(addr, size), pm'rw_addr)
  Implies
    transformers_ok?(pm'states,
      union(memory_write_transformers(pm'mem, address_block(addr, size)),
        memory_write_side_effect_super_transformers(pm'mem,
          address_block(addr, size))))

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% unchanged_memory_invariant
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% proof status :-)
plain_memory_unchanged_read_block_rw_addr : Lemma
  plain_memory?(pm) And
  subset?(address_block(addr, size), pm'rw_addr)
  Implies
    unchanged_memory_invariant?(pm'mem, pm'states,
      memory_read_transformers(pm'mem, address_block(addr, size)),
      pm'rw_addr)

% proof status :-)
plain_memory_unchanged_read_ro_rw_addr : Lemma
  plain_memory?(pm) Implies
    unchanged_memory_invariant?(pm'mem, pm'states,
      memory_read_transformers(pm'mem, union(pm'ro_addr, pm'rw_addr)),
      union(pm'ro_addr, pm'rw_addr))

% proof status :-)
plain_memory_unchanged_read_block_ro_rw_addr : Lemma
  plain_memory?(pm) And
  subset?(address_block(addr, size), union(pm'ro_addr, pm'rw_addr))
  Implies
    unchanged_memory_invariant?(pm'mem, pm'states,
      memory_read_transformers(pm'mem, address_block(addr, size)),
      union(pm'ro_addr, pm'rw_addr))

% proof status :-)
plain_memory_unchanged_write_block_ro_addr : Lemma
  plain_memory?(pm) And
  subset?(address_block(addr, size), pm'rw_addr)
  Implies
    unchanged_memory_invariant?(pm'mem, pm'states,
      memory_write_transformers(pm'mem, address_block(addr, size)),
      pm'ro_addr)

% unchanged invariants of reading
% proof status :-)

```

```

plain_memory_unchanged_read_block : Lemma
  plain_memory?(pm) And
  subset?(address_block(addr, size), union(pm'ro_addr, pm'rw_addr))
  Implies
    unchanged_memory_invariant?(pm'mem, pm'states,
      memory_read_transformers(pm'mem, address_block(addr, size)),
      address_block(addr, size))

% proof status :-)
plain_memory_unchanged_read_side_effect_block_ro_rw : Lemma
  plain_memory?(pm) And
  subset?(address_block(addr, size), union(pm'ro_addr, pm'rw_addr))
  Implies
    unchanged_memory_invariant?(pm'mem, pm'states,
      memory_read_side_effect_super_transformers(pm'mem,
        address_block(addr, size)),
      union(pm'ro_addr, pm'rw_addr))

% proof status :-)
plain_memory_unchanged_read_side_effect_block : Lemma
  plain_memory?(pm) And
  subset?(address_block(addr, size), union(pm'ro_addr, pm'rw_addr))
  Implies
    unchanged_memory_invariant?(pm'mem, pm'states,
      memory_read_side_effect_super_transformers(pm'mem,
        address_block(addr, size)),
      address_block(addr, size))

% proof status :-)
plain_memory_unchanged_write_side_effect_block_ro_rw : Lemma
  plain_memory?(pm) And
  subset?(address_block(addr, size), pm'rw_addr)
  Implies
    unchanged_memory_invariant?(pm'mem, pm'states,
      memory_write_side_effect_super_transformers(pm'mem,
        address_block(addr, size)),
      union(pm'ro_addr, pm'rw_addr))

% proof status :-)
plain_memory_unchanged_write_side_effect_block : Lemma
  plain_memory?(pm) And
  subset?(address_block(addr, size), pm'rw_addr)
  Implies
    unchanged_memory_invariant?(pm'mem, pm'states,
      memory_write_side_effect_super_transformers(pm'mem,
        address_block(addr, size)),
      address_block(addr, size))

%%%%%%%%%%
%
% unchanged_memory_write_invariant
%
```

B PVS Theory Sources

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% proof status :-)
plain_memory_unchanged_block : Lemma
  plain_memory?(pm) And
  subset?(address_block(addr, size), pm'rw_addr)
  Implies
    unchanged_memory_write_invariant?(pm'mem, pm'states,
                                        address_block(addr, size))

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% changed_memory_invariant
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% proof status :-)
plain_memory_changed_block : Lemma
  plain_memory?(pm) And
  subset?(address_block(addr, size), pm'rw_addr)
  Implies
    changed_memory_invariant?(pm'mem, pm'states, address_block(addr, size))

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% side_effect_content_unchanged
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% proof status :-)
plain_memory_side_effect_content_unchanged_read_block : Lemma
  plain_memory?(pm) And
  subset?(address_block(addr, size), union(pm'ro_addr, pm'rw_addr))
  Implies
    side_effect_content_unchanged(address_block(addr, size), pm'states,
                                  memory_read_side_effect(pm'mem))

% proof status :-)
plain_memory_side_effect_content_unchanged_write_block : Lemma
  plain_memory?(pm) And
  subset?(address_block(addr, size), pm'rw_addr)
  Implies
    side_effect_content_unchanged(address_block(addr, size), pm'states,
                                  memory_write_side_effect(pm'mem))

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% single reads/writes
%
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
% proof status :-)
```

```
plain_memory_memory_read_ok : Lemma Forall(s : State) :
  plain_memory?(pm) And
  pm'states(s) And
  union(pm'ro_addr, pm'rw_addr)(addr) Implies
    OK?(memory_read(pm'mem)(addr)(s))
```

```
% proof status :-)
```

```
% plain_memory_states_memory_read_TCC1 : TCC Obligation
```

```
% proof status :-)
```

```
plain_memory_states_memory_read : Lemma Forall(s : State) :
  plain_memory?(pm) And
  pm'states(s) And
  union(pm'ro_addr, pm'rw_addr)(addr) Implies
    pm'states(state(memory_read(pm'mem)(addr)(s)))
```

```
% proof status :-)
```

```
% plain_memory_memory_read_memory_read_TCC1 : TCC Obligation
```

```
% proof status :-)
```

```
% plain_memory_memory_read_memory_read_TCC2 : TCC Obligation
```

```
% proof status :-)
```

```
% plain_memory_memory_read_memory_read_TCC3 : TCC Obligation
```

```
% proof status :-)
```

```
plain_memory_memory_read_memory_read : Lemma
Forall(s : State, addr1, addr2 : Address) :
  plain_memory?(pm) And
  pm'states(s) And
  union(pm'ro_addr, pm'rw_addr)(addr1) And
  union(pm'ro_addr, pm'rw_addr)(addr2) Implies
    data(memory_read(pm'mem)(addr1)(state(memory_read(pm'mem)(addr2)(s)))) =
    data(memory_read(pm'mem)(addr1)(s))
```

```
% proof status :-)
```

```
plain_memory_memory_write_ok : Lemma Forall(s : State, byte : Byte) :
  plain_memory?(pm) And
  pm'states(s) And
  pm'rw_addr(addr) Implies
    OK?(memory_write(pm'mem)(addr, byte)(s))
```

```
% proof status :-)
```

```
% plain_memory_states_memory_write_TCC1 : TCC Obligation
```

```
% proof status :-)
```

```
plain_memory_states_memory_write : Lemma Forall(s : State, byte : Byte) :
  plain_memory?(pm) And
  pm'states(s) And
```

```

pm'rw_addr(addr) Implies
  pm'states(state(memory_write(pm'mem)(addr, byte)(s)))

% proof status :-)
% plain_memory_memory_read_memory_write_other_TCC1 : TCC Obligation

% proof status :-)
% plain_memory_memory_read_memory_write_other_TCC2 : TCC Obligation

% proof status :-)
% plain_memory_memory_read_memory_write_other_TCC3 : TCC Obligation

% proof status :-)
plain_memory_memory_read_memory_write_other : Lemma
Forall(s : State, ro_addr, rw_addr : Address, byte : Byte) :
  plain_memory?(pm) And
  pm'states(s) And
  union(pm'ro_addr, pm'rw_addr)(ro_addr) And
  pm'rw_addr(rw_addr) And
  Not ro_addr = rw_addr Implies
    data(memory_read(pm'mem)(ro_addr)
      (state(memory_write(pm'mem)(rw_addr, byte)(s)))) =
    data(memory_read(pm'mem)(ro_addr)(s))

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% restricting changes a la separation logic
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% proof status :-)
% stays_unchanged_TCC1 : TCC Obligation

% proof status :-)
% stays_unchanged_TCC2 : TCC Obligation

stays_unchanged(pm : (plain_memory?))
  (s1, s2 : (pm'states), addresses : PRED[Address]) : bool =
Forall(addr : Address) :
  union(pm'ro_addr, pm'rw_addr)(addr) And
  addresses(addr) Implies
    data(memory_read(pm'mem)(addr)(s1)) =
    data(memory_read(pm'mem)(addr)(s2))

% proof status :-)
% stays_unchanged_unchanged_TCC1 : TCC Obligation

% proof status :-)
% stays_unchanged_unchanged_TCC2 : TCC Obligation

```

```

% proof status :-)
stays_unchanged_unchanged : Lemma
  Forall(s1, s2 : State, addresses : PRED[Address], addr : Address) :
    plain_memory?(pm) And
    pm'states(s1) And pm'states(s2) And
    union(pm'ro_addr, pm'rw_addr)(addr) And
    stays_unchanged(pm)(s1, s2, addresses) And
    addresses(addr)
    Implies
      data(memory_read(pm'mem)(addr)(s1)) =
        data(memory_read(pm'mem)(addr)(s2))

% proof status :-)
stays_unchanged_symmetric : Lemma
  Forall(s1, s2 : State, addresses : PRED[Address]) :
    plain_memory?(pm) And
    pm'states(s1) And pm'states(s2) Implies
      stays_unchanged(pm)(s1, s2, addresses) =
        stays_unchanged(pm)(s2, s1, addresses)

% proof status :-)
stays_unchanged_mono : Lemma
  Forall(s1, s2 : State, addresses_1, addresses_2 : PRED[Address]) :
    plain_memory?(pm) And
    pm'states(s1) And pm'states(s2) And
    subset?(addresses_1, addresses_2) And
    stays_unchanged(pm)(s1, s2, addresses_2) Implies
      stays_unchanged(pm)(s1, s2, addresses_1)

only_changes(pm : (plain_memory?))
  (s1, s2 : (pm'states), addresses : PRED[Address]) : bool =
  stays_unchanged(pm)(s1, s2,
    union(pm'ro_addr, difference(pm'rw_addr, addresses)))

% proof status :-)
stays_unchanged_only_changes : Lemma
  Forall(s1, s2 : State, addresses : PRED[Address]) :
    plain_memory?(pm) And
    pm'states(s1) And pm'states(s2) And
    subset?(pm'ro_addr, addresses) Implies
      stays_unchanged(pm)(s1, s2, addresses) =
        only_changes(pm)(s1, s2, difference(pm'rw_addr, addresses))

% proof status :-)
stays_unchanged_only_changes_disjoint : Lemma
  Forall(s1, s2 : State, addresses_1, addresses_2 : PRED[Address]) :
    plain_memory?(pm) And
    pm'states(s1) And pm'states(s2) And
    subset?(addresses_2, union(pm'ro_addr, pm'rw_addr)) And

```


B PVS Theory Sources

```
disjoint?(addresses_1, addresses_2) And
only_changes(pm)(s1, s2, addresses_1)
Implies
  stays_unchanged(pm)(s1, s2, addresses_2)

% proof status :- )
only_changes_symmetric : Lemma
Forall(s1, s2 : State, addresses : PRED[Address]) :
  plain_memory?(pm) And
  pm'states(s1) And pm'states(s2) Implies
    only_changes(pm)(s1, s2, addresses) =
      only_changes(pm)(s2, s1, addresses)

% proof status :- )
only_changes_mono : Lemma
Forall(s1, s2 : State, addresses_1, addresses_2 : PRED[Address]) :
  plain_memory?(pm) And
  pm'states(s1) And pm'states(s2) And
  subset?(addresses_1, addresses_2) And
  only_changes(pm)(s1, s2, addresses_1) Implies
    only_changes(pm)(s1, s2, addresses_2)

% proof status :- )
% only_changes_unchanged_TCC1 : TCC Obligation

% proof status :- )
% only_changes_unchanged_TCC2 : TCC Obligation

% proof status :- )
only_changes_unchanged : Lemma
Forall(s1, s2 : State, addresses : PRED[Address]) :
  plain_memory?(pm) And
  pm'states(s1) And pm'states(s2) And
  only_changes(pm)(s1, s2, addresses) And
  (pm'ro_addr(addr) OR
   pm'rw_addr(addr) And Not addresses(addr))
Implies
  data(memory_read(pm'mem)(addr)(s1)) =
    data(memory_read(pm'mem)(addr)(s2))

% proof status :- )
% plain_memory_stays_unchanged_memory_read_TCC1 : TCC Obligation

% proof status :- )
plain_memory_stays_unchanged_memory_read : Lemma
Forall(s : State, addr : Address) :
  plain_memory?(pm) And
  pm'states(s) And
  union(pm'ro_addr, pm'rw_addr)(addr)
Implies
  stays_unchanged(pm)(s, state(memory_read(pm'mem)(addr)(s)),
```

```
union(pm'ro_addr, pm'rw_addr))
```

```
% proof status :-)
```

```
% plain_memory_stays_unchanged_memory_write_TCC1 : TCC Obligation
```

```
% proof status :-)
```

```
plain_memory_stays_unchanged_memory_write : Lemma
```

```
Forall(s : State, addr : Address, byte : Byte) :
```

```
plain_memory?(pm) And
```

```
pm'states(s) And
```

```
pm'rw_addr(addr)
```

```
Implies
```

```
stays_unchanged(pm)(s,
```

```
state(memory_write(pm'mem)(addr, byte)(s)),
```

```
remove(addr, union(pm'ro_addr, pm'rw_addr)))
```

```
End Plain_Memory
```

```
Abstract_Read_Write_Plain [State, Data : Type] : Theory
```

```
Begin
```

```
Importing Plain_Memory, Abstract_Read_Write
```

```
pm : Var Plain_Memory[State]
```

```
dt : Var (interpreted_data_type?[Data])
```

```
s : Var State
```

```
addr : Var Address
```

```
data : Var Data
```

```
write_data(pm, dt)(addr, data) : [State -> ExprResult[State, Unit]] =  
write_data(pm'mem, dt)(addr, data)
```

```
read_data(pm, dt)(addr) : [State -> ExprResult[State, Data]] =  
read_data(pm'mem, dt)(addr)
```

```
valid_in_mem(pm, dt)(addr) : [State -> bool] =  
valid_in_mem(pm'mem, dt)(addr)
```

```
End Abstract_Read_Write_Plain
```

```
Plain_Mem_Properties[State : Type] : Theory
```

```
%
```

```
% primitive (non-datatype) read and writes on plain memory
```

```
Begin
```

```
IMPORTING Plain_Memory[State]
```

```
pm : Var Plain_Memory[State]
```

B PVS Theory Sources

```

s : Var State
addr : Var Address
size : Var nat

% reading a block of memory is a transformer invariant

% writing to a block of memory is a transformer invariant

% proof status :-)
% plain_mem_write_list_read_list_TCC1 : TCC Obligation

% proof status :-)
plain_mem_write_list_read_list : Lemma
  Forall(size : nat, bl : list[Byte]) :
    size = length(bl) AND
    plain_memory?(pm) And
    pm'states(s) And
    subset?(address_block(addr, size), pm'rw_addr) And
    OK?(memory_write_list(pm'mem)(addr, bl)(s)) And
    OK?(memory_read_list(pm'mem)(addr, size)
      (state(memory_write_list(pm'mem) (addr, bl)(s))))
    Implies
      data(memory_read_list(pm'mem)(addr, size)
        (state(memory_write_list (pm'mem) (addr, bl) (s))))
      = bl

% proof status :-)
% plain_mem_q_read_list_TCC1 : TCC Obligation

% proof status :-)
plain_mem_q_read_list : Lemma Forall(q : [State -> SuperResult[State]]) :
  OK?(memory_read_list(pm'mem)(addr, size)(s)) And
  OK?(q(s)) And
  OK?(memory_read_list(pm'mem)(addr, size)(state(q(s)))) And
  plain_memory?(pm) And
  unchanged_memory_invariant?(pm'mem, pm'states, singleton(q),
    address_block(addr, size)) And
  pm'states(s) And
  subset?(address_block(addr, size), union(pm'ro_addr, pm'rw_addr))
  Implies
    data(memory_read_list(pm'mem)(addr, size) (state(q(s))))
    = data(memory_read_list(pm'mem)(addr, size) (s))

End Plain_Mem_Properties

Plain_Mem_Properties_2[State, Data : Type] : Theory
Begin
  IMPORTING Plain_Mem_Properties, Abstract_Read_Write_Plain

  pm : Var Plain_Memory[State]

```

```

dt : Var (interpreted_data_type?[Data])
s : Var State
addr : Var Address
data : Var Data

% proof status :-)
% plain_memory_inv_pred_write_data_TCC1 : TCC Obligation

% proof status :-)
plain_memory_inv_pred_write_data : Lemma
  plain_memory?(pm) And
  pm'states(s) And
  subset?(address_block(addr, size(uidt(dt))), pm'rw_addr) And
  OK?(write_data(pm, dt)(addr, data)(s))
  Implies
    pm'states(state(write_data(pm, dt)(addr, data)(s)))

% proof status :-)
plain_memory_read_data_ok : Lemma
  plain_memory?(pm) AND
  pm'states(s) And
  in_blessed_memory?(dt, addr, union(pm'ro_addr, pm'rw_addr)) And
  valid_in_mem(pm, dt)(addr)(s)
  Implies
    OK?(read_data(pm,dt)(addr)(s))

% proof status :-)
plain_memory_write_data_ok : Lemma
  plain_memory?(pm) AND
  pm'states(s) And
  in_blessed_memory?(dt, addr, pm'rw_addr)
  Implies
    OK?(write_data(pm,dt)(addr, data)(s))

% proof status :-)
% plain_memory_write_data_valid_TCC1 : TCC Obligation

% proof status :-)
plain_memory_write_data_valid : Lemma
  plain_memory?(pm) AND
  pm'states(s) And
  in_blessed_memory?(dt, addr, pm'rw_addr)
  Implies
    valid_in_mem(pm, dt)(addr)
    (state(write_data(pm, dt)(addr, data)(s)))

% proof status :-)
plain_memory_transformers_ok_write_data : Lemma
  plain_memory?(pm) AND

```

B PVS Theory Sources

```

in_blessed_memory?(dt, addr, pm'rw_addr)
Implies
  transformers_ok?(pm'states,
    singleton(expr_2_super(write_data(pm, dt) (addr, data))))

% proof status :-)
plain_memory_transformer_invariant_write_data : Lemma
  plain_memory?(pm) AND
  in_blessed_memory?(dt, addr, pm'rw_addr)
Implies
  transformer_invariant?(pm'states,
    singleton(expr_2_super(write_data(pm, dt) (addr, data))))

% proof status :-)
plain_memory_transformer_invariant_read_data : Lemma
  plain_memory?(pm) AND
  in_blessed_memory?(dt, addr, union(pm'ro_addr, pm'rw_addr))
Implies
  transformer_invariant?(pm'states,
    singleton(expr_2_super(read_data(pm, dt) (addr))))

% proof status :-)
plain_memory_unchanged_memory_invariant_write_data : Lemma
  plain_memory?(pm) AND
  in_blessed_memory?(dt, addr, pm'rw_addr)
IMPLIES
  unchanged_memory_invariant?(pm'mem, pm'states,
    singleton(expr_2_super(write_data(pm, dt)(addr, data))),
    difference(union(pm'ro_addr, pm'rw_addr),
      address_block(addr, size(uidt(dt)))))

% proof status :-)
plain_memory_unchanged_memory_invariant_read_data : Lemma
  plain_memory?(pm) AND
  in_blessed_memory?(dt, addr, union(pm'ro_addr, pm'rw_addr))
IMPLIES
  unchanged_memory_invariant?(pm'mem, pm'states,
    singleton(expr_2_super(read_data(pm, dt)(addr))),
    union(pm'ro_addr, pm'rw_addr))

% proof status :-)
% plain_memory_read_data_q_ok_TCC1 : TCC Obligation

% proof status :-)
plain_memory_read_data_q_ok : Lemma
Forall(q : [State -> SuperResult[State]]) :
  plain_memory?(pm) AND
  in_blessed_memory?(dt, addr, union(pm'ro_addr, pm'rw_addr)) AND
  unchanged_memory_invariant?(pm'mem, pm'states, singleton(q),
    address_block(addr, size(uidt(dt)))) And

```

```

pm'states(s) And
OK?(q(s)) And
valid_in_mem(pm,dt)(addr)(s) Implies
  OK?(read_data(pm, dt)(addr)(state(q(s))))

% proof status :-)
% plain_memory_read_data_q_same_TCC1 : TCC Obligation

% proof status :-)
% plain_memory_read_data_q_same_TCC2 : TCC Obligation

% proof status :-)
plain_memory_read_data_q_same : Lemma
Forall(q : [State -> SuperResult[State]]) :
  plain_memory?(pm) AND
  in_blessed_memory?(dt, addr, union(pm'ro_addr, pm'rw_addr)) AND
  unchanged_memory_invariant?(pm'mem, pm'states, singleton(q),
    address_block(addr, size(uidt(dt)))) And

  pm'states(s) And
  OK?(q(s)) And
  valid_in_mem(pm,dt)(addr)(s) Implies
    data(read_data(pm, dt)(addr)(state(q(s)))) =
      data(read_data(pm, dt)(addr)(s))

% proof status :-)
plain_memory_read_write_ok : Lemma
plain_memory?(pm) AND
pm'states(s) And
in_blessed_memory?(dt, addr, pm'rw_addr)
IMPLIES
  OK?((
    write_data(pm,dt)(addr, data) ##
    read_data(pm,dt)(addr)
  )(s)
  )

% proof status :-)
% plain_memory_read_write_res_TCC1 : TCC Obligation

% proof status :-)
plain_memory_read_write_res : Lemma
plain_memory?(pm) AND
pm'states(s) And
in_blessed_memory?(dt, addr, pm'rw_addr)
IMPLIES
  data((
    write_data(pm,dt)(addr, data) ##
    read_data(pm,dt)(addr)
  )(s))
  = data

```

B PVS Theory Sources

```
addresses : VAR PRED[Address]

% proof status :-)
plain_memory_unchanged_memory_ok_result : Lemma
  Forall (d : Data) :
    plain_memory?(pm) And
    subset?(addresses, union(pm'ro_addr, pm'rw_addr))
    Implies
      unchanged_memory_invariant?(pm'mem, pm'states,
        singleton(expr_2_super[State, Data](ok_result(d))),
        addresses)

end Plain_Mem_Properties_2

Plain_Mem_Properties_3[State, Data1, Data2 : Type] : Theory
%
% reads after write at different places return the previous item
Begin
  Importing Plain_Mem_Properties_2, Memory_Change_4
    % Memory_Change_To_Unit

  pm : Var Plain_Memory[State]

  dt1 : Var (interpreted_data_type?[Data1])
  dt2 : Var (interpreted_data_type?[Data2])
  s : Var State
  addr1, addr2 : Var Address
  data1 : Var Data1

  % proof status :-)
  plain_memory_read_write_other_ok : Lemma
    plain_memory?(pm) AND
    pm'states(s) And
    in_blessed_memory?(dt1, addr1, pm'rw_addr) And
    in_blessed_memory?(dt2, addr2, union(pm'ro_addr, pm'rw_addr)) And
    blocks_disjoint?(addr1, size(uidt(dt1)), addr2, size(uidt(dt2))) AND
    valid_in_mem(pm, dt2)(addr2)(s)
    IMPLIES
      OK?
      ((
        write_data(pm, dt1)(addr1, data1) ##
        read_data(pm, dt2)(addr2)
      )(s)
      )

  % proof status :-)
  % plain_memory_read_write_other_res_TCC1 : TCC Obligation

  % proof status :-)
  % plain_memory_read_write_other_res_TCC2 : TCC Obligation
```

```

% proof status :-)
plain_memory_read_write_other_res : Lemma
  plain_memory?(pm) AND
  pm'states(s) And
  in_blessed_memory?(dt1, addr1, pm'rw_addr) AND
  in_blessed_memory?(dt2, addr2, union(pm'ro_addr, pm'rw_addr)) AND
  blocks_disjoint?(addr1, size(uidt(dt1)), addr2, size(uidt(dt2))) AND
  valid_in_mem(pm,dt2)(addr2)(s)
  IMPLIES
    data((
      write_data(pm,dt1)(addr1, data1) ##
      read_data(pm,dt2)(addr2)
    )(s))
  =
  data(read_data(pm,dt2)(addr2)(s))

% proof status :-)
plain_memory_read_read_ok: Lemma
  plain_memory?(pm) AND
  pm'states(s) And
  in_blessed_memory?(dt1, addr1, union(pm'ro_addr, pm'rw_addr)) AND
  in_blessed_memory?(dt2, addr2, union(pm'ro_addr, pm'rw_addr)) AND
  valid_in_mem(pm,dt1)(addr1)(s) And
  valid_in_mem(pm,dt2)(addr2)(s)
  IMPLIES
    OK?(
      ( read_data(pm, dt1)(addr1) ##
        read_data(pm, dt2)(addr2)
      )(s)
    )

% proof status :-)
% plain_memory_read_read_TCC1 : TCC Obligation

% proof status :-)
% plain_memory_read_read_TCC2 : TCC Obligation

% proof status :-)
plain_memory_read_read: Lemma
  plain_memory?(pm) AND
  pm'states(s) And
  in_blessed_memory?(dt1, addr1, union(pm'ro_addr, pm'rw_addr)) AND
  in_blessed_memory?(dt2, addr2, union(pm'ro_addr, pm'rw_addr)) AND
  valid_in_mem(pm,dt1)(addr1)(s) And
  valid_in_mem(pm,dt2)(addr2)(s)
  IMPLIES
    data(
      ( read_data(pm, dt1)(addr1) ##
        read_data(pm, dt2)(addr2)
      )(s)
    )

```



```

= data(read_data(pm, dt2)(addr2)(s))

q : Var [State -> ExprResult[State, Data1]]
r : Var [Data1 -> [State -> ExprResult[State, Data2]]]
addresses : VAR PRED[Address]

% proof status :-)
plain_memory_unchanged_composition : Lemma
Forall (P : PRED[Data1]) :
  plain_memory?(pm) And
  subset?(addresses, union(pm'ro_addr, pm'rw_addr)) And
  (Forall (s : (pm'states)) : OK?(q(s)) Implies P(data(q(s)))) And
  unchanged_memory_invariant?(pm'mem, pm'states,
    singleton(expr_2_super(q)), addresses)

And
  (Forall (d : (P)) : unchanged_memory_invariant?(pm'mem, pm'states,
    singleton(expr_2_super(r(d))), addresses))

Implies
  unchanged_memory_invariant?(pm'mem, pm'states,
    singleton(expr_2_super(q ## r)), addresses)

End Plain_Mem_Properties_3

```

B.23 plain_memory_rewrites.pvs

```

% Rewrite rules
%=====

Block_Disjoint_Rewrites : Theory
Begin

  Importing Address_Util

  addr1, addr2 : Var Address
  sz1, sz2 : Var nat

  % different registers are block disjoint
  %-----

  % proof status :-)
  different_registers_blocks_disjoint : Lemma
  type_of(addr1) /= type_of(addr2)
  Implies
  blocks_disjoint?(addr1, sz1, addr2, sz2)

End Block_Disjoint_Rewrites

Plain_Mem_Rewrites1[State : Type] : Theory
Begin

  Importing Plain_Mem_Properties_3, ExpressionStatements

```

```

pm : Var Plain_Memory[State]
s : Var State

% proof status :-)
% pm_q_prop_TCC1 : TCC Obligation

pm_q_prop(pm : Plain_Memory[State])(res : StmtResult[State]) : bool =
  OK?(res) And pm'states(state(res))

% proof status :-)
pm_q_prop_ok_stmt : Lemma % !!! Lemma does not rewrite
Forall (q : [State -> StmtResult[State]]):
  plain_memory?(pm) And
  pm_q_prop(pm)(q(s))
Implies
  OK?(q(s))

End Plain_Mem_Rewrites1

Plain_Mem_Rewrites2[State, Data : Type] : Theory
Begin

Importing Plain_Mem_Rewrites1

pm : Var Plain_Memory[State]

dt : Var (interpreted_data_type?[Data])
s : Var State
addr : Var Address
data : Var Data

% in blessed memory rewrite
% proof status :-)
in_blessed_memory_rw_ro : Lemma % (P: 4)
  in_blessed_memory?(dt, addr, pm'rw_addr)
Implies
  in_blessed_memory?(dt, addr, union(pm'ro_addr, pm'rw_addr))

% pm_q_prop
%-----

% proof status :-)
% pm_q_prop_TCC1 : TCC Obligation

pm_q_prop(pm : Plain_Memory[State])(res : ExprResult[State, Data]) : bool =
  OK?(res) And pm'states(state(res))

% proof status :-)
pm_q_prop_ok_expr : Lemma % !!! Lemma does not rewrite

```

B PVS Theory Sources

```

Forall (q : [State -> ExprResult[State, Data]]) :
  plain_memory?(pm) And
  pm_q_prop(pm)(q(s))
Implies
  OK?(q(s))

% pm_q_prop simplification
%-----

% proof status :-)
pm_q_prop_single_read : Lemma %(P: 4)
  plain_memory?(pm) And
  pm'states(s) And
  in_blessed_memory?(dt, addr, union(pm'ro_addr, pm'rw_addr)) And
  valid_in_mem(pm, dt)(addr)(s)
Implies
  pm_q_prop(pm)(read_data(pm, dt)(addr)(s))

% proof status :-)
pm_q_prop_ok_result_single : Lemma % (P: 4)
  pm'states(s)
Implies
  pm_q_prop(pm)(ok_result(data)(s))

% plain_write_data_q_ok
%-----

% proof status :-)
plain_memory_write_ok_single : Lemma
  plain_memory?(pm) And
  pm'states(s) And
  in_blessed_memory?(dt, addr, pm'rw_addr)
Implies
  OK?(write_data(pm, dt)(addr, data)(s))

% proof status :-)
% pm_q_prop_read_TCC1 : TCC Obligation

pm_q_prop_read(pm : Plain_Memory[State], dt, addr)(res : StmtResult[State]) : bool =
  OK?(res) And OK?(read_data(pm, dt)(addr)(state(res))) And
  pm'states(state(res))

End Plain_Mem_Rewrites2

Plain_Mem_Rewrites3[State, Data, Data_q : Type] : Theory
Begin

  Importing Plain_Mem_Rewrites2

  pm : Var Plain_Memory[State]

  dt : Var (interpreted_data_type?[Data])

```

```

s : Var State
addr : Var Address

% proof status :-)
% pm_q_prop_read_TCC1 : TCC Obligation

pm_q_prop_read(pm : Plain_Memory[State], dt, addr)(res : ExprResult[State, Data_q]) : bool =
  OK?(res) And OK?(read_data(pm, dt)(addr)(state(res))) And
  pm'states(state(res))

```

End Plain_Mem_Rewrites3

```

% Rewrite rules for lemmas from Plain_Mem_Properties_3
Plain_Mem_Rewrites4[State, Data, Data_q : Type] : Theory
Begin

```

Importing Plain_Mem_Rewrites3

```
pm : Var Plain_Memory[State]
```

```
dt : Var (interpreted_data_type?[Data])
```

```
s : Var State
```

```
addr : Var Address
```

```
data : Var Data
```

```

% pm_q_prop
%-----

```

```
% proof status :-)
```

```
plain_memory_write_ok_q_expr : Lemma
Forall (q : [State -> ExprResult[State, Data_q]]) :
  plain_memory?(pm) And
  in_blessed_memory?(dt, addr, pm'rw_addr) And
  pm_q_prop(pm)(q(s))
```

Implies

```
OK?((q ## write_data(pm, dt)(addr, data))(s))
```

```
% proof status :-)
```

```
pm_q_prop_read_ok_expr : Lemma
Forall (q : [State -> ExprResult[State, Data_q]]) :
  plain_memory?(pm) And
  pm_q_prop_read(pm, dt, addr)(q(s))
```

Implies

```
OK?((q ## read_data(pm, dt)(addr))(s))
```

```
% proof status :-)
```

```
pm_q_prop_read_pm_q_prop_expr : Lemma
Forall (q : [State -> ExprResult[State, Data_q]]) :
  plain_memory?(pm) And
  in_blessed_memory?(dt, addr, union(pm'ro_addr, pm'rw_addr)) And
  pm_q_prop_read(pm, dt, addr)(q(s))
```

```

Implies
  pm_q_prop(pm)((q ## read_data(pm, dt)(addr))(s))

% proof status :-)
pm_q_prop_read_write_q_expr : Lemma
Forall (q : [State -> ExprResult[State, Data_q]]) :
  plain_memory?(pm) And
  in_blessed_memory?(dt, addr, pm'rw_addr) And
  pm_q_prop(pm)(q(s))
Implies
  pm_q_prop_read(pm, dt, addr)((q ## write_data(pm, dt)(addr, data))(s))

% proof status :-)
pm_q_prop_ok_result_q_expr : Lemma
Forall (q : [State -> ExprResult[State, Data_q]]) :
  plain_memory?(pm) And
  pm_q_prop(pm)(q(s))
Implies
  pm_q_prop(pm)((q ## ok_result[State, Data](data))(s))

data_q : Var Data_q

% proof status :-)
pm_q_prop_read_ok_result_single : Lemma
  plain_memory?(pm) And
  pm'states(s) And
  in_blessed_memory?(dt, addr, union(pm'ro_addr, pm'rw_addr)) And
  valid_in_mem(pm, dt)(addr)(s)
Implies
  pm_q_prop_read(pm, dt, addr)(ok_result[State, Data_q](data_q)(s))

% data
%-----

% proof status :-)
% plain_memory_read_write_q_data_expr_TCC1 : TCC Obligation

% proof status :-)
plain_memory_read_write_q_data_expr : Lemma
Forall (q : [State -> ExprResult[State, Data_q]]) :
  plain_memory?(pm) And
  in_blessed_memory?(dt, addr, pm'rw_addr) And
  pm_q_prop(pm)(q(s))
Implies
  data((q ## write_data(pm, dt)(addr, data) ##
      read_data(pm, dt)(addr))(s)) = data

% pm_q_prop simplification
%-----

dt1 : Var (interpreted_data_type?[Data])
dt2 : Var (interpreted_data_type?[Data_q])

```

```

addr1 : Var Address
addr2 : Var Address

% proof status :-)
pm_q_prop_read_write_other_single : Lemma %(P: 4)
  plain_memory?(pm) And
  blocks_disjoint?(addr1, size(uidt(dt1)), addr2, size(uidt(dt2))) And
  in_blessed_memory?(dt1, addr1, pm'rw_addr) And
  in_blessed_memory?(dt2, addr2, union(pm'ro_addr, pm'rw_addr)) And
  valid_in_mem(pm, dt2)(addr2)(s) And
  pm'states(s)
Implies
  pm_q_prop_read(pm, dt2, addr2)(write_data(pm, dt1)(addr1, data)(s))

% proof status :-)
pm_q_prop_read_read_single : Lemma %(P: 4)
  plain_memory?(pm) And
  in_blessed_memory?(dt1, addr1, union(pm'ro_addr, pm'rw_addr)) And
  in_blessed_memory?(dt2, addr2, union(pm'ro_addr, pm'rw_addr)) And
  valid_in_mem(pm, dt1)(addr1)(s) And
  valid_in_mem(pm, dt2)(addr2)(s) And
  pm'states(s)
Implies
  pm_q_prop_read(pm, dt2, addr2)(read_data(pm, dt1)(addr1)(s))

```

End Plain_Mem_Rewrites4

Plain_Mem_Rewrites5[State, Data : **Type**] : **Theory**
Begin

Importing Plain_Mem_Rewrites4

pm : **Var** Plain_Memory[State]

dt : **Var** (interpreted_data_type?[Data])

s : **Var** State

addr : **Var** Address

data : **Var** Data

% pm_q_prop simplification

%-----

% proof status :-)

```

plain_memory_write_ok_q_stmt : Lemma
  Forall (q : [State -> StmtResult[State]]):
    plain_memory?(pm) And
    in_blessed_memory?(dt, addr, pm'rw_addr) And
    pm_q_prop(pm)(q(s))
Implies
  OK?((q ## write_data(pm, dt)(addr, data))(s))

```

```

% proof status :-)
pm_q_prop_read_ok_stmt : Lemma
  Forall (q : [State -> StmtResult[State]]) :
    plain_memory?(pm) And
    pm_q_prop_read(pm, dt, addr)(q(s))
  Implies
    OK?((q ## read_data(pm, dt)(addr))(s))

% proof status :-)
pm_q_prop_read_pm_q_prop_stmt : Lemma
  Forall (q : [State -> StmtResult[State]]) :
    plain_memory?(pm) And
    in_blessed_memory?(dt, addr, union(pm'ro_addr, pm'rw_addr)) And
    pm_q_prop_read(pm, dt, addr)(q(s))
  Implies
    pm_q_prop(pm)((q ## read_data(pm, dt)(addr))(s))

% proof status :-)
pm_q_prop_read_write_q_stmt : Lemma
  Forall (q : [State -> StmtResult[State]]) :
    plain_memory?(pm) And
    in_blessed_memory?(dt, addr, pm'rw_addr) And
    pm_q_prop(pm)(q(s))
  Implies
    pm_q_prop_read(pm, dt, addr)((q ## write_data(pm, dt)(addr, data))(s))

% proof status :-)
pm_q_prop_ok_result_q_stmt : Lemma
  Forall (q : [State -> StmtResult[State]]) :
    plain_memory?(pm) And
    pm_q_prop(pm)(q(s))
  Implies
    pm_q_prop(pm)((q ## ok_result[State, Data](data))(s))

% proof status :-)
% plain_memory_read_write_q_data_stmt_TCC1 : TCC Obligation

% proof status :-)
plain_memory_read_write_q_data_stmt : Lemma
  Forall (q : [State -> StmtResult[State]]) :
    plain_memory?(pm) And
    in_blessed_memory?(dt, addr, pm'rw_addr) And
    pm_q_prop(pm)(q(s))
  Implies
    data((q ## write_data(pm, dt)(addr, data) ##
          read_data(pm, dt)(addr))(s)) = data

% proof status :-)
pm_q_prop_single_write : Lemma
  plain_memory?(pm) And
  in_blessed_memory?(dt, addr, pm'rw_addr) And
  pm'states(s)

```

Implies

```
pm_q_prop(pm)(write_data(pm, dt)(addr, data)(s))
```

```
% proof status :-)
```

```
pm_q_prop_read_write_single : Lemma
```

```
plain_memory?(pm) And
```

```
in_blessed_memory?(dt, addr, pm'rw_addr) And
```

```
pm'states(s)
```

Implies

```
pm_q_prop_read(pm, dt, addr)(write_data(pm, dt)(addr, data)(s))
```

```
% plain_memory_read_write_single_data = plain_memory_read_write_res
```

```
expr : VAR [State -> ExprResult[State, Data]]
```

```
stmt : VAR [State -> StmtResult[State]]
```

```
% proof status :-)
```

```
pm_q_prop_e2s : Lemma
```

```
pm_q_prop(pm)(e2s(expr)(s)) = pm_q_prop(pm)(expr(s))
```

```
% proof status :-)
```

```
pm_q_prop_stmt_e2s : Lemma
```

```
pm_q_prop(pm)(stmt(s)) Implies
```

```
pm_q_prop(pm)((stmt ## e2s(expr))(s)) = pm_q_prop(pm)((stmt ## expr)(s))
```

```
End Plain_Mem_Rewrites5
```

```
Plain_Mem_Rewrites6[State, Data1, Data2, Data_q : Type] : Theory
```

Begin

```
Importing Plain_Mem_Rewrites5
```

```
pm : Var Plain_Memory[State]
```

```
dt1 : Var (interpreted_data_type?[Data1])
```

```
dt2 : Var (interpreted_data_type?[Data2])
```

```
s : Var State
```

```
addr1, addr2 : Var Address
```

```
data1 : Var Data1
```

```
data2 : Var Data2
```

```
% pm_q_prop simplification
```

```
%-----
```

```
% proof status :-)
```

```
pm_q_prop_read_write_other_q_expr : Lemma %(P: 6)
```

```
Forall (q : [State -> ExprResult[State, Data_q]]) :
```

```
plain_memory?(pm) And
```

```
blocks_disjoint?(addr1, size(uidt(dt1)), addr2, size(uidt(dt2))) And
```

```
in_blessed_memory?(dt2, addr2, pm'rw_addr) And
```



```

    in_blessed_memory?(dt1, addr1, union(pm'ro_addr, pm'rw_addr)) And
    pm_q_prop_read(pm, dt1, addr1)(q(s))
Implies
    pm_q_prop_read(pm, dt1, addr1)((q ## write_data(pm, dt2)(addr2, data2))(s))

% proof status :- )
pm_q_prop_read_read_q_expr : Lemma %(P: 6)
Forall (q : [State -> ExprResult[State, Data_q]]) :
    plain_memory?(pm) And
    in_blessed_memory?(dt1, addr1, union(pm'ro_addr, pm'rw_addr)) And
    in_blessed_memory?(dt2, addr2, union(pm'ro_addr, pm'rw_addr)) And
    pm_q_prop_read(pm, dt1, addr1)(q(s)) And
    pm_q_prop_read(pm, dt2, addr2)(q(s))
Implies
    pm_q_prop_read(pm, dt2, addr2)((q ## read_data(pm, dt1)(addr1))(s))

% proof status :- )
pm_q_prop_read_ok_result_q_expr : Lemma % (P: 4)
Forall (q : [State -> ExprResult[State, Data_q]]) :
    plain_memory?(pm) And
    pm_q_prop_read(pm, dt2, addr2)(q(s))
Implies
    pm_q_prop_read(pm, dt2, addr2)((q ## ok_result[State, Data1](data1))(s))

% plain_memory_read_read_single_data = plain_memory_read_read

% proof status :- )
% plain_memory_read_read_q_data_expr_TCC1 : TCC Obligation

% proof status :- )
% plain_memory_read_read_q_data_expr_TCC2 : TCC Obligation

% proof status :- )
plain_memory_read_read_q_data_expr : Lemma %(P: 6)
Forall (q : [State -> ExprResult[State, Data_q]]) :
    plain_memory?(pm) And
    in_blessed_memory?(dt1, addr1, union(pm'ro_addr, pm'rw_addr)) And
    in_blessed_memory?(dt2, addr2, union(pm'ro_addr, pm'rw_addr)) And
    pm_q_prop_read(pm, dt2, addr2)(q(s)) And
    pm_q_prop_read(pm, dt1, addr1)(q(s))
Implies
    data((q ## read_data(pm, dt2)(addr2) ## read_data(pm, dt1)(addr1))(s)) =
    data((q ## read_data(pm, dt1)(addr1))(s))

End Plain_Mem_Rewrites6

Plain_Mem_Rewrites7 [State, Data1, Data2 : Type] : Theory
Begin

    Importing Plain_Mem_Rewrites6

```

```

pm : VAR Plain_Memory[State]
s : VAR State
expr : VAR [State -> ExprResult[State, Data1]]
expr1 : VAR [State -> ExprResult[State, Data2]]

dt1 : Var (interpreted_data_type?[Data1])
dt2 : Var (interpreted_data_type?[Data2])
addr1, addr2 : Var Address
data1 : Var Data1
data2 : Var Data2

% proof status :-)
pm_q_prop_read_write_other_q_stmt : Lemma %(P: 6)
Forall (q : [State -> StmtResult[State]]) :
  plain_memory?(pm) And
  blocks_disjoint?(addr1, size(uidt(dt1)), addr2, size(uidt(dt2))) And
  in_blessed_memory?(dt2, addr2, pm'rw_addr) And
  in_blessed_memory?(dt1, addr1, union(pm'ro_addr, pm'rw_addr)) And
  pm_q_prop_read(pm, dt1, addr1)(q(s))
Implies
  pm_q_prop_read(pm, dt1, addr1)((q ## write_data(pm, dt2)(addr2, data2))(s))

% proof status :-)
% plain_memory_read_write_other_single_data_TCC1 : TCC Obligation

% proof status :-)
% plain_memory_read_write_other_single_data_TCC2 : TCC Obligation

% proof status :-)
plain_memory_read_write_other_single_data : Lemma %(P: 4)
  plain_memory?(pm) And
  in_blessed_memory?(dt1, addr1, union(pm'ro_addr, pm'rw_addr)) And
  in_blessed_memory?(dt2, addr2, pm'rw_addr) And
  blocks_disjoint?(addr1, size(uidt(dt1)), addr2, size(uidt(dt2))) And
  pm_q_prop(pm)(read_data(pm, dt1)(addr1)(s)) And
  pm'states(s)
Implies
  data((write_data(pm, dt2)(addr2, data2) ## read_data(pm, dt1)(addr1))(s)) =
  data((read_data(pm, dt1)(addr1))(s))

% proof status :-)
pm_q_prop_read_read_q_stmt : Lemma %(P: 6)
Forall (q : [State -> StmtResult[State]]) :
  plain_memory?(pm) And
  in_blessed_memory?(dt1, addr1, union(pm'ro_addr, pm'rw_addr)) And
  in_blessed_memory?(dt2, addr2, union(pm'ro_addr, pm'rw_addr)) And
  pm_q_prop_read(pm, dt1, addr1)(q(s)) And
  pm_q_prop_read(pm, dt2, addr2)(q(s))
Implies
  pm_q_prop_read(pm, dt2, addr2)((q ## read_data(pm, dt1)(addr1))(s))

% proof status :-)

```

B PVS Theory Sources

```

pm_q_prop_read_ok_result_q_stmt : Lemma % (P: 4)
  Forall (q : [State -> StmtResult[State]]) :
    plain_memory?(pm) And
    pm_q_prop_read(pm, dt2, addr2)(q(s))
  Implies
    pm_q_prop_read(pm, dt2, addr2)((q ## ok_result[State, Data1](data1))(s))

% proof status :-)
% plain_memory_read_write_other_q_data_stmt_TCC1 : TCC Obligation

% proof status :-)
% plain_memory_read_write_other_q_data_stmt_TCC2 : TCC Obligation

% proof status :-)
plain_memory_read_write_other_q_data_stmt : Lemma %(P: 6)
  Forall (q : [State -> StmtResult[State]]) :
    plain_memory?(pm) And
    in_blessed_memory?(dt1, addr1, union(pm'ro_addr, pm'rw_addr)) And
    in_blessed_memory?(dt2, addr2, pm'rw_addr) And
    blocks_disjoint?(addr1, size(uidt(dt1)), addr2, size(uidt(dt2))) And
    pm_q_prop_read(pm, dt1, addr1)(q(s))
  Implies
    data((q ## write_data(pm, dt2)(addr2, data2) ## read_data(pm, dt1)(addr1))(s)) =
      data((q ## read_data(pm, dt1)(addr1))(s))

% proof status :-)
% plain_memory_read_read_q_data_stmt_TCC1 : TCC Obligation

% proof status :-)
% plain_memory_read_read_q_data_stmt_TCC2 : TCC Obligation

% proof status :-)
plain_memory_read_read_q_data_stmt : Lemma %(P: 6)
  Forall (q : [State -> StmtResult[State]]) :
    plain_memory?(pm) And
    in_blessed_memory?(dt1, addr1, union(pm'ro_addr, pm'rw_addr)) And
    in_blessed_memory?(dt2, addr2, union(pm'ro_addr, pm'rw_addr)) And
    pm_q_prop_read(pm, dt2, addr2)(q(s)) And
    pm_q_prop_read(pm, dt1, addr1)(q(s))
  Implies
    data((q ## read_data(pm, dt2)(addr2) ## read_data(pm, dt1)(addr1))(s)) =
      data((q ## read_data(pm, dt1)(addr1))(s))

% proof status :-)
pm_q_prop_read_e2s : Lemma
  pm_q_prop_read(pm, dt1, addr1)(expr1(s)) Implies
    pm_q_prop_read(pm, dt1, addr1)(e2s(expr1(s)))

stmt : VAR [State -> StmtResult[State]]

% proof status :-)
pm_q_prop_read_stmt_e2s : Lemma

```

```

pm_q_prop_read(pm, dt1, addr1)(stmt(s)) Implies
  pm_q_prop_read(pm, dt1, addr1)((stmt ## e2s(expr1))(s)) =
    pm_q_prop_read(pm, dt1, addr1)((stmt ## expr1)(s))

```

End Plain_Mem_Rewrites7

Plain_Mem_Rewrites8 [State, Data1, Data2, Data_q : **Type**] : **Theory**
Begin

Importing Plain_Mem_Rewrites7

```

pm : VAR Plain_Memory[State]
s : VAR State

```

```

dt1 : Var (interpreted_data_type?[Data1])
dt2 : Var (interpreted_data_type?[Data2])
addr1, addr2 : Var Address
data1 : Var Data1
data2 : Var Data2

```

```

% proof status :-)
% plain_memory_read_write_other_q_data_expr_TCC1 : TCC Obligation

```

```

% proof status :-)
% plain_memory_read_write_other_q_data_expr_TCC2 : TCC Obligation

```

```

% proof status :-)
plain_memory_read_write_other_q_data_expr : Lemma %(P: 6)
Forall (q : [State -> ExprResult[State, Data_q]]) :
  plain_memory?(pm) And
  in_blessed_memory?(dt1, addr1, union(pm'ro_addr, pm'rw_addr)) And
  in_blessed_memory?(dt2, addr2, pm'rw_addr) And
  blocks_disjoint?(addr1, size(uidt(dt1)), addr2, size(uidt(dt2))) And
  pm_q_prop_read(pm, dt1, addr1)(q(s))
Implies
  data((q ## write_data(pm, dt2)(addr2, data2) ## read_data(pm, dt1)(addr1))(s)) =
    data((q ## read_data(pm, dt1)(addr1))(s))

```

End Plain_Mem_Rewrites8

Plain_Mem_Rewrites : **Theory**
Begin

Importing Plain_Mem_Rewrites8

End Plain_Mem_Rewrites

```
Rewrite_Test[(Importing IA32) max : Memory_Address,
              State, Data1, Data2 : Type] : Theory
```

Begin

```
Importing Plain_Memory[State]
Importing Plain_Mem_Rewrites
```

```
pm : Var Plain_Memory[State]
```

```
dt1 : Var (interpreted_data_type?[Data1])
```

```
dt2 : Var (interpreted_data_type?[Data2])
```

```
s : Var State
```

```
addr1, addr2 : Var Address
```

```
data1 : Var Data1
```

```
data2 : Var Data2
```

```
% Test for Plain Mem Rewrites
```

```
%-----
```

```
% proof status :-)
```

```
rewrite_test : Lemma
```

```
  plain_memory?(pm) And
```

```
  in_blessed_memory?(dt1, addr1, pm'rw_addr) And
```

```
  in_blessed_memory?(dt2, addr2, pm'rw_addr) And
```

```
  blocks_disjoint?(addr1, size(uidt(dt1)), addr2, size(uidt(dt2))) And
```

```
  blocks_disjoint?(addr2, size(uidt(dt2)), addr1, size(uidt(dt1))) And
```

```
  pm'states(s)
```

```
Implies
```

```
  OK?(
```

```
    (write_data(pm, dt1)(addr1, data1) ##
```

```
    write_data(pm, dt2)(addr2, data2) ##
```

```
    read_data(pm, dt1)(addr1) ##
```

```
    write_data(pm, dt2)(addr2, data2) ##
```

```
    read_data(pm, dt2)(addr2) ##
```

```
    read_data(pm, dt1)(addr1)
```

```
  )(s))
```

```
% proof status :-)
```

```
% rewrite_test_data_TCC1 : TCC Obligation
```

```
% proof status :-)
```

```
rewrite_test_data : Lemma
```

```
  plain_memory?(pm) And
```

```
  in_blessed_memory?(dt1, addr1, pm'rw_addr) And
```

```
  in_blessed_memory?(dt2, addr2, pm'rw_addr) And
```

```
  blocks_disjoint?(addr1, size(uidt(dt1)), addr2, size(uidt(dt2))) And
```

```
  blocks_disjoint?(addr2, size(uidt(dt2)), addr1, size(uidt(dt1))) And
```

```
  pm'states(s)
```

```
Implies
```

```
  data(
```

```
    (write_data(pm, dt1)(addr1, data1) ##
```



```

    read_data(pm, dt2)(addr2) ##
    read_data(pm, dt1)(addr1)
  )(s) = data1

% proof status :-)
rewrite_test_data_eval_if_ok_ok : Lemma
  plain_memory?(pm) And
  in_blessed_memory?(dt1, addr1, pm'rw_addr) And
  in_blessed_memory?(dt2, addr2, pm'rw_addr) And
  blocks_disjoint?(addr1, size(uidt(dt1)), addr2, size(uidt(dt2))) And
  blocks_disjoint?(addr2, size(uidt(dt2)), addr1, size(uidt(dt1))) And
  pm'states(s)
Implies
  OK?(
    (write_data(pm, dt1)(addr1, data1) ##
     write_data(pm, dt2)(addr2, data2) ##
     read_data(pm, dt1)(addr1) ##
     write_data(pm, dt2)(addr2, data2) ##
     read_data(pm, dt2)(addr2) ##
     Lambda(d : Data2) :
       read_data(pm, dt1)(addr1))(s))

% proof status :-)
% rewrite_test_data_eval_if_ok_TCC1 : TCC Obligation

% proof status :-)
rewrite_test_data_eval_if_ok : Lemma
  plain_memory?(pm) And
  in_blessed_memory?(dt1, addr1, pm'rw_addr) And
  in_blessed_memory?(dt2, addr2, pm'rw_addr) And
  blocks_disjoint?(addr1, size(uidt(dt1)), addr2, size(uidt(dt2))) And
  blocks_disjoint?(addr2, size(uidt(dt2)), addr1, size(uidt(dt1))) And
  pm'states(s)
Implies
  data(
    (write_data(pm, dt1)(addr1, data1) ##
     write_data(pm, dt2)(addr2, data2) ##
     read_data(pm, dt1)(addr1) ##
     write_data(pm, dt2)(addr2, data2) ##
     read_data(pm, dt2)(addr2) ##
     Lambda(d : Data2) :
       read_data(pm, dt1)(addr1)
    )(s) = data1

% proof status :-)
rewrite_test_data_eval_if_ok_ok_result_ok : Lemma
  plain_memory?(pm) And
  in_blessed_memory?(dt1, addr1, pm'rw_addr) And
  in_blessed_memory?(dt2, addr2, pm'rw_addr) And
  blocks_disjoint?(addr1, size(uidt(dt1)), addr2, size(uidt(dt2))) And
  blocks_disjoint?(addr2, size(uidt(dt2)), addr1, size(uidt(dt1))) And

```

```

pm'states(s)
Implies
OK?[State, Data2](
  (write_data(pm, dt1)(addr1, data1) ##
   write_data(pm, dt2)(addr2, data2) ##
   read_data(pm, dt1)(addr1) ##
   write_data(pm, dt2)(addr2, data2) ##
   read_data(pm, dt2)(addr2) ##
   Lambda(d : Data2) :
     ok_result(d)
  )(s))

% proof status :-)
% rewrite_test_data_eval_if_ok_ok_result_TCC1 : TCC Obligation

% proof status :-)
rewrite_test_data_eval_if_ok_ok_result : Lemma
plain_memory?(pm) And
in_blessed_memory?(dt1, addr1, pm'rw_addr) And
in_blessed_memory?(dt2, addr2, pm'rw_addr) And
blocks_disjoint?(addr1, size(uidt(dt1)), addr2, size(uidt(dt2))) And
blocks_disjoint?(addr2, size(uidt(dt2)), addr1, size(uidt(dt1))) And
pm'states(s)
Implies
data(
  (write_data(pm, dt1)(addr1, data1) ##
   write_data(pm, dt2)(addr2, data2) ##
   read_data(pm, dt1)(addr1) ##
   write_data(pm, dt2)(addr2, data2) ##
   read_data(pm, dt2)(addr2) ##
   Lambda(d : Data2) :
     ok_result(d)
  )(s) = data2

End Rewrite_Test

% Rewrites: Plain Memory
% (auto-rewrite
% "plain_memory_write_ok_single"
% "plain_memory_write_ok_q_expr"
% "plain_memory_write_ok_q_stmt"
% "in_blessed_memory_rw_ro"
% "pm_q_prop_ok"
% "pm_q_prop_single_read"
% "pm_q_prop_single_write"
% "pm_q_prop_read_write_single"
% "pm_q_prop_read_write_other_single"
% "pm_q_prop_read_read_single"
% "plain_memory_read_write_other_single_data"
% "plain_memory_read_read"
% "plain_memory_read_write_res"
% "pm_q_prop_read_ok_expr"

```


B PVS Theory Sources

```
% "pm_q_prop_read_ok_stmt"  
% "pm_q_prop_read_pm_q_prop_expr"  
% "pm_q_prop_read_pm_q_prop_stmt"  
% "pm_q_prop_read_write_q_expr"  
% "pm_q_prop_read_write_q_stmt"  
% "plain_memory_read_write_q_data_expr"  
% "plain_memory_read_write_q_data_stmt"  
% "pm_q_prop_read_write_other_q_expr"  
% "pm_q_prop_read_write_other_q_stmt"  
% "pm_q_prop_read_read_q_expr"  
% "pm_q_prop_read_read_q_stmt"  
% "plain_memory_read_write_other_q_data_expr"  
% "plain_memory_read_write_other_q_data_stmt"  
% "plain_memory_read_read_q_data_expr"  
% "plain_memory_read_read_q_data_stmt"  
% "pm_q_prop_ok_result_single"  
% "pm_q_prop_ok_result_q_expr"  
% "pm_q_prop_ok_result_q_stmt"  
% "pm_q_prop_read_ok_result_single"  
% "pm_q_prop_read_ok_result_q_expr"  
% "pm_q_prop_read_ok_result_q_stmt"  
% "pm_q_prop_e2s"  
% "pm_q_prop_stmt_e2s"  
% "pm_q_prop_read_e2s"  
% "pm_q_prop_read_stmt_e2s"  
% )
```

B.24 ptab-sync-master-defs.pvs

reinterpret_cast_thy [State, Data1, Data2 : Type] : Theory
Begin

Importing Cpp_Verification

dt1 : **VAR** (interpreted_data_type?[Data1])
dt2 : **VAR** (interpreted_data_type?[Data2])

reinterpret_cast(dt1, dt2)(ex : ET[State, Data2]) : ET[State, Data1] =
(
 ex ## **lambda** (d2 : Data2) :
 allocate_stack(uidt(dt2)) ## **lambda** (a : Address) :
 write_data(pm, dt2)(a, d2) ##
 read_data(pm, dt1)(a) ## **lambda** (d1 : Data1) :
 deallocate_stack(uidt(dt2), a) ##
 ok_result[State, Data1](d1)
)

End reinterpret_cast_thy

ptab_types : **Theory**

Begin

Paging : **Type**
 Buddy : **Type**
 Ptab : **Type**

void : **Type**

End ptab_typesptab_sync_master_defs_state_transformers [State : **Type**] : **Theory****Begin****Importing** Cpp_Verification, ptab_types

```
static_cast_nocv_unsigned_long_long_reference_to_nocv_unsigned_long_int
  (ex : ET[State, (range(ulonglong))]) : ET[State, (range(ulong))] =
(
  ex ## lambda (t : (range(ulonglong))) :
  If true Then % (TODO) FIXME: dt_pred(ulonglong)(t) Then
    ok_result(t)
  Else
    fatal_result
  Endif
)
```

```
static_cast_nocv_void_nocv_pointer_to_nocv_Ptab_nocv_pointer
  (ex : ET[State, (range(pointer(void)))] : ET[State, (range(pointer(class("Ptab", false))))])
```

```
int_conv_nocv_unsigned_long_int_reference_to_const_unsigned_int
  (ex : ET[State, (range(ulong))] : ET[State, (range(uint))])
```

```
int_conv_nocv_unsigned_long_int_to_nocv_unsigned_int
  (ex : ET[State, (range(ulong))] : ET[State, (range(uint))])
```

End ptab_sync_master_defs_state_transformersptab_sync_master_defs : **Theory****Begin****Importing** ptab_sync_master_defs_state_transformers, reinterpret_cast_thy

```
% To do:
%=====
% - generate statically initialised variables and constants
% - generate offsets

% offsets
% MV: >> fix pointer
```

B PVS Theory Sources

```
Ofs_Paging_type : Type = [#
  val : nat
#]

offsets_Paging : Ofs_Paging_type

Ofs_Buddy_type : Type = [#
  l_base : nat,
  r_base : nat,
  p_base : nat
#]

offsets_Buddy : Ofs_Buddy_type

% Enum
ATTR_PRESENT : (range(ulonglong)) = shift_bits_right(1, 0)
ATTR_NOEXEC : (range(ulonglong)) = shift_bits_right(1, 63)
ATTR_ALL : (range(ulonglong)) = 0 %bitor(ATTR_NOEXEC, (shift_bits_right(1, 12) - 1))

% static members; static const initialisation is missing
levels : Address
bits_per_level : Address

allocator : Address

Ptab : Cpp_Type

End ptab_sync_master_defs
```

B.25 ptab-sync-master.pvs

```
ptab_sync_master[State : Type] : THEORY
BEGIN

  IMPORTING Cpp_Verification
  IMPORTING ptab_sync_master_defs

  % Function semantics for addr
  % File "nova/include/paging.h", line 69, character 9
  addr(receiver : Address)(ret_addr : Address) : ST[State] =
    return(pm, dt_ulonglong, ret_addr)(bitand(ulonglong)(member(id(receiver),
                                                                    offsets_Paging'val),
                                                                    unary_bitnot(int)(id(ATTR_ALL))))

  call_addr(receiver : ET[State, Semantics_reference[Paging]]) : ET[State,
    Semantics_ulonglong] =
    with_new_returnvar(pm, dt_ulonglong)(lambda (ret_addr : Address) :
      receiver ##
      (lambda (receiver_addr : Address):
        addr(receiver_addr)(ret_addr)))
```

```

% Function semantics for operator=
% File "nova/include/paging.h", line 16, character 1
operator_assign_Paging(receiver : Address, other : Address)(ret_addr : Address)
  : ST[State] =
  e2s(assign(pm, dt_ulonglong)(member(id(receiver), offsets_Paging'val),
                                member(id(other), offsets_Paging'val))) ##
  return(pm, dt_reference[Paging], ret_addr)(id(receiver))

call_operator_assign_Paging(receiver : ET[State,
                             Semantics_reference[Paging]],
                             other : ET[State, Semantics_reference[Paging]]) :
  ET[State, Semantics_reference[Paging]] =
with_new_returnvar(pm, dt_reference[Paging])(lambda (ret_addr : Address) :
  other ##
  (lambda (other_addr : Address):
  receiver ##
  (lambda (receiver_addr : Address):
  operator_assign_Paging(receiver_addr, other_addr)(ret_addr))))

% Function semantics for operator=
% File "nova/include/ptab.h", line 20, character 1
operator_assign_Ptab(receiver : Address, other : Address)(ret_addr : Address)
  : ST[State] =
  e2s(call_operator_assign(id(receiver), id(other))) ##
  return(pm, dt_reference[Ptab], ret_addr)(id(receiver))

call_operator_assign_Ptab(receiver : ET[State, Semantics_reference[Ptab]],
                           other : ET[State, Semantics_reference[Ptab]]) :
  ET[State, Semantics_reference[Ptab]] =
with_new_returnvar(pm, dt_reference[Ptab])(lambda (ret_addr : Address) :
  other ##
  (lambda (other_addr : Address):
  receiver ##
  (lambda (receiver_addr : Address):
  operator_assign_Ptab(receiver_addr, other_addr)(ret_addr))))

% Function semantics for present
% File "nova/include/paging.h", line 81, character 9
present(receiver : Address)(ret_addr : Address) : ST[State] =
  return(pm, dt_ulonglong, ret_addr)(bitand(ulonglong)(member(id(receiver),
                                                                offsets_Paging'val),
                                                                id(ATTR_PRESENT)))

call_present(receiver : ET[State, Semantics_reference[Paging]]) : ET[State,
                           Semantics_bool] =
with_new_returnvar(pm, dt_bool)(lambda (ret_addr : Address) :
  receiver ##
  (lambda (receiver_addr : Address):
  present(receiver_addr)(ret_addr)))

```

B PVS Theory Sources

```
% Function semantics for phys_to_virt
% File "nova/include/buddy.h", line 74, character 9
phys_to_virt(receiver : Address, phys : Address)(ret_addr : Address) :
  ST[State] =
  return(pm, dt_ulong, ret_addr)(plus(ulong)(minus(ulong)(id(phys),
                                                                    member(id(receiver),
                                                                    offsets_Buddy'p_base)),
                                                                    member(id(receiver),
                                                                    offsets_Buddy'l_base)))

call_phys_to_virt(receiver : ET[State, Semantics_reference[Buddy]],
                  phys : ET[State, Semantics_ulong] : ET[State,
                  Semantics_ulong] =
  with_new_returnvar(pm, dt_ulong)(lambda (ret_addr : Address) :
  with_new_stackvar(dt_ulong)(lambda (phys_addr : Address):
  e2s(assign(pm, dt_ulong)(id(phys_addr), phys)) ##
  receiver ##
  (lambda (receiver_addr : Address):
  phys_to_virt(receiver_addr, phys_addr)(ret_addr))))

% Function semantics for phys_to_ptr
% File "nova/include/buddy.h", line 97, character 9
phys_to_ptr(phys : Address)(ret_addr : Address) : ST[State] =
  return(pm, dt_pointer[void], ret_addr)(reinterpret_cast(dt_pointer[void],
                                                            dt_ulong)
  (call_phys_to_virt(id(allocator),
                    static_cast_nocv_unsigned_long_long_reference_to_nocv_unsigned_long_int
                    (id(phys)))))

call_phys_to_ptr(phys : ET[State, Semantics_ulonglong]) : ET[State,
  Semantics_pointer[void]] =
  with_new_returnvar(pm, dt_pointer[void])(lambda (ret_addr : Address) :
  with_new_stackvar(dt_ulonglong)(lambda (phys_addr : Address):
  e2s(assign(pm, dt_ulonglong)(id(phys_addr), phys)) ##
  phys_to_ptr(phys_addr)(ret_addr)))

% Function semantics for master
% File "nova/include/ptab.h", line 36, character 9
master(ret_addr : Address) : ST[State] =
  with_new_stackvar(lambda (master_l : Address):
  e2s(skip) ##
  return(pm, dt_Address, ret_addr)(id(master_l)))

call_master : ET[State, Semantics_pointer[Ptab]] =
  with_new_returnvar(pm, dt_pointer[Ptab])(lambda (ret_addr : Address) :
  master)

% Function semantics for sync_master
```

% File "nova/ptab.cpp", line 203, character 1

```

sync_master(receiver : Address, virt : Address)(ret_addr : Address) :
  ST[State] =
  with_new_stackvar(lambda (lev : Address):
    with_new_stackvar(lambda (pte : Address):
      with_new_stackvar(lambda (mst : Address):
        e2s(assign(pm, dt_uint)(id(lev), l2r(pm, dt_uint)(id(levels)))) ##
        e2s(skip ## skip) ##
        for(e2s(comma(reference[pointer[Ptab]])(assign(pm, dt_pointer[Ptab])
          (id(pte), id(receiver)),
          assign(pm, dt_pointer[Ptab])
          (id(mst), call_master))),

          n2b(literal(true)),
          comma(reference[pointer[Ptab]])(assign(pm, dt_pointer[Ptab])
            (id(pte),
              static_cast_nocv_void_nocv_pointer_to_nocv_Ptab_nocv_pointer
              (call_phys_to_ptr(
                call_addr(id(pte))))),
            assign(pm, dt_pointer[Ptab])
            (id(mst),
              static_cast_nocv_void_nocv_pointer_to_nocv_Ptab_nocv_pointer
              (call_phys_to_ptr(
                call_addr(id(mst))))))),

        with_new_stackvar(lambda (shift : Address):
          with_new_stackvar(lambda (slot : Address):
            with_new_stackvar(lambda (size : Address):
              e2s(assign(pm, dt_uint)(id(shift),
                plus(uint)(times(uint)
                  (predec(pm, reference[uint])(id(lev)),
                    id(bits_per_level)),
                    literal[State, Semantics_int](12)))) ##

              e2s(assign(pm, dt_uint)(id(slot),
                int_conv_nocv_unsigned_long_int_to_nocv_unsigned_int(
                  bitand(ulong)(shr(ulong)
                    (id(virt),
                      id(shift)),
                    minus(ulong)
                    (shl(ulong)
                      (literal[State, Semantics_ulong](1),
                        id(bits_per_level)),
                      literal[State, Semantics_int](1)))))) ##

              e2s(assign(pm, dt_uint)(id(size),
                int_conv_nocv_unsigned_long_int_to_nocv_unsigned_int(
                  shl(ulong)(literal[State, Semantics_ulong](1),
                    id(shift)))))) ##

              e2s(assign_plus(pm, pointer[Ptab])(id(mst),
                l2r(pm, dt(int))
                (id(slot)))) ##

              e2s(assign_plus(pm, pointer[Ptab])(id(pte),
                l2r(pm, dt(int))
                (id(slot)))) ##

            if_else(call_present(id(mst)),

```

```

        if_else(eq(bool))(id(slot),
                bitand(uint)(shr(uint)(literal[State, Semantics_uint](-807403520),
                        id(shift)),
                        minus(ulong)
                        (shl(ulong)
                        (literal[State, Semantics_ulong](1),
                        id(bits_per_level)),
                        literal[State, Semantics_int](1))))),
                skip ##
                continue,
                skip) ##
        e2s(call_operator_assign(id(pte), id(mst))) ##
        skip,
        skip) ##
        return(pm, dt_reference[uint], ret_addr)(id(size)))))) ##
return_void

call_sync_master(receiver : ET[State, Semantics_reference[Ptab]],
                virt : ET[State, Semantics_ulong]) : ET[State,
                Semantics_uint] =
with_new_returnvar(pm, dt_uint)(lambda (ret_addr : Address) :
with_new_stackvar(dt_ulong)(lambda (virt_addr : Address):
e2s(assign(pm, dt_ulong)(id(virt_addr), virt)) ##
receiver ##
(lambda (receiver_addr : Address):
sync_master(receiver_addr, virt_addr)(ret_addr))))

```

END ptab_sync_master

B.26 random_device.pvs

```

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Random Device
%
% base : base address of the device
% [base + rnd_seed] : Word (32 bit) - seed value % 2^32
% [base + mem_cnt] : Word - memory access count % 2^32 ;
% writing to mem_cnt clears the counter to 0
% [base + rnd_val] : Word - random value based on choose(seed x cnt x N);
% the value is random as long as states are
% initialised with differing seed values
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

```

```

Random_Device[Physical_memory : Type,
                (Importing Plain_Memory)
                pm : Plain_Memory[Physical_memory]] : Theory

```

Begin

Assuming

Importing Cpp_Typesdt_uint_size : **Assumption**0 < size(uidt(dt_uint)) **And** 3 * size(uidt(dt_uint)) <= max_linear_offset**EndAssuming**

% Device State

%-----

Random_device_state : **Type** = [#

seed : nat,

memory_access_count : nat,

counter_offset : {n : nat | n <= memory_access_count}

#]

% proof status :-)

% base_TCC1 : TCC Obligation

size : nat = 3 * size(uidt(dt_uint))

base : ({m : Memory_Address_4G | m + size <= max_linear})

% proof status :-)

% rnd_seed_TCC1 : TCC Obligation

% proof status :-)

% mem_cnt_TCC1 : TCC Obligation

% proof status :-)

% rnd_val_TCC1 : TCC Obligation

rnd_seed : Memory_Address_4G = base + 0 * size(uidt(dt_uint))

mem_cnt : Memory_Address_4G = base + 1 * size(uidt(dt_uint))

rnd_val : Memory_Address_4G = base + 2 * size(uidt(dt_uint))

Importing Device_Memory[Physical_memory, pm, Random_device_state]Random_device_memory : **Type** = Device_memorys : **Var** Random_device_memory

% Accessor functions

%-----

seed(s) : nat = s'expansion'seed

access_count(s) : nat = s'expansion'memory_access_count

counter_offset(s) : nat = s'expansion'counter_offset

% proof status :-)

% increase_access_count_TCC1 : TCC Obligation

increase_access_count(s)(n : nat) : Random_device_memory =

s **With** [(expansion) := s'expansion **With** [(memory_access_count) := access_count(s) + n]]

B PVS Theory Sources

```

% proof status :-)
% clear_counter_TCC1 : TCC Obligation

clear_counter(s) : Random_device_memory =
  s With [(expansion) := s'expansion With [(counter_offset) := access_count(s)]]

% Side Effects
%-----
access_count(a : Address, bl : list[Byte], cp : bool)(s) :
  ExprResult[Random_device_memory, list[Byte]] =
  OK(increase_access_count(s)(length(bl)), bl)

unaligned_access(a : Address, bl : list[Byte], cp : bool) :
  [Random_device_memory -> ExprResult[Random_device_memory, list[Byte]]] =
  If null?(bl) Or disjoint?(address_block(a, length(bl)),
    address_block(base, size)) Or
    % check aligned access
    (Not cp And length(bl) = size(uidt(dt_uint)) And
      (a = rnd_seed Or a = mem_cnt Or a = rnd_val)) Then
    ok_result(bl)
  Else
    fatal_result
  Endif

% proof status :-)
% random_TCC1 : TCC Obligation

random(seed : nat, count : nat) :
  {l : list[Byte] | length(l) = size(uidt(dt_uint))}

random(s : Random_device_memory) : list[Byte] =
  random(seed(s), access_count(s))

read_rnd_val(a : Address, bl : list[Byte], cp : bool) :
  [Random_device_memory -> ExprResult[Random_device_memory, list[Byte]]] =
  If cons?(bl) And a = rnd_val Then
    Lambda (s : Random_device_memory) : ok_result(random(s))(s)
  Else
    ok_result(bl)
  Endif

% proof status :-)
% read_mem_cnt_TCC1 : TCC Obligation

% proof status :-)
% read_mem_cnt_TCC2 : TCC Obligation

% proof status :-)
% read_mem_cnt_TCC3 : TCC Obligation

read_mem_cnt(a : Address, bl : list[Byte], cp : bool) :
  [Random_device_memory -> ExprResult[Random_device_memory, list[Byte]]] =

```

```

If cons?(bl) And a = mem_cnt Then
  Lambda (s : Random_device_memory) :
    ok_result(dt_uint'to_byte(
      rem(max_value(uint))(access_count(s) - counter_offset(s), mem_cnt))(s)
    )
Else
  ok_result(bl)
Endif

% proof status :-)
% read_seed_TCC1 : TCC Obligation

% proof status :-)
% read_seed_TCC2 : TCC Obligation

% proof status :-)
% read_seed_TCC3 : TCC Obligation

read_seed(a : Address, bl : list[Byte], cp : bool) :
  [Random_device_memory -> ExprResult[Random_device_memory, list[Byte]]] =
If cons?(bl) And a = rnd_seed Then
  Lambda (s : Random_device_memory) : ok_result(
    dt_uint'to_byte(rem(max_value(uint))(seed(s), rnd_seed))(s)
  )
Else
  ok_result(bl)
Endif

write_rnd_dev(a : Address, bl : list[Byte], cp : bool) :
  [Random_device_memory -> ExprResult[Random_device_memory, list[Byte]]] =
If cons?(bl) And (a = rnd_val Or a = rnd_seed) Then
  fatal_result
Else
  ok_result(bl)
Endif

reset_count(a : Address, bl : list[Byte], cp : bool)(s) :
  ExprResult[Random_device_memory, list[Byte]] =
If cons?(bl) And a = mem_cnt Then
  OK(clear_counter(s), bl)
Else
  ok_result(bl)(s)
Endif

% read side effects
rnd_dev_read_side_effect(a : Address, bl : list[Byte], cp : bool) :
  [Random_device_memory -> ExprResult[Random_device_memory, list[Byte]]] =
(
  unaligned_access(a, bl, cp) ## Lambda (bl : list[Byte]) :
  access_count(a, bl, cp) ## Lambda (bl : list[Byte]) :
  read_rnd_val(a, bl, cp) ## Lambda (bl : list[Byte]) :
  read_mem_cnt(a, bl, cp) ## Lambda (bl : list[Byte]) :
  read_seed(a, bl, cp)
)

```

```

rnd_dev_write_side_effect(a : Address, bl : list[Byte], cp : bool) :
  [Random_device_memory -> ExprResult[Random_device_memory, list[Byte]]] =
(
  unaligned_access(a, bl, cp) ## Lambda (bl : list[Byte]) :
  access_count(a, bl, cp) ## Lambda (bl : list[Byte]) :
  write_rnd_dev(a, bl, cp) ## Lambda (bl : list[Byte]) :
  reset_count(a, bl, cp)
)

% Plain Memory
%=====

% Memory Struct
%-----
random_device_pm_mem : Memory_struct[Random_device_memory] =
  device_pm(rnd_dev_read_side_effect, rnd_dev_write_side_effect)

% Random_Device PM Struct
%-----

random_device_pm : Plain_Memory[Random_device_memory] = (#
  mem := random_device_pm_mem,
  states := em_lift(pm'states),
  ro_addr := difference(pm'ro_addr, address_block(base, size)),
  rw_addr := difference(pm'rw_addr, address_block(base, size)),
  other_actions := em_lift(pm'other_actions)
#)

End Random_Device

Challenge_Random_Device[Physical_memory : Type,
  (Importing Plain_Memory)
  pm_phy : Plain_Memory[Physical_memory]] : Theory

Begin

Assuming

  Importing Cpp_uint, Extended_Real

  dt_uint_size : Assumption
    0 < size(uidt(dt_uint)) And 3 * size(uidt(dt_uint)) <= max_linear_offset

EndAssuming

% proof status :-)
% IMP_Random_Device_TCC1 : TCC Obligation

Importing Random_Device[Physical_memory, pm_phy],
  Challenge_Device_Memory[Physical_memory, pm_phy, Random_device_state]

% Side Effect Conditions

```

```

%-----

pm : Plain_Memory[Random_device_memory] = random_device_pm

% proof status :-)
in_blessed_memory_disjoint_device : Lemma
  Forall (a : Address, l : nat) :
    subset?(address_block(a, l), union(pm'ro_addr, pm'rw_addr))
      Implies
        disjoint?(address_block(a, l), address_block(base, size))

% proof status :-)
in_blessed_memory_not_device_address : Lemma
  Forall (a : Address, bl : list[Byte]) :
    subset?(address_block(a, length(bl)), union(pm'ro_addr, pm'rw_addr)) And cons?(bl)
      Implies
        a /= rnd_seed And a /= mem_cnt And a /= rnd_val

% proof status :-)
random_side_effects_unchanged_invariant : Lemma
  unchanged_memory_invariant?(pm'mem, pm'states, union(
    drse_super_transformers(union(pm'ro_addr, pm'rw_addr),
      rnd_dev_read_side_effect),
    dwse_super_transformers(pm'rw_addr, rnd_dev_write_side_effect)),
    union(pm'ro_addr, pm'rw_addr))

% proof status :-)
random_side_effects_transformers_ok : Lemma
  transformers_ok?(pm'states, union(
    drse_super_transformers(union(pm'ro_addr, pm'rw_addr),
      rnd_dev_read_side_effect),
    dwse_super_transformers(pm'rw_addr, rnd_dev_write_side_effect)))

% proof status :-)
random_side_effects_side_effect_content_unchanged_read : Lemma
  side_effect_content_unchanged(union(pm'ro_addr, pm'rw_addr), pm'states,
    rnd_dev_read_side_effect)

% proof status :-)
random_side_effects_side_effect_content_unchanged_write : Lemma
  side_effect_content_unchanged(pm'rw_addr, pm'states,
    rnd_dev_write_side_effect)

% Plain Memory
%=====
% proof status :-)
random_device_plain_memory : Lemma
  plain_memory?(pm_phy) Implies
    plain_memory?(pm)

% Fail on Unaligned Accesses
%=====

```

B PVS Theory Sources

```
% proof status :-)
unaligned_fails_write : Lemma
  Forall (s : Random_device_memory, a : Address, bl : (cons?[Byte])) :
    Not disjoint?(address_block(a, length(bl)), address_block(base, size)) And
      Not (length(bl) = size(uidt(dt_uint))) And (a = rnd_seed Or a = mem_cnt Or a = rnd_val)) And
        % underlying write side effect is executed first ; require it to be ok
        OK?(memory_write_side_effect(pm_phy'mem)(a, bl, false)(s'state)) And
          data(memory_write_side_effect(pm_phy'mem)(a, bl, false)(s'state)) = bl
    Implies
      Fatal?(memory_write_list(pm'mem)(a, bl)(s))

% proof status :-)
unaligned_fails_read : Lemma
  Forall (s : Random_device_memory, a : Address, bl : (cons?[Byte])) :
    Not disjoint?(address_block(a, length(bl)), address_block(base, size)) And
      Not (length(bl) = size(uidt(dt_uint))) And (a = rnd_seed Or a = mem_cnt Or a = rnd_val)) And
        % underlying read list and read side effect is executed first ; require it to be ok
        OK?(memory_read_list_nse(pm'mem)(a, length(bl))(s)) And
          (Let ures = (memory_read_list_nse(pm'mem)(a, length(bl)) ##
            LAMBDA (bl1: list[Byte]): em_lift[Physical_memory,
              Random_device_state[Physical_memory, pm_phy], list[Byte]]
              (memory_read_side_effect(pm_phy'mem)
                (a, bl1, false))(s) In
                OK?(ures) And length(data(ures)) = length(bl))
          Implies
            Fatal?(memory_read_list(pm'mem)(a, length(bl))(s))

End Challenge_Random_Device
```

B.27 result.pvs

```
% $Id: result.pvs,v 1.32.10.2 2008/05/14 14:25:00 tews Exp $
%
% Author: Hendrik Tews
% Author: Marcus Voelp
% Author: Tjark Weber, (c) 2007 Radboud University
%
% Result types for C++ expressions and statements.

% TODO: The Pagefault/Exception related definitions need cleanup when it is
% clear what exactly we want to model (and at which level). Perhaps
% they could be moved to a different file also.
```

```
Memory_access: DATATYPE
BEGIN
  Read : Read?
  Write : Write?
  Execute : Execute?
END Memory_access
```

*% This differs from CPU privilege levels in that there are only two.
 % We assume that CPL 1 and CPL 2 are never ever used.*

Memory_privilege : **DATATYPE**

BEGIN

Supervisor : Supervisor?

User : User?

END Memory_privilege

Memory_access_util : **Theory**

% utility function for Memory_access

Begin

Importing Memory_access

RX_Memory_access : **Type** = {p : pred[Memory_access] | subset?(add(Execute, singleton(Read)), p)}

R_Memory_access : **Type** = {p : pred[Memory_access] | member(Read, p)}

memory_access_to_bools(ar : PRED[Memory_access]) : [bool, bool] =
 (member(Write, ar), member(Execute, ar))

bools_to_memory_access(write : bool, fetch : bool) : PRED[Memory_access] =

If write **Then**

If fetch **Then**

add(Execute, add(Write, singleton(Read)))

Else

add(Write, singleton(Read))

Endif

Else

If fetch **Then**

add(Execute, singleton(Read))

Else

singleton(Read)

Endif

Endif

End Memory_access_util

Memory_privilege_util : **Theory**

% utility functions for Memory_privilege

Begin

Importing Memory_privilege

memory_privilege_to_bool(ar : Memory_privilege) : bool =

Cases ar **of**

Supervisor : **false**,

User : **true**

EndCases

bool_to_memory_privilege(b : bool) : Memory_privilege =

IF b **Then** User **Else** Supervisor **Endif**

B PVS Theory Sources

```
% proof status :-)
bool_to_memory_privilege_iso : Lemma Forall(ar : Memory_privilege) :
  bool_to_memory_privilege(memory_privilege_to_bool(ar)) = ar
```

End Memory_privilege_util

Error_Code_Types : **Theory**

Begin

Importing Memory_privilege, Memory_access_util

```
PF_error_code_type : Type = [#
  reserved_bit_violation : bool, % page fault not modeled
  privilege : Memory_privilege,
  access : R_Memory_access,
  present : bool
#]
```

End Error_Code_Types

Exception_type : **Datatype**

Begin

Importing Error_Code_Types

```
% Exceptions which may possibly occur in the kernel and which are
% smoothly handled. If a not listed exceptions occurs in kernel code
% this is a fatal error. The following lists when these exceptions
% occur within the kernel:
```

```
% – the kernel copies from / to user memory
% – sync with master page tables
Page_fault(ec : PF_error_code_type) : Page_fault?
```

```
% – lazy segment loads
General_protection : General_protection? %
```

End Exception_type

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% ExprResult: the result type for C++ expressions
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

ExprResult[State, Data : **Type**] : **DATATYPE**

BEGIN

```
%% IMPORTING Pagefault_Flag
IMPORTING Exception_type
```

```
% result OK with successor state and result data
OK(state: State, data: Data) : OK?
```

```
% not (yet) delivered exception or interrupt
% Before the exception handler can be invoked the exception must be
% delivered. At delivery the IDT is traversed and the conditions for
% double faults need to be checked and the stack frame must be
% constructed.
Exception(
  ex_type : Exception_type,
  state : State % state during which exception occurred
) : Exception?
```

```
%% % not (yet) handled page fault
%% Page_fault(
%% pfa : Address,
%% page_fault_flags : Pagefault_flag) : Page_fault?
%%
%% % TLB inconsistent with current page directory
%% TLB_fault : TLB_fault?
%%
%% % general protection fault (in case somebody wants to handle those)
%% % XXX should all exceptions get a result?
%% GP_fault : GP_fault?
%%
%% % return from interrupt
%% % Attention: this result must be caught immediately. It must not
%% % be passed through other state transformers!
%% Iret(next_state : State) : Iret?
%%
```

```
% worst case crash (undefined behavior)
Fatal : Fatal?
```

```
% a non-terminating program statement has been encountered
Hang : Hang?
```

```
END ExprResult
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% StmtResult: the result type for C++ statements
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
StmtResult[State : Type] : DATATYPE
```

```
BEGIN
```

```
  Importing Exception_type
```

```
  % result OK with successor state
```


B PVS Theory Sources

```
% Note that this, unlike the "ExprResult.OK" constructor, does not have a
% "data" argument. (If it had, we would have to invent a term of type "Data"
% at some places in the C++ program, e.g. at the beginning of a function, or
% after catching a break/continue.)
OK(state : State) : OK?

% "break" has been executed
Break(state : State) : Break?

% "continue" has been executed
Continue(state : State) : Continue?

% "return" has been executed
% We model return expr as expr ## lambda (res : Data) : write(ret_addr, data)
% where ret_addr can be an unused region of the stack frame suitable
% to hold the returned type or a register of suitable size.
% This model includes returning classes by allocating them in a
% caller allocated memory and returning integral types in the eax register as
% specified in the Unix System V ABI for Intel386 Processors.
Return(state : State) : Return?

% "goto label" has been executed
% The formalization of goto statements, although possible, would introduce
% two complications:
% 1. C++ allows gotos into nested substatements. Thus every statement
% potentially catches a Goto result.
% 2. Since backwards jumps may lead to non-termination, gotos need to be
% treated similar to loops.
Goto(state : State, label : string) : Goto?

% "switch (case)" has been executed, and we're looking for a matching
% "case"
% Note that "case" cannot have type "Data", as that type is already reserved
% for the result of the function in which the switch statement occurs. Thus
% we're using the fixed type "int". Alternatively, StmtResult could take
% another type argument "SwitchData : Type".
Switch(state : State, case : int) : Switch?

% a switch statement has been executed, and we already know that there is no
% matching "case" (hence the "default" must be taken if there is one)
Default(state : State) : Default?

% Not (yet) delivered exception or interrupt, see also ExprResult.
Exception(
  ex_type : Exception_type,
  state : State % state during which exception occurred
) : Exception?

% worst case crash (undefined behavior)
Fatal : Fatal?

% a non-terminating program statement has been encountered
```

Hang : Hang?

END StmtResult

*% The following is a reduced super type of ExprResult and StmtResult.
 % Transformer invariants and other general stuff, eg transformer_ok?,
 % are formulated with SuperResult. Expression transformers and statement
 % transformers can be cast into a super transformer in the obvious way:
 %
 % OK → OK
 % abnormalities with successor state → Abnormal
 % everything else → Bottom
 %
 % See Super_Embedding below.*

SuperResult[State : Type] : DATATYPE

Begin

OK(state : State) : OK?

Abnormal(state : State) : abnormal?

Bottom : bottom?

End SuperResult

Super_Result_Util[State : Type] : Theory

Begin

Importing SuperResult

has_next_state(res : SuperResult[State]) : bool =

Cases res **OF**

OK(state) : **true**,

Abnormal(state) : **true**,

Bottom : **false**

EndCases

End Super_Result_Util

Super_Embedding_Expr[State, Data : Type] : Theory

Begin

Importing ExprResult, Super_Result_Util

has_next_state(expr : ExprResult[State, Data]) : bool =

Cases expr **OF**

OK(state, date) : **true**,

Exception(ex_type, state) : **true**,

Fatal : **false**,

Hang : **false**

EndCases

% proof status :—)

% expr_2_super_res_TCC1 : TCC Obligation

```
% proof status :—)
% expr_2_super_res_TCC2 : TCC Obligation

expr_2_super_res(expr : ExprResult[State, Data]) : SuperResult[State] =
  IF OK?(expr) Then OK(state(expr))
  Elif has_next_state(expr) Then Abnormal(state(expr))
  Else Bottom
  Endif

End Super_Embedding_Expr

Super_Embedding_Stmt[State : Type] : Theory
Begin
  Importing StmtResult, Super_Result_Util

  has_next_state(stmt : StmtResult[State]) : bool =
    Cases stmt OF
      OK(state) : true,
      Break(state) : true,
      Continue(state) : true,
      Return(state) : true,
      Switch(state, case) : true,
      Default(state) : true,
      Exception(ex_type, state) : true,
      Fatal : false,
      Hang : false
    EndCases

  % proof status :—)
  % stmt_2_super_res_TCC1 : TCC Obligation

  % proof status :—)
  % stmt_2_super_res_TCC2 : TCC Obligation

  stmt_2_super_res(stmt : StmtResult[State]) : SuperResult[State] =
    IF OK?(stmt) Then OK(state(stmt))
    Elif has_next_state(stmt) Then Abnormal(state(stmt))
    Else Bottom
    Endif

End Super_Embedding_Stmt

Super_Embedding : Theory
Begin
  Importing Super_Embedding_Stmt, Super_Embedding_Expr

End Super_Embedding
```

B.28 search-example.pvs

```
% Additional_Rewrites[State, Data : Type] : Theory
```

```

% Begin

% address_of_q
% check_bounds_q
% same_array_left_q
% same_array_right_q
% not_null_q

% End Additional_Rewrites

Search_Example[State : Type] : Theory
  Begin

    IMPORTING Cpp_Verification[State]

    % Variable and Constant Definition
    %=====

    first : Address % pointer to first array element
    last  : Address % pointer to last array element

    current : Address % pointer to current array element

    N : posnat = 2 % array size

    % proof status :-)
    % value_TCC1 : TCC Obligation

    value : (range(uint)) % value to search for

    % Preconditions
    %=====

    % proof status :-)
    % Search_Pre_TCC1 : TCC Obligation

    % proof status :-)
    % Search_Pre_TCC2 : TCC Obligation

    % proof status :-)
    % Search_Pre_TCC3 : TCC Obligation

    % proof status :-)
    % Search_Pre_TCC4 : TCC Obligation

    % proof status :-)
    % Search_Pre_TCC5 : TCC Obligation

    % proof status :-)
    % Search_Pre_TCC6 : TCC Obligation

    % proof status :-)

```

B PVS Theory Sources

```

% Search_Pre_TCC7 : TCC Obligation

% proof status :-)
% Search_Pre_TCC8 : TCC Obligation

% proof status :-)
% Search_Pre_TCC9 : TCC Obligation

Search_Pre : PRED[State] =
  Lambda (s : State) :
    plain_memory?(pm) And
    pm'states(s) And
    % Parameters
    % 1. first and last are allocated in blessed memory
    in_blessed_memory?(dt_cv_pointer(pointer(uint)), first,
                        union(pm'ro_addr, pm'rw_addr)) And
    in_blessed_memory?(dt_cv_pointer(pointer(uint)), last,
                        union(pm'ro_addr, pm'rw_addr)) And
    % 2. the memory locations at first and last store valid pointer values
    valid_in_mem(pm, dt_cv_pointer(pointer(uint)))(first)(s) And
    valid_in_mem(pm, dt_cv_pointer(pointer(uint)))(last)(s) And
    % Local Variables
    % 3. current is allocated in blessed writable memory
    in_blessed_memory?(dt_cv_pointer(pointer(uint)),
                        current, pm'rw_addr) And
    % 4. current is disjoint from first
    blocks_disjoint?(first, size_of(pointer(uint)),
                       current, size_of(pointer(uint))) And
    blocks_disjoint?(current, size_of(pointer(uint)),
                       first, size_of(pointer(uint))) And
    % 5. current is disjoint from last
    blocks_disjoint?(last, size_of(pointer(uint)),
                       current, size_of(pointer(uint))) And
    blocks_disjoint?(current, size_of(pointer(uint)),
                       last, size_of(pointer(uint))) And
    % Assertions
    Let first_value =
      data(read_data(pm, dt_cv_pointer(pointer(uint)))(first)(s)),
      last_value =
      data(read_data(pm, dt_cv_pointer(pointer(uint)))(last)(s)) In
    % 6. first and last are non null
    not_null?(pointer(uint))(first_value) And
    not_null?(pointer(uint))(last_value) And
    % Array
    % 7. array elements are in blessed memory
    (Forall (i : below[N]) :
      in_blessed_memory?[(range(uint))](dt(uint),
      down(address_of(pointer(uint))
            (add(pointer(uint))(first_value, i * size_of(uint))))),
      union(pm'ro_addr, pm'rw_addr)) And
    % 8. array elements are disjoint from current
    (Forall (i : below[N]) :

```

```

    blocks_disjoint?(
      down(address_of(pointer(uint))
            (add(pointer(uint))(first_value, i * size_of(uint)))),
      size_of(uint),
      current,
      size_of(pointer(uint)))) And
  (Forall (i : below[N]) :
    blocks_disjoint?(
      current, size_of(pointer(uint)),
      down(address_of(pointer(uint))
            (add(pointer(uint))(first_value, i * size_of(uint)))),
      size_of(uint)) And
    % 9. array elements contain valid integers
    (Forall (i : below[N]) :
      valid_in_mem(pm, dt(uint))
      (down(address_of(pointer(uint))
            (add(pointer(uint))(first_value, i * size_of(uint)))))(s))
    And
    % 10. array bounds
    data(read_data(pm, dt_cv_pointer(pointer(uint)))(last)(s) =
      add(pointer(uint))(first_value, N * size_of(uint)) And
    % 11. same array
    same_array(pointer(uint))(first_value, last_value) And
    (Forall (i : below[N]) :
      same_array(pointer(uint))(first_value,
        add(pointer(uint))(first_value, i * size_of(uint))))

% /**
% * search the array element whose value is val
% *
% * @param val the value to search for
% * @param first pointer to the first element of the array (from which
% * to start the search)
% * @param last pointer beyond the last element of the array
% * @return pointer to the first occurrence of val after (including) first
% */
% uint * search(uint val, uint * first, uint * last) {
% assert(first);
% assert(last);

% 1) uint * current = first;

% 2) while(current < last) {
% 3) if (*current == val)
% 4) break;

% 5) ++current;
% }

% proof status :-)
% search_TCC1 : TCC Obligation

```

B PVS Theory Sources

```

% proof status :-)
% search_TCC2 : TCC Obligation

% proof status :-)
% search_TCC3 : TCC Obligation

% proof status :-)
% search_TCC4 : TCC Obligation

search : ST[State] =
  % 1)
  e2s[State, Address]
    (assign(pm, dt_cv_pointer(pointer(uint)))
      (id(current),
        l2r(pm, dt_cv_pointer(pointer(uint)))(id(first)))) ##
  % 2)
  while(N + 1,
    lt_ptr(pointer(uint))
      (l2r(pm, dt_cv_pointer(pointer(uint)))(id(current)),
        l2r(pm, dt_cv_pointer(pointer(uint)))(id(last))),
  % 3)
    if_else(
      eq(l2r(pm, dt(uint))(deref(pm, pointer(uint)))
        (l2r(pm, dt_cv_pointer(pointer(uint)))(id(current))))),
      literal(value)),
  % 4)
    break,
    skip) ###
  % 5)
  e2s[State, Address](preinc_ptr(pm, pointer(uint))(id(current)))

% Postconditions:
%=====

% read first after search
read_first : PRED[State] =
  Lambda (s : State) :
    OK?(read_data(pm, dt_cv_pointer(pointer(uint)))(first)(s))

% read current after search
read_current : PRED[State] =
  Lambda (s : State) :
    OK?(read_data(pm, dt_cv_pointer(pointer(uint)))(current)(s)) And
    not_null?(pointer(uint))(data(read_data(pm, dt_cv_pointer(pointer(uint)))(current)(s)))

% proof status :-)
% deref_current_TCC1 : TCC Obligation

% proof status :-)
% deref_current_TCC2 : TCC Obligation

% read current after search

```

```

deref_current : PRED[State] =
  Lambda (s : State) :
    read_current(s) And
    Let current_value =
      data(read_data(pm, dt_cv_pointer(pointer(uint)))(current)(s)) In
      not_null?(pointer(uint))(current_value) And
      OK?(read_data(pm, dt(uint))
          (down(address_of(pointer(uint))(current_value)))(s))

% read last after search
read_last : PRED[State] =
  Lambda (s : State) :
    OK?(read_data(pm, dt_cv_pointer(pointer(uint)))(last)(s))

% proof status :-)
% read_array_TCC1 : TCC Obligation

% proof status :-)
% read_array_TCC2 : TCC Obligation

% proof status :-)
% read_array_TCC3 : TCC Obligation

% read array after search
read_array : PRED[State] =
  Lambda (s : State) :
    read_first(s) And
    Let first_value =
      data(read_data(pm, dt_cv_pointer(pointer(uint)))(first)(s)) In
      not_null?(pointer(uint))(first_value)
    And
    Forall (i : below[N]) :
      OK?(read_data(pm, dt(uint))
          (down(address_of(pointer(uint))
              (add(pointer(uint))(first_value, i * size_of(uint)))))(s))

% proof status :-)
% not_found_TCC1 : TCC Obligation

% proof status :-)
% not_found_TCC2 : TCC Obligation

% proof status :-)
% not_found_TCC3 : TCC Obligation

% proof status :-)
% not_found_TCC4 : TCC Obligation

% not found:
% the value could not be found in the array; current = last
not_found : PRED[State] =
  Lambda (s : State) :

```



```

read_first(s) And
read_current(s) And
read_last(s) And
read_array(s) And
(Let first_value =
  data(read_data(pm, dt_cv_pointer(pointer(uint)))(first)(s)) In
  Forall (i : below[N]) :
    (data(read_data(pm, dt(uint))
      (down(address_of(pointer(uint))(add(pointer(uint))
        (first_value, i * size_of(uint))))(s)) /= value))

Implies
  data(read_data(pm, dt_cv_pointer(pointer(uint)))(last)(s)) =
    data(read_data(pm, dt_cv_pointer(pointer(uint)))(current)(s))

% proof status :- )
% found_TCC1 : TCC Obligation

% proof status :- )
% found_TCC2 : TCC Obligation

% proof status :- )
% found_TCC3 : TCC Obligation

found : PRED[State] =
Lambda (s : State) :
  (read_first(s) And
  read_current(s) And
  read_last(s) And
  read_array(s) And
  deref_current(s) And
  data(read_data(pm, dt_cv_pointer(pointer(uint)))
    (last)(s)) /=
    data(read_data(pm, dt_cv_pointer(pointer(uint)))(current)(s)))
Implies
  data(read_data(pm, dt(uint))
    (down(address_of(pointer(uint))
      (data(read_data(pm, dt_cv_pointer(pointer(uint)))(current)(s))))(s))
    = value

% Lemmas
%=====

% proof status :- )
% uchar_range_values_TCC1 : TCC Obligation

% proof status :- )
uchar_range_values : Lemma
  Forall (n : nat) : n <= 255 Implies range(uchar)(n)

% proof status :- )
search_terminates : Lemma

```

```

Valid(
  Search_Pre,
  search,
  Lambda (s : State) : true)

% proof status :-)
search_read_first : Lemma
  valid(
    Search_Pre,
    search,
    read_first)

% proof status :-)
search_read_current : Lemma
  valid(
    Search_Pre,
    search,
    read_current)

% proof status :-)
search_deref_current : Lemma
  valid(
    Search_Pre,
    search,
    deref_current)

% proof status :-)
search_read_last : Lemma
  valid(
    Search_Pre,
    search,
    read_last)

% proof status :-)
search_read_array : Lemma
  valid(
    Search_Pre,
    search,
    read_array)

% proof status :-)
search_not_found : Lemma
  valid(
    Search_Pre,
    search,
    not_found)

% proof status :-)
search_found : Lemma
  valid(
    Search_Pre,
    search,

```

found)

End Search_Example

B.29 statement-rewrites.pvs

```

% $Id: statement-rewrites.pvs,v 1.10.4.1.4.2 2008/05/14 14:25:00 tewz Exp $
%
% Author: Marcus Völp
%
% Rewriting rules for C++ statements. See statements.pvs for the statement
% formalization.
%
%
% The statement rewriting rules simplify code by symbolic
% simplification of the control flow constructs. For example a break will
% remove all following statements until it reaches a catch break in
% which case break ## catch_break is eliminated from the code.
% Statements are simplified right to left. Here one expression
% transformer respectively one lexpr transformer (StmtResult ->
% ExprResult) is allowed to occur after the simplification point. The
% forward rules simplify these two cases plus those cases with a leading
% statement automatically with the help of the respective rules for the
% standalone case. Because of the requirement on rewriting rules these
% lemmas have to be defined for any form a standalone simplification may
% assume. The forms are denoted by the s-c->s, ... sequence which stands
% for statement1 ## complex statement simplifies to statement1.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

Single_Statement_Rewrites0[State : Type] : THEORY
BEGIN

  Importing Statements

  stmt : VAR [State -> StmtResult[State]]
  stmt1 : VAR [State -> StmtResult[State]]
  stmt2 : VAR [State -> StmtResult[State]]
  stmt3 : VAR [State -> StmtResult[State]]
  cstmt : VAR [StmtResult[State] -> StmtResult[State]]
  cstmt1 : VAR [StmtResult[State] -> StmtResult[State]]
  fcstmt : VAR [[State -> StmtResult[State]] ->
                [StmtResult[State] -> StmtResult[State]]]

  s : VAR State
  n : VAR int

  % Switch / Default / Continue / Return elimination
  %=====

  % Break

```

```

%-----

% proof status :-)
break_catch_break : Lemma
  OK?(stmt(s)) Implies
    (stmt ## break ## catch_break)(s) = stmt(s)

% proof status :-)
break_break_stmt : Lemma
  Break?(stmt(s)) Implies
    (stmt ## stmt1 ## catch_break)(s) = (stmt ## catch_break)(s)

% proof status :-)
break_break_catch_continue : Lemma
  Break?(stmt(s)) Implies
    (stmt ## catch_continue ## catch_break)(s) = (stmt ## catch_break)(s)

% proof status :-)
break_break_catch_default : Lemma
  Break?(stmt(s)) Implies
    (stmt ## catch_default ## catch_break)(s) = (stmt ## catch_break)(s)

% proof status :-)
break_break_lift_stmt : Lemma
  Break?(stmt(s)) Implies
    (stmt ## lift(stmt1) ## catch_break)(s) = (stmt ## catch_break)(s)

% proof status :-)
break_break_case : Lemma
  Break?(stmt(s)) Implies
    (stmt ## case(n) ## catch_break)(s) = (stmt ## catch_break)(s)

% proof status :-)
break_break_default : Lemma
  Break?(stmt(s)) Implies
    (stmt ## default ## catch_break)(s) = (stmt ## catch_break)(s)

% proof status :-)
break_break : Lemma
  Break?(break(s))

% proof status :-)
stmt_break_break : Lemma
  OK?(stmt(s)) Implies
    Break?((stmt ## break)(s))

% proof status :-)
break_stmt_break : Lemma
  Break?(stmt(s)) Implies
    Break?((stmt ## stmt1)(s))

% proof status :-)

```

```

break_stmt_catch_continue : Lemma
  Break?(stmt(s)) Implies
    Break?((stmt ## catch_continue)(s))

% proof status :- )
break_stmt_catch_default : Lemma
  Break?(stmt(s)) Implies
    Break?((stmt ## catch_default)(s))

% proof status :- )
break_stmt_lift : Lemma
  Break?(stmt(s)) Implies
    Break?((stmt ## lift(stmt1))(s))

% proof status :- )
break_stmt_case : Lemma
  Break?(stmt(s)) Implies
    Break?((stmt ## case(n))(s))

% proof status :- )
break_stmt_default : Lemma
  Break?(stmt(s)) Implies
    Break?((stmt ## default)(s))

% Continue
%-----
% proof status :- )
continue_catch_continue : Lemma
  OK?(stmt(s)) Implies
    (stmt ## continue ## catch_continue)(s) = stmt(s)

% proof status :- )
continue_break_stmt : Lemma
  Continue?(stmt(s)) Implies
    (stmt ## stmt1 ## catch_continue)(s) = (stmt ## catch_continue)(s)

% proof status :- )
continue_break_catch_continue : Lemma
  Continue?(stmt(s)) Implies
    (stmt ## catch_break ## catch_continue)(s) = (stmt ## catch_continue)(s)

% proof status :- )
continue_break_catch_default : Lemma
  Continue?(stmt(s)) Implies
    (stmt ## catch_default ## catch_continue)(s) = (stmt ## catch_continue)(s)

% proof status :- )
continue_break_lift_stmt : Lemma
  Continue?(stmt(s)) Implies
    (stmt ## lift(stmt1) ## catch_continue)(s) = (stmt ## catch_continue)(s)

% proof status :- )

```

```

continue_break_case : Lemma
  Continue?(stmt(s)) Implies
    (stmt ## case(n) ## catch_continue)(s) = (stmt ## catch_continue)(s)

% proof status :-)
continue_break_default : Lemma
  Continue?(stmt(s)) Implies
    (stmt ## default ## catch_continue)(s) = (stmt ## catch_continue)(s)

% proof status :-)
continue_continue : Lemma
  Continue?(continue(s))

% proof status :-)
stmt_continue_continue : Lemma
  OK?(stmt(s)) Implies
    Continue?((stmt ## continue)(s))

% proof status :-)
continue_stmt_continue : Lemma
  Continue?(stmt(s)) Implies
    Continue?((stmt ## stmt1)(s))

% proof status :-)
continue_stmt_catch_break : Lemma
  Continue?(stmt(s)) Implies
    Continue?((stmt ## catch_break)(s))

% proof status :-)
continue_stmt_catch_default : Lemma
  Continue?(stmt(s)) Implies
    Continue?((stmt ## catch_default)(s))

% proof status :-)
continue_stmt_lift : Lemma
  Continue?(stmt(s)) Implies
    Continue?((stmt ## lift(stmt1))(s))

% proof status :-)
continue_stmt_case : Lemma
  Continue?(stmt(s)) Implies
    Continue?((stmt ## case(n))(s))

% proof status :-)
continue_stmt_default : Lemma
  Continue?(stmt(s)) Implies
    Continue?((stmt ## default)(s))

% switch / default are free-standing ; use fwd rewriting
%=====

% switch

```

B PVS Theory Sources

```
%-----  
% proof status :-)  
switch_stmt : Lemma  
  switch_result(n) ## stmt = switch_result(n)  
  
% proof status :-)  
stmt_switch_stmt : Lemma  
  stmt ## switch_result(n) ## stmt1 = stmt ## switch_result(n)  
  
% proof status :-)  
switch_case_taken : Lemma  
  (switch_result[State](n) ## case(n)) = skip  
  
% proof status :-)  
stmt_switch_case_taken : Lemma  
  OK?(stmt(s)) Implies  
  (stmt ## switch_result(n) ## case(n))(s) = stmt(s)  
  
% proof status :-)  
switch_case_not_taken : Lemma  
  Forall (n1, n2 : int) : n1 /= n2 Implies  
  switch_result[State](n1) ## case(n2) = switch_result(n1)  
  
% proof status :-)  
stmt_switch_case_not_taken : Lemma  
  Forall (n1, n2 : int) : OK?(stmt(s)) And n1 /= n2 Implies  
  (stmt ## switch_result(n1) ## case(n2))(s) = (stmt ## switch_result(n1))(s)  
  
% proof status :-)  
switch_default : Lemma  
  switch_result[State](n) ## default = switch_result(n)  
  
% proof status :-)  
stmt_switch_default : Lemma  
  OK?(stmt(s)) Implies  
  (stmt ## switch_result(n) ## default)(s) = (stmt ## switch_result(n))(s)  
  
% proof status :-)  
switch_catch_break : Lemma  
  switch_result[State](n) ## catch_break = switch_result(n)  
  
% proof status :-)  
stmt_switch_catch_break : Lemma  
  OK?(stmt(s)) Implies  
  (stmt ## switch_result(n) ## catch_break)(s) = (stmt ## switch_result(n))(s)  
  
% proof status :-)  
switch_catch_continue : Lemma  
  switch_result[State](n) ## catch_continue = switch_result(n)  
  
% proof status :-)  
stmt_switch_catch_continue : Lemma
```

```

OK?(stmt(s)) Implies
  (stmt ## switch_result(n) ## catch_continue)(s) = (stmt ## switch_result(n))(s)

% proof status :-)
switch_catch_default : Lemma
  switch_result[State](n) ## catch_default = switch_result(n)

% proof status :-)
stmt_switch_catch_default : Lemma
  OK?(stmt(s)) Implies
    (stmt ## switch_result(n) ## catch_default)(s) = (stmt ## switch_result(n))(s)

% default
%-----
% proof status :-)
default_stmt : Lemma
  default_result ## stmt = default_result

% proof status :-)
stmt_default_stmt : Lemma
  stmt ## default_result ## stmt1 = stmt ## default_result

% proof status :-)
default_case : Lemma
  default_result[State] ## case(n) = default_result

% proof status :-)
stmt_default_case : Lemma
  OK?(stmt(s)) Implies
    (stmt ## default_result ## case(n))(s) = (stmt ## default_result)(s)

% proof status :-)
default_default : Lemma
  default_result[State] ## default = skip

% proof status :-)
stmt_default_default : Lemma
  OK?(stmt(s)) Implies
    (stmt ## default_result ## default)(s) = stmt(s)

% proof status :-)
default_catch_break : Lemma
  default_result[State] ## catch_break = default_result

% proof status :-)
stmt_default_catch_break : Lemma
  OK?(stmt(s)) Implies
    (stmt ## default_result ## catch_break)(s) = (stmt ## default_result)(s)

% proof status :-)
default_catch_continue : Lemma
  default_result[State] ## catch_continue = default_result

```


B PVS Theory Sources

```
% proof status :-)
stmt_default_catch_continue : Lemma
  OK?(stmt(s)) Implies
  (stmt ## default_result ## catch_continue)(s) = (stmt ## default_result)(s)

% proof status :-)
default_catch_default : Lemma
  default_result[State] ## catch_default = skip

% proof status :-)
stmt_default_catch_default : Lemma
  OK?(stmt(s)) Implies
  (stmt ## default_result ## catch_default)(s) = stmt(s)

% Return rewrites
%=====

% proof status :-)
return_void_result : Lemma
  return_void[State] = return_result

% Skip Elimination
%=====

% proof status :-)
skip_ok : Lemma
  OK?(skip(s))

% proof status :-)
% skip_state_TCC1 : TCC Obligation

% proof status :-)
skip_state : Lemma
  state(skip(s)) = s

% proof status :-)
skip_elimination : Lemma
  (stmt ## skip) = stmt

% proof status :-)
stmt_skip_elimination : Lemma
  (skip ## stmt) = stmt

% Stmt Ok Elimination
%=====

% proof status :-)
stmt_remove_catch_default : Lemma
  OK?(stmt(s)) Implies
  (stmt ## catch_default)(s) = stmt(s)
```

```

% proof status :-)
stmt_remove_case : Lemma
  OK?(stmt1(s)) Implies
    (stmt1 ## case(n))(s) = stmt1(s)

% proof status :-)
stmt_remove_default : Lemma
  OK?(stmt1(s)) Implies
    (stmt1 ## default)(s) = stmt1(s)

% proof status :-)
stmt_remove_catch_continue : Lemma
  OK?(stmt(s)) Implies
    (stmt ## catch_continue)(s) = stmt(s)

% proof status :-)
stmt_remove_catch_break : Lemma
  OK?(stmt(s)) Implies
    (stmt ## catch_break)(s) = stmt(s)

% proof status :-)
stmt_ok_lift : Lemma
  OK?(stmt1(s)) Implies
    (stmt1 ## lift(stmt2))(s) = (stmt1 ## stmt2)(s)

% While
%=====

b_ex : VAR [State -> ExprResult[State, bool]]
max : Var nat

% proof status :-)
% while_unroll_TCC1 : TCC Obligation

% proof status :-)
while_unroll : Lemma
  max > 0 Implies
    while(max, b_ex, stmt)(s) =
      if_else(b_ex, stmt ## catch_continue ##
        while(max - 1, b_ex, stmt) ## catch_break, skip)(s)

% proof status :-)
stmt_while_unroll : Lemma
  max > 0 Implies
    (stmt1 ## while(max, b_ex, stmt2))(s) =
      (stmt1 ## if_else(b_ex, stmt2 ## catch_continue ##
        while(max - 1, b_ex, stmt2) ## catch_break, skip))(s)

% Condition universally true in all states implies hang of while
% proof status :-)
iterate_while_ok : Lemma

```

```

Forall (n : nat) :
  (Forall (s : State) : OK?((e2s(b_ex) ## stmt)(s)))
Implies
  OK?(iterate_while(n, b_ex, stmt)(s))

% proof status :-)
while_hang : Lemma
  (Forall (s : State) :
    OK?(b_ex(s)) And data(b_ex(s)) And
    OK?((e2s(b_ex) ## stmt)(s)))
Implies
  while(b_ex, stmt) = hang_result

% Composition
%=====

% proof status :-)
composition_assoc_rewrite_stmt_ok : Lemma
  OK?(stmt(s)) Implies
  (stmt ## (stmt1 ## cstmt))(s) = ((stmt ## stmt1) ## cstmt)(s)

% proof status :-)
composition_assoc_rewrite_stmt_ok_cstmt : Lemma
  OK?(stmt(s)) Implies
  ((stmt ## (stmt1 ## cstmt)) ## cstmt1)(s) = (((stmt ## stmt1) ## cstmt) ## cstmt1)(s)

END Single_Statement_Rewrites0

Single_Statement_Rewrites1[State, Data : Type] : THEORY
BEGIN

  Importing Single_Statement_Rewrites0, Plain_Mem_Rewrites

  stmt : VAR [State -> StmtResult[State]]
  stmt1 : VAR [State -> StmtResult[State]]
  stmt2 : VAR [State -> StmtResult[State]]
  stmt3 : VAR [State -> StmtResult[State]]
  cstmt : VAR [StmtResult[State] -> StmtResult[State]]
  cstmt1 : VAR [StmtResult[State] -> StmtResult[State]]
  fcstmt : VAR [[State -> StmtResult[State]] ->
    [StmtResult[State] -> StmtResult[State]]]
  fstmt : VAR [Data -> [State -> StmtResult[State]]]
  expr : VAR [State -> ExprResult[State, Data]]
  lexpr : VAR [StmtResult[State] -> ExprResult[State, Data]]

  data : VAR Data

  s : VAR State
  n : VAR int

  % Stmt Ok Elimination Expr Forms

```

```

%=====
% proof status :-)
forward_s_c_s_of_s_expr : Lemma
  (stmt ## cstmt)(s) = stmt(s) Implies
    (stmt ## cstmt ## expr)(s) = (stmt ## expr)(s)

% proof status :-)
stmt_ok_lift_expr : Lemma
  OK?(stmt1(s)) Implies
    (stmt1 ## lift(stmt2) ## expr)(s) = (stmt1 ## stmt2 ## expr)(s)

% proof status :-)
break_break_stmt_expr : Lemma
  Break?(stmt(s)) Implies
    (stmt ## stmt1 ## catch_break ## expr)(s) = (stmt ## catch_break ## expr)(s)

% proof status :-)
break_break_catch_continue_expr : Lemma
  Break?(stmt(s)) Implies
    (stmt ## catch_continue ## catch_break ## expr)(s) = (stmt ## catch_break ## expr)(s)

% proof status :-)
break_break_catch_default_expr : Lemma
  Break?(stmt(s)) Implies
    (stmt ## catch_default ## catch_break ## expr)(s) = (stmt ## catch_break ## expr)(s)

% proof status :-)
break_break_lift_stmt_expr : Lemma
  Break?(stmt(s)) Implies
    (stmt ## lift(stmt1) ## catch_break ## expr)(s) = (stmt ## catch_break ## expr)(s)

% proof status :-)
break_break_case_expr : Lemma
  Break?(stmt(s)) Implies
    (stmt ## case(n) ## catch_break ## expr)(s) = (stmt ## catch_break ## expr)(s)

% proof status :-)
break_break_default_expr : Lemma
  Break?(stmt(s)) Implies
    (stmt ## default ## catch_break ## expr)(s) = (stmt ## catch_break ## expr)(s)

% proof status :-)
continue_break_stmt_expr : Lemma
  Continue?(stmt(s)) Implies
    (stmt ## stmt1 ## catch_break ## expr)(s) = (stmt ## catch_break ## expr)(s)

% proof status :-)
continue_break_catch_continue_expr : Lemma
  Continue?(stmt(s)) Implies
    (stmt ## catch_break ## catch_continue ## expr)(s) = (stmt ## catch_continue ## expr)(s)

% proof status :-)

```

```

continue_break_catch_default_expr : Lemma
  Continue?(stmt(s)) Implies
    (stmt ## catch_default ## catch_continue ## expr)(s) = (stmt ## catch_continue ## expr)(s)

% proof status :-)
continue_break_lift_stmt_expr : Lemma
  Continue?(stmt(s)) Implies
    (stmt ## lift(stmt1) ## catch_continue ## expr)(s) = (stmt ## catch_continue ## expr)(s)

% proof status :-)
continue_break_case_expr : Lemma
  Continue?(stmt(s)) Implies
    (stmt ## case(n) ## catch_continue ## expr)(s) = (stmt ## catch_continue ## expr)(s)

% proof status :-)
continue_break_default_expr : Lemma
  Continue?(stmt(s)) Implies
    (stmt ## default ## catch_continue ## expr)(s) = (stmt ## catch_continue ## expr)(s)

% While
%=====

b_ex : VAR [State -> ExprResult[State, bool]]
max : Var nat

% proof status :-)
% while_unroll_expr_TCC1 : TCC Obligation

% proof status :-)
while_unroll_expr : Lemma
  max > 0 Implies
    (while(max, b_ex, stmt) ## expr)(s) =
    (if_else(b_ex, stmt ## catch_continue ##
      while(max - 1, b_ex, stmt) ## catch_break, skip) ## expr)(s)

% proof status :-)
while_unroll_lexpr : Lemma
  max > 0 Implies
    (while(max, b_ex, stmt) ## lexpr)(s) =
    (if_else(b_ex, stmt ## catch_continue ##
      while(max - 1, b_ex, stmt) ## catch_break, skip) ## lexpr)(s)

% proof status :-)
stmt_while_unroll_expr : Lemma
  max > 0 Implies
    (stmt1 ## while(max, b_ex, stmt2) ## expr)(s) =
    (stmt1 ## if_else(b_ex, stmt2 ## catch_continue ##
      while(max - 1, b_ex, stmt2) ## catch_break, skip) ## expr)(s)

% proof status :-)
stmt_while_unroll_lexpr : Lemma
  max > 0 Implies

```

```

(stmt1 ## while(max, b_ex, stmt2) ## lexpr)(s) =
(stmt1 ## if_else(b_ex, stmt2 ## catch_continue ##
  while(max - 1, b_ex, stmt2) ## catch_break, skip) ## lexpr)(s)

% E2s-Final Rewrite
%=====
% proof status :-)
e2s_ok : Lemma
  OK?(expr(s)) Implies
  OK?(e2s(expr)(s))

% proof status :-)
stmt_e2s_ok : Lemma
  OK?((stmt ## expr)(s)) Implies
  OK?((stmt ## e2s(expr))(s))

% proof status :-)
% e2s_state_TCC1 : TCC Obligation

% proof status :-)
% e2s_state_TCC2 : TCC Obligation

% proof status :-)
e2s_state : Lemma
  OK?(expr(s)) Implies
  state(e2s(expr)(s)) = state(expr(s))

% proof status :-)
% stmt_e2s_state_TCC1 : TCC Obligation

% proof status :-)
% stmt_e2s_state_TCC2 : TCC Obligation

% proof status :-)
stmt_e2s_state : Lemma
  OK?((stmt ## expr)(s)) Implies
  state((stmt ## e2s(expr))(s)) = state((stmt ## expr)(s))

% Fexpr / Fstmt Rewrite
%=====

% proof status :-)
comp_eval_lif_ok_fstmt : Lemma
  OK?(expr(s)) Implies
  (expr ## fstmt)(s) = (e2s(expr) ## fstmt(data(expr(s))))(s)

% proof status :-)
stmt_eval_lif_ok_fstmt : Lemma
  OK?((stmt ## expr)(s))
Implies
  (stmt ## (expr ## fstmt))(s) = (stmt ## (e2s(expr) ## fstmt(data((stmt ## expr)(s)))))(s)

```

B PVS Theory Sources

```

% proof status :-)
comp_eval_if_ok_fstmt_cstmt : Lemma
  OK?(expr(s)) Implies
    ((expr ## fstmt) ## cstmt)(s) = (e2s(expr) ## fstmt(data(expr(s))) ## cstmt)(s)

% proof status :-)
stmt_eval_if_ok_fstmt_cstmt : Lemma
  OK?((stmt ## expr)(s))
Implies
  (stmt ## (expr ## fstmt) ## cstmt)(s) = (stmt ## e2s(expr) ## fstmt(data((stmt ## expr)(s))) ## cstm

% Composition
%=====

% proof status :-)
composition_assoc_rewrite_expr_1 : Lemma
  (stmt ## (stmt1 ## expr)) = ((stmt ## stmt1) ## expr)

% proof status :-)
composition_assoc_rewrite_expr_2 : Lemma
  (stmt ## (cstmt ## expr)) = ((stmt ## cstmt) ## expr)

% proof status :-)
composition_assoc_rewrite_expr_3 : Lemma
  (cstmt ## (cstmt1 ## expr)) = ((cstmt ## cstmt1) ## expr)

% proof status :-)
composition_assoc_rewrite_expr_4 : Lemma
  (stmt ## (lift(stmt1) ## lexpr)) = ((stmt ## stmt1) ## lexpr)

% proof status :-)
composition_assoc_rewrite_expr_5 : Lemma
  (stmt ## (cstmt ## lexpr)) = ((stmt ## cstmt) ## lexpr)

% proof status :-)
composition_assoc_rewrite_expr_6 : Lemma
  (cstmt ## (cstmt1 ## lexpr)) = ((cstmt ## cstmt1) ## lexpr)

% proof status :-)
composition_assoc_rewrite_stmt_ok : Lemma
  OK?(stmt(s)) Implies
  (stmt ## (stmt1 ## cstmt))(s) = ((stmt ## stmt1) ## cstmt)(s)

% proof status :-)
composition_assoc_rewrite_stmt_ok_expr : Lemma
  OK?(stmt(s)) Implies
  (stmt ## (stmt1 ## cstmt) ## expr)(s) = ((stmt ## stmt1) ## cstmt ## expr)(s)

% proof status :-)
composition_assoc_rewrite_stmt_ok_expr_lexpr : Lemma
  OK?(stmt(s)) Implies
  (stmt ## (stmt1 ## cstmt) ## lexpr)(s) = ((stmt ## stmt1) ## cstmt ## lexpr)(s)

```

```

% proof status :-)
ok_result_fstmt : Lemma
  Forall (data : Data) :
    ok_result(data) ## fstmt = fstmt(data)

pm : Var Plain_Memory[State]
dt : Var (interpreted_data_type?[Data])
addr : Var Address

% proof status :-)
pm_q_prop_read_ok_s_read_ok : Lemma
  pm_q_prop_read(pm, dt, addr)(skip(s)) Implies
    OK?(read_data(pm, dt)(addr)(s))

END Single_Statement_Rewrites1

Single_Statement_Rewrites2 [State, Data1, Data2 : Type] : THEORY
BEGIN

Importing Single_Statement_Rewrites1

s : VAR State

stmt : VAR [State -> StmtResult[State]]
cstmt : VAR [StmtResult[State] -> StmtResult[State]]
expr : VAR [State -> ExprResult[State, Data1]]
expr1 : VAR [State -> ExprResult[State, Data2]]
fexpr : VAR [Data2 -> [State -> ExprResult[State, Data1]]]
fstmt : VAR [Data1 -> [State -> StmtResult[State]]]

% Lemmas to speed up rewriting
%=====
% proof status :-)
ok_result_fexpr : Lemma
  Forall (data : Data2) :
    ok_result(data) ## fexpr = fexpr(data)

% E2s Expr
%=====

% proof status :-)
e2s_merge : Lemma
  (e2s(expr) ## e2s(expr1)) = e2s(expr ## expr1)

% proof status :-)
stmt_e2s_merge : Lemma
  (stmt ## e2s(expr) ## e2s(expr1)) =
    (stmt ## e2s(expr ## expr1))

% proof status :-)

```



```

e2s_expr : Lemma
  (e2s(expr1) ## expr)(s) = (expr1 ## expr)(s)

% proof status :-)
stmt_e2s_expr : Lemma
  (stmt ## e2s(expr1) ## expr)(s) = (stmt ## expr1 ## expr)(s)

% proof status :-)
composition_assoc_expression_rewrite_stmt : Lemma
  (stmt ## (expr ## expr1)) = ((stmt ## expr) ## expr1)

% proof status :-)
comp_eval_if_ok_fstmt_expr : Lemma
  OK?(expr(s)) Implies
    ((expr ## fstmt) ## expr1)(s) = (e2s(expr) ## fstmt(data(expr(s))) ## expr1)(s)

% proof status :-)
stmt_eval_if_ok_fstmt_expr : Lemma
  OK?((stmt ## expr)(s)) Implies
    (stmt ## (expr ## fstmt) ## expr1)(s) =
    (stmt ## e2s(expr) ## fstmt(data((stmt ## expr)(s))) ## expr1)(s)

% proof status :-)
stmt_eval_if_ok_fexpr : Lemma
  OK?((stmt ## expr1)(s)) Implies
    (stmt ## (expr1 ## fexpr))(s) = (stmt ## expr1 ## fexpr(data((stmt ## expr1)(s))))(s)

% proof status :-)
comp_eval_if_ok_fstmt_cstmt_expr : Lemma
  OK?(expr(s)) Implies
    (((expr ## fstmt) ## cstmt) ## expr1)(s) =
    (((e2s(expr) ## fstmt(data(expr(s)))) ## cstmt) ## expr1)(s)

% proof status :-)
stmt_eval_if_ok_fstmt_cstmt_expr : Lemma
  OK?((stmt ## expr)(s)) Implies
    (stmt ## (expr ## fstmt) ## cstmt ## expr1)(s) =
    (stmt ## e2s(expr) ## fstmt(data((stmt ## expr)(s))) ## cstmt ## expr1)(s)

```

End Single_Statement_Rewrites2

Single_Statement_Rewrites3 [State, Data1, Data2, Data3 : **Type**] : **THEORY**
BEGIN

Importing Single_Statement_Rewrites2

% Assumption: there is not always an expression at the end !!!

```

expr : VAR [State -> ExprResult[State, Data1]]
expr1 : VAR [State -> ExprResult[State, Data2]]
expr2 : VAR [State -> ExprResult[State, Data3]]

```

```

fexpr : VAR [Data1 -> [State -> ExprResult[State, Data3]]]

stmt : VAR [State -> StmtResult[State]]

s : VAR State

% E2s Expr
%=====

% proof status :-)
ok_result_elimination_2 : Lemma
  Forall (data : Data3) :
    ((expr ## ok_result[State, Data3](data)) ## expr1) = (expr ## expr1)

% Composition
%=====

% proof status :-)
comp_eval_if_ok_fexpr_expr : Lemma
  OK?(expr(s)) Implies
    ((expr ## fexpr) ## expr1)(s) = (expr ## fexpr(data(expr(s))) ## expr1)(s)

% proof status :-)
expr_eval_if_ok_fexpr : Lemma
  OK?((expr1 ## expr)(s)) Implies
    (expr1 ## (expr ## fexpr))(s) = (expr1 ## expr ## fexpr(data((expr1 ## expr)(s))))(s)

% proof status :-)
composition_assoc_expression_rewrite : Lemma
  (expr ## (expr1 ## expr2)) = ((expr ## expr1) ## expr2)

% proof status :-)
stmt_eval_if_ok_fexpr_expr : Lemma
  OK?((stmt ## expr)(s)) Implies
    (stmt ## (expr ## fexpr) ## expr1)(s) =
      (stmt ## expr ## fexpr(data((stmt ## expr)(s))) ## expr1)(s)

END Single_Statement_Rewrites3

```

```

Single_Statement_Rewrites4 [State, Data1, Data2, Data3, Data4 : Type] : THEORY
BEGIN

```

```

  Importing Single_Statement_Rewrites3

```

```

  expr : VAR [State -> ExprResult[State, Data1]]
  expr1 : VAR [State -> ExprResult[State, Data2]]
  expr2 : VAR [State -> ExprResult[State, Data4]]
  fexpr : VAR [Data2 -> [State -> ExprResult[State, Data3]]]

  s : VAR State

```

```

% Composition
%=====

% proof status :-)
expr_eval_if_ok_fexpr_expr : Lemma
  OK?((expr ## expr1)(s)) Implies
    (expr ## (expr1 ## fexpr) ## expr2)(s) =
      (expr ## expr1 ## fexpr(data((expr ## expr1)(s))) ## expr2)(s)

END Single_Statement_Rewrites4

While_Hoare[State, Ord : Type] : Theory
Begin

  Importing Single_Statement_Rewrites4, More_Relations

  s0 : Var State
  < : Var PRED[[Ord, Ord]]

  b_ex : Var [State -> ExprResult[State, bool]]
  body : Var [State -> StmtResult[State]]

  n : Var nat

  % proof status :-)
  iterate_while_right : Lemma
    Forall (n : nat) :
      iterate_while(1 + n, b_ex, body) =
        iterate_while(n, b_ex, body) ## lift(e2s(b_ex)) ## body ## catch_continue)

  % While Invariant
  %-----

  % proof status :-)
  % while_variant?_TCC1 : TCC Obligation

  while_variant?(b_ex, body, <)(variant : [State -> Ord],
    Q : PRED[StmtResult[State]]) : bool =
    well_founded?(<)
  And
    while_invariant?(b_ex, body)(Q, Q)
  And
    Forall(s : State) :
      Q(OK(s)) And OK?(b_ex(s)) And data(b_ex(s)) And
      OK?((e2s(b_ex) ## body ## catch_continue)(s))
  Implies
    variant(state((e2s(b_ex) ## body ## catch_continue)(s))) < variant(s)

  % proof status :-)
  % while_no_cb_TCC1 : TCC Obligation

```

```

while_no_cb(b_ex, body) : [State -> StmtResult[State]] =
  Lambda (s : State) :
    If Exists (n : nat): while_termination_point?(b_ex, body, s)(n) Then
      Let
        i = min[nat](while_termination_point?(b_ex, body, s))
      In
        (iterate_while(i, b_ex, body) ## e2s(b_ex))(s)
    Else
      Hang
    Endif

% proof status :-)
while_no_cb_cb_while : Lemma
  while(b_ex, body) = (while_no_cb(b_ex, body) ## catch_break)

% proof status :-)
iterate_while_invariant : Lemma
  Forall (inv_abnorm?, inv_norm? : PRED[StmtResult[State]], n : nat) :
    while_invariant?(b_ex, body)(inv_abnorm?, inv_norm?)
  And
    inv_abnorm?(OK(s0))
  And
    (Forall (m : nat) : m < n Implies Not while_termination_point?(b_ex, body, s0)(m))
  Implies
    inv_abnorm?(iterate_while(n, b_ex, body)(s0))

% proof status :-)
min_termination_point_no_less : Lemma
  Forall (m : nat) :
    (Exists (n : nat) : while_termination_point?(b_ex, body, s0)(n))
  And
    m < min[nat](while_termination_point?(b_ex, body, s0))
  Implies
    Not while_termination_point?(b_ex, body, s0)(m)

% proof status :-)
while_terminates : Lemma
  Forall (term_inv? : PRED[StmtResult[State]], variant : [State -> Ord]) :
    while_variant?(b_ex, body, <)(variant, term_inv?) And %term_inv?(s0)
    term_inv?(OK(s0))
  Implies
    Exists (n : nat) : while_termination_point?(b_ex, body, s0)(n)

% proof status :-)
hoare_while : Lemma
  Forall (term_inv? : PRED[StmtResult[State]],

```

```

        data_inv_abnorm?, data_inv_norm? : PRED[StmtResult[State]],
        variant : [State -> Ord]) :
    while_variant?(b_ex, body, <)(variant, term_inv?)
And
    while_invariant?(b_ex, body)(data_inv_abnorm?, data_inv_norm?)
And
    data_inv_abnorm?(OK(s0)) And %term_inv?(s0)
    term_inv?(OK(s0))
Implies
    Let res = while_no_cb(b_ex, body)(s0) In
        term_inv?(res)
    And
        ((data_inv_abnorm?(res) And Not OK?(res)) OR
        data_inv_norm?(res))

End While_Hoare

While_Rewrites[State, Ord : Type] : Theory
Begin

    Importing While_Hoare

    s : VAR State
    pm : Var Plain_Memory[State]

    b_ex : VAR [State -> ExprResult[State, bool]]
    body : VAR [State -> StmtResult[State]]

    stmt : VAR [State -> StmtResult[State]]

    % data invariant: P = abnormal termination, Q = normal termination
    % termination invariant : Q
    P, Q, R : VAR [State -> PRED[StmtResult[State]]]

    % property
    S : VAR PRED[StmtResult[State]]

    % variant
    variant : VAR [State -> [State -> Ord]]
    < : VAR PRED[[Ord, Ord]]

    while_no_cb(b_ex, body)(P, Q, R, variant, <) : [State -> StmtResult[State]] =
        while_no_cb[State, Ord](b_ex, body)

    % proof status :-)
    while_to_while_no_cb : Lemma
        while[State, Ord](b_ex, body)(P, Q, R, variant, <) =
            (while_no_cb(b_ex, body)(P, Q, R, variant, <) ## catch_break)

```

```

% proof status :-)
while_inv_rewrite_termination_result : Lemma
  P(s)(skip(s)) And R(s)(skip(s))
And
  while_variant?(b_ex, body, <)(variant(s), R(s))
And
  while_invariant?(b_ex, body)(P(s), Q(s))
And
  (Forall (res : StmtResult[State]) : R(s)(res) Implies S(res))
Implies
  S(while_no_cb(b_ex, body)(P, Q, R, variant, <)(s))

```

```

% proof status :-)
while_inv_rewrite_data_ok : Lemma
  P(s)(skip(s)) And R(s)(skip(s))
And
  while_variant?(b_ex, body, <)(variant(s), R(s))
And
  while_invariant?(b_ex, body)(P(s), Q(s))
And
  OK?(while_no_cb(b_ex, body)(P, Q, R, variant, <)(s))
And
  (Forall (res : StmtResult[State]) : Q(s)(res) Implies S(res))
Implies
  S(while_no_cb(b_ex, body)(P, Q, R, variant, <)(s))

```

```

% proof status :-)
while_inv_rewrite_data_break : Lemma
  P(s)(skip(s)) And R(s)(skip(s))
And
  while_variant?(b_ex, body, <)(variant(s), R(s))
And
  while_invariant?(b_ex, body)(P(s), Q(s))
And
  Break?(while_no_cb(b_ex, body)(P, Q, R, variant, <)(s))
And
  (Forall (res : StmtResult[State]) : (P(s)(res) Or Q(s)(res)) And Break?(res) Implies S(res))
Implies
  S(while_no_cb(b_ex, body)(P, Q, R, variant, <)(s))

```

```

% proof status :-)
while_inv_rewrite_data_return : Lemma
  P(s)(skip(s)) And R(s)(skip(s))
And
  while_variant?(b_ex, body, <)(variant(s), R(s))
And
  while_invariant?(b_ex, body)(P(s), Q(s))
And
  Return?(while_no_cb(b_ex, body)(P, Q, R, variant, <)(s))

```

```

And
  (Forall (res : StmtResult[State]) : (P(s)(res) Or Q(s)(res)) And Return?(res) Implies S(res))
Implies
  S(while_no_cb(b_ex, body)(P, Q, R, variant, <))(s))

% proof status :-)
% stmt_while_inv_rewrite_termination_result_TCC1 : TCC Obligation

% proof status :-)
stmt_while_inv_rewrite_termination_result : Lemma
  OK?(stmt(s)) And P(state(stmt(s)))(stmt(s)) And R(state(stmt(s)))(stmt(s))
And
  while_variant?(b_ex, body, <)(variant(state(stmt(s))), R(state(stmt(s))))
And
  while_invariant?(b_ex, body)(P(state(stmt(s))), Q(state(stmt(s))))
And
  (Forall (res : StmtResult[State]) : R(state(stmt(s)))(res) Implies S(res))
Implies
  S((stmt ## while_no_cb(b_ex, body)(P, Q, R, variant, <)))(s))

% (auto-rewrite "stmt_while_inv_rewrite_data_ok!")
% Can't rewrite using stmt_while_inv_rewrite_data_ok: LHS key S is bad.

% proof status :-)
stmt_while_inv_rewrite_data_ok : Lemma
  OK?(stmt(s)) And P(state(stmt(s)))(stmt(s)) And R(state(stmt(s)))(stmt(s))
And
  while_variant?(b_ex, body, <)(variant(state(stmt(s))), R(state(stmt(s))))
And
  while_invariant?(b_ex, body)(P(state(stmt(s))), Q(state(stmt(s))))
And
  OK?((stmt ## while_no_cb(b_ex, body)(P, Q, R, variant, <)))(s))
And
  (Forall (res : StmtResult[State]) : Q(state(stmt(s)))(res) Implies S(res))
Implies
  S((stmt ## while_no_cb(b_ex, body)(P, Q, R, variant, <)))(s))

% proof status :-)
stmt_while_inv_rewrite_data_break : Lemma
  OK?(stmt(s)) And P(state(stmt(s)))(stmt(s)) And R(state(stmt(s)))(stmt(s))
And
  while_variant?(b_ex, body, <)(variant(state(stmt(s))), R(state(stmt(s))))
And
  while_invariant?(b_ex, body)(P(state(stmt(s))), Q(state(stmt(s))))
And
  Break?((stmt ## while_no_cb(b_ex, body)(P, Q, R, variant, <)))(s))
And
  (Forall (res : StmtResult[State]) :
    (P(state(stmt(s)))(res) Or Q(state(stmt(s)))(res)) And Break?(res) Implies S(res))
Implies
  S((stmt ## while_no_cb(b_ex, body)(P, Q, R, variant, <)))(s))

```

```

% proof status :-)
stmt_while_inv_rewrite_data_return : Lemma
  OK?(stmt(s) And P(state(stmt(s)))(stmt(s)) And R(state(stmt(s)))(stmt(s))
And
  while_variant?(b_ex, body, <)(variant(state(stmt(s))), R(state(stmt(s))))
And
  while_invariant?(b_ex, body)(P(state(stmt(s))), Q(state(stmt(s))))
And
  Return?((stmt ## while_no_cb(b_ex, body)(P, Q, R, variant, <))(s))
And
  (Forall (res : StmtResult[State]) :
    (P(state(stmt(s)))(res) Or Q(state(stmt(s)))(res)) And Return?(res) Implies S(res))
Implies
  S((stmt ## while_no_cb(b_ex, body)(P, Q, R, variant, <))(s))

```

```

% proof status :-)
pm_q_prop_while : Lemma
Let pm_q_prop_inv = Lambda (s : State) : Lambda (res : StmtResult[State]) :
  pm_q_prop(pm)(catch_break(res))
In
  pm'states(s) And R(s)(skip(s))
And
  while_variant?(b_ex, body, <)(variant(s), R(s))
And
  while_invariant?(b_ex, body)(pm_q_prop_inv(s), pm_q_prop_inv(s))
Implies
  pm_q_prop(pm)(while(b_ex, body)(P, Q, R, variant, <))(s)

```

```

% proof status :-)
% pm_q_prop_stmt_while_TCC1 : TCC Obligation

```

```

% proof status :-)
pm_q_prop_stmt_while : Lemma
Let pm_q_prop_inv = Lambda (s : State) : Lambda (res : StmtResult[State]) :
  pm_q_prop(pm)(catch_break(res))
In
  pm_q_prop(pm)(stmt(s)) And R(state(stmt(s)))(stmt(s))
And
  while_variant?(b_ex, body, <)(variant(state(stmt(s))), R(state(stmt(s))))
And
  while_invariant?(b_ex, body)(pm_q_prop_inv(state(stmt(s))), pm_q_prop_inv(state(stmt(s))))
Implies
  pm_q_prop(pm)((stmt ## while(b_ex, body)(P, Q, R, variant, <))(s))

```

End While_Rewrites

```

%%%%%%%%%%
%
```


B PVS Theory Sources

```
% Statement_Rewrites: imports everything
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
Statement_Rewrites[State : Type] : Theory
Begin
```

```
    Importing While_Rewrites
```

```
End Statement_Rewrites
```

B.30 statements.pvs

```
% $Id: statements.pvs,v 1.37.2.1.2.1.2.2 2008/05/14 14:25:00 tews Exp $
%
% Author: Tjark Weber (based on lecture material by Hendrik Tews)
% Marcus Völz (rewriting rules)
% (c) 2007 Radboud University
%
% A shallow embedding of (some) C++ statements as state transformers.

% Section numbers in comments throughout this file refer to
% "Programming languages – C++", ISO/IEC 14882:1998(E)

% Most statements have type
% State -> StmtResult[State]
%
% "case" and "default" label statements however have type
% StmtResult[State] -> StmtResult[State]
% so that they can catch a "Switch" or "Default" result.
%
% Functions of the former type are lifted to the latter type by modifying
% results of the form
% OK(state)
% only, and passing the result through unchanged otherwise (see the definitions
% of ## in state-transformer.pvs).

% Deviations from the C++ standard:
%
% "goto" (6.6.4) and goto labels (6.1) have not been modeled. This would not
% be difficult in principle, but the possibility of jumps into substatements
% (e.g. into an "if" or "while") would make the formalization of these
% statements more complicated. Also note that (backward) gotos may cause
% nontermination and would need to be modeled somewhat similar to loops.
%
% For the same reason, "case" and "default" labels are required to appear on
% top-level statements contained within the (compound) substatement of a
% "switch" statement only, contrary to what 6.4.2.6 specifies. (Note that this
% rules out code like Duff's Device.)
%
% The condition of a "switch" statement must evaluate to a value of type int
% (cf. 6.4.2.2).
```

```

% >> This is according to the standard: the condition must be of integral type,
% enum type (which has an integral base type) or of a class type that can
% be converted to an integral or enum type.
%
% Functions must always use a "return" statement to return; flowing off the
% end of a function is not allowed (cf. 6.6.3.2). This can be achieved through
% a trivial source code transformation, where a "return_void" statement is
% appended (as the body's last statement) to each function with return type
% void. Alternatively, "catch_return" (see expressions.pvs) could be
% overloaded to convert OK(s) into OK(s, unit), but that would probably cause
% ambiguities in type checking.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% 6.1: Labeled statement
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

```

LabeledStatements[State : Type] : THEORY

BEGIN

IMPORTING StmtResult

```

const : VAR int
stmt : VAR [State -> StmtResult[State]]

```

```

% res : VAR StmtResult[State]
% ident : VAR string

```

```

% goto target: not modeled
% label(ident)(res) : StmtResult[State] =
% CASES res OF
% Goto(state, label):
% if label = ident then OK(state) else res endif
% ELSE res
% ENDCASES

```

```

% switch: case target
case(const) : [StmtResult[State] -> StmtResult[State]] =
  Lambda (res : StmtResult[State]) :
    CASES res OF
      Switch(state, case):
        if case = const then OK(state) else res endif
    ELSE res
  ENDCASES

```

```

% switch: default target
default : [StmtResult[State] -> StmtResult[State]] =
  Lambda (res : StmtResult[State]) :
    CASES res OF
      Default(state): OK(state)
    ELSE res

```

ENDCASES

END LabeledStatements

```

%%%%%%%%%%
%
% 6.2: Expression statement
%
%%%%%%%%%%

```

SkipStatements[State : **Type**] : **THEORY**
BEGIN

IMPORTING StmtResult

s : **VAR** State

```

skip : [State -> StmtResult[State]] =
  Lambda (s : State) : OK(s)

```

END SkipStatements

ExpressionStatements[State, ExprData : **Type**] : **THEORY**
BEGIN

IMPORTING ExprResult, StmtResult

s : **VAR** State

ex : **VAR** [State -> ExprResult[State,ExprData]]

```

% expression-to-statement
e2s(ex) : [State -> StmtResult[State]] =
  Lambda (s : State) :
    CASES ex(s) OF
      OK(state, data): OK(state), % forget the expression data
      Fatal: Fatal,
      Hang: Hang,
      % TODO: other possible expression results?
      Exception(ex, state) : Exception(ex, state)
    ENDCASES

```

END ExpressionStatements

```

%%%%%%%%%%
%
% 6.3: Compound statement or block
%
%%%%%%%%%%

```

% There is no need to model blocks here (or so I hope — what about the scope of

```
% declarations though?). Composition of statements is defined as
% ##
% in state-transformer.pvs.
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% Auxiliary catch_XXX functions; used to model jumps (cf. 6.6)
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
CatchAux[State : Type] : THEORY
BEGIN
```

```
IMPORTING ExprResult, StmtResult
```

```
res : VAR StmtResult[State]
```

```
catch_break(res) : StmtResult[State] =
  CASES res OF
    Break(state): OK(state)
  ELSE res
  ENDCASES
```

```
catch_continue(res) : StmtResult[State] =
  CASES res OF
    Continue(state): OK(state)
  ELSE res
  ENDCASES
```

```
% catch_switch: not needed since a switch statement will be entered with a
% "Switch" result only if there is a matching case target
```

```
catch_default(res) : StmtResult[State] =
  CASES res OF
    Default(state): OK(state)
  ELSE res
  ENDCASES
```

```
catch_return(res) : StmtResult[State] =
  CASES res OF
    Return(state): OK(state)
  ELSE res
  ENDCASES
```

```
END CatchAux
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% 6.4: Selection statements (if, switch)
```

B PVS Theory Sources

```
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
SelectionStatements[State : Type] : THEORY
BEGIN

  IMPORTING State_Transformer, CatchAux

  s : VAR State

  b_ex : VAR [State -> ExprResult[State,bool]]
  t1, t2: VAR [State -> StmtResult[State]]

  n_ex : VAR [State -> ExprResult[State,int]]
  case_consts : VAR list[int]
  case_stm : VAR [StmtResult[State] -> StmtResult[State]]

  % 6.4.1
  % if ... else ...
  if_else(b_ex, t1, t2) : [State -> StmtResult[State]] =
    (b_ex ## lambda (b : bool): if b then t1 else t2 endif)

  % if without else (6.4.1) is equivalent to if_else(b_ex, t1, skip)

  % 6.4.2
  % switch

  % case_consts is a list of all constants that occur as case labels within the
  % switch statement. We need it to determine beforehand if there is one such
  % label that matches; otherwise the default label (if there is one) must be
  % taken (cf. 6.4.2.5).

  switch(n_ex, case_consts, case_stm) : [State -> StmtResult[State]] =
  (
    n_ex ## (lambda (n : int) :
      if member(n, case_consts) then
        switch_result(n) ## case_stm
      else
        default_result ## case_stm ## catch_default
      endif) ## catch_break
    )

END SelectionStatements

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% 6.5: Iteration statements (while, do while, for)
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
IterationStatements1[State : Type] : THEORY
```

BEGIN

IMPORTING SkipStatements, ExpressionStatements,
SelectionStatements, Number_Props

s : **VAR** State

b_ex: **VAR** [State → ExprResult[State,bool]]

body: **VAR** [State → StmtResult[State]]

% 6.5.1

% while

% proof status :-)

% iterate_while_TCC1 : TCC Obligation

% proof status :-)

% iterate_while_TCC2 : TCC Obligation

*% unroll the loop n times (must return the state immediately before
% evaluation of b_ex, so that we can define while_termination_point?)*

iterate_while(n : nat, b_ex, body)(s) : **RECURSIVE**

 StmtResult[State] =

(if n = 0 **then**

 skip

else

 e2s(b_ex) ##

 body ##

 catch_continue ##

 iterate_while(n-1, b_ex, body)

endif)(s)

MEASURE n

% proof status :-)

% while_termination_point?_TCC1 : TCC Obligation

while_termination_point?(b_ex, body, s)(n : nat) : bool =

let res = iterate_while(n, b_ex, body)(s) **in**

not OK?(res) **or**

not OK?(b_ex(state(res))) **or**

not data(b_ex(state(res)))

% proof status :-)

% while_TCC1 : TCC Obligation

while(b_ex, body) : [State → StmtResult[State]] =

Lambda (s : State) :

if exists (n : nat): while_termination_point?(b_ex, body, s)(n) **then**

let

 i = min[nat](while_termination_point?(b_ex, body, s))

in (

 iterate_while(i, b_ex, body) ##

```

        e2s(b_ex) ## % evaluate b_ex one more time
        catch_break
    )(s)
else
    Hang
endif

% proof status :-)
nonzero_min_while_termination_point : Lemma
  (Exists (n : nat) : while_termination_point?(b_ex, body, s)(n)) And
  Not min[nat](while_termination_point?(b_ex, body, s)) = 0
  Implies
  OK?(b_ex(s)) And data(b_ex(s))

% Warning do not auto-rewrite with while_as_if_while.
% Use while(max, ...) and while_unroll instead
% proof status :-)
while_as_if_while : Lemma
  while(b_ex, body) = if_else(b_ex, body ## catch_continue ##
    while(b_ex, body) ## catch_break, skip)

while(max : nat, b_ex, body) : [State -> StmtResult[State]] =
  while(b_ex, body)

% Warning do not auto-rewrite with this lemma --- takes too long;
% use while_unroll_rewrite* instead
% while_unroll : Lemma
% Forall (max : nat) : max > 0
% Implies
% while(max, b_ex, body) =
% if_else(b_ex, body ## catch_continue ## while(max - 1, b_ex, body) ##
% catch_break, skip)

% while_unroll_0 : Lemma
% while(0, b_ex, body) = while(b_ex, body)

while_invariant?(b_ex, body)(P, Q : PRED[StmtResult[State]]) : bool =
  Forall (s : State) :
    P(skip(s)) Implies
    If OK?(b_ex(s)) And data(b_ex(s)) Then
      P((e2s(b_ex) ## body ## catch_continue[State])(s))
    Else
      Q(e2s(b_ex)(s))
    Endif

% 6.5.2
% do

% unroll the loop n times (must return the state immediately before
% evaluation of b_ex, so that we can define do_termination_point?)

```

```

iterate_do(n : nat, body, b_ex)(s) : RECURSIVE
  StmtResult[State] =
  (if n = 0 then
    body ##
    catch_continue
  else
    body ##
    catch_continue ##
    e2s(b_ex) ##
    iterate_do(n-1, body, b_ex)
  endif)(s)
MEASURE n

% proof status :-)
iterate_do_as_while : Lemma
  Forall (n : nat) :
    iterate_do(n, body, b_ex) = (body ## catch_continue ##
                                  iterate_while(n, b_ex, body))

% proof status :-)
% do_termination_point?_TCC1 : TCC Obligation

do_termination_point?(body, b_ex, s)(n : nat) : bool =
  let res = iterate_do(n, body, b_ex)(s) in
  not OK?(res) or
  not OK?(b_ex(state(res))) or
  not data(b_ex(state(res)))

% proof status :-)
% termination_point_do_as_while_TCC1 : TCC Obligation

% proof status :-)
termination_point_do_as_while : Lemma
  OK?((body ## catch_continue)(s)) Implies
  do_termination_point?(body, b_ex, s) =
  while_termination_point?(b_ex, body, state(
    (body ## catch_continue)(s)))

% proof status :-)
% do_while_TCC1 : TCC Obligation

do_while(body, b_ex) : [State -> StmtResult[State]] =
  Lambda (s : State) :
  if exists (n : nat): do_termination_point?(body, b_ex, s)(n) then
  let
    i = min[nat](do_termination_point?(body, b_ex, s))
  in (
    iterate_do(i, body, b_ex) ##
    e2s(b_ex) ## % evaluate b_ex one more time
    catch_break
  )(s)
  else

```



```

    Hang
  endif

% proof status :-)
do_as_while : Lemma
  do_while(body, b_ex) = (body ## catch_continue ## while(b_ex, body) ##
    catch_break)

do_while(max : nat, body, b_ex) : [State -> StmtResult[State]] =
  do_while(body, b_ex)

% proof status :-)
do_while_unroll : Lemma
  Forall (max : nat) :
    do_while(max, body, b_ex) = (body ## catch_continue ##
      while(max, b_ex, body) ## catch_break)

END IterationStatements1

IterationStatements2[State, ExprData : Type] : THEORY
BEGIN

  IMPORTING IterationStatements1

  s : VAR State

  b_ex: VAR [State -> ExprResult[State,bool]]
  body: VAR [State -> StmtResult[State]]

  % 6.5: A for-init-statement is either an expression-statement or a
  % simple-declaration. We have not modeled the latter (yet).
  init: VAR [State -> StmtResult[State]]
  expr: VAR [State -> ExprResult[State,ExprData]]

  % 6.5.3
  % for

  % proof status :-)
  % iterate_for_TCC1 : TCC Obligation

  % proof status :-)
  % iterate_for_TCC2 : TCC Obligation

  % unroll the loop n times (must return the state immediately before
  % evaluation of b_ex, so that we can define for_termination_point?)
  iterate_for(n : nat, b_ex, expr, body)(s) : RECURSIVE
    StmtResult[State] =
    (if n = 0 then
      skip
    else
      e2s(b_ex) ##

```

```

    body ##
    catch_continue ##
    e2s(expr) ##
    iterate_for(n-1, b_ex, expr, body)
endif(s)
MEASURE n

% proof status :-)
% iterate_for_as_while_TCC1 : TCC Obligation

% proof status :-)
iterate_for_as_while : Lemma
  Forall (n : nat) : n > 0 Implies
    iterate_for(n, b_ex, expr, body) =
      (e2s(b_ex) ## body ## catch_continue ##
       iterate_while(n - 1, expr ## b_ex, body) ## e2s(expr))

% proof status :-)
% for_termination_point?_TCC1 : TCC Obligation

for_termination_point?(b_ex, expr, body, s)(n : nat) : bool =
  let res = iterate_for(n, b_ex, expr, body)(s) in
    not OK?(res) or
    not OK?(b_ex(state(res))) or
    not data(b_ex(state(res)))

% proof status :-)
% termination_point_for_as_while_TCC1 : TCC Obligation

% proof status :-)
termination_point_for_as_while : Lemma
  Forall (n : nat) :
    n > 0 And
    OK?((e2s(b_ex) ## body ## catch_continue)(s))
    Implies
    for_termination_point?(b_ex, expr, body, s)(n) =
      while_termination_point?(expr ## b_ex, body,
        state((e2s(b_ex) ## body ## catch_continue)(s)))(n - 1)

% proof status :-)
% min_termination_point_for_as_while_TCC1 : TCC Obligation

% proof status :-)
% min_termination_point_for_as_while_TCC2 : TCC Obligation

% proof status :-)
% min_termination_point_for_as_while_TCC3 : TCC Obligation

% proof status :-)
min_termination_point_for_as_while : Lemma
  (Exists (n : nat):
    for_termination_point?(b_ex, expr, body, s)(n)) And

```

```

OK?[State]((e2s(b_ex) ## body ## catch_continue)(s)) And
Not min[nat](for_termination_point?(b_ex, expr, body, s)) = 0 Implies
  min[nat](for_termination_point?(b_ex, expr, body, s)) =
    min[nat](while_termination_point?(expr ## b_ex, body,
      state[State]((e2s(b_ex) ## body ## catch_continue)(s)))) + 1

% proof status :-)
% nonzero_min_for_termination_point_TCC1 : TCC Obligation

% proof status :-)
nonzero_min_for_termination_point : Lemma
  (Exists (n : nat):
    for_termination_point?(b_ex, expr, body, s)(n)) And
Not min[nat](for_termination_point?(b_ex, expr, body, s)) = 0
Implies
  OK?(b_ex(s)) And data(b_ex(s))

for(init, b_ex, expr, body) : [State -> StmtResult[State]] =
  (init ##
    lambda (state : State):
      if exists (n : nat):
        for_termination_point?(b_ex, expr, body, state)(n) then
          let
            i = min[nat](for_termination_point?(b_ex, expr, body, state))
          in (
            iterate_for(i, b_ex, expr, body) ##
            e2s(b_ex) ## % evaluate b_ex one more time
            catch_break
          )(state)
        else
          Hang
        endif
      )
  )

% proof status :-)
for_as_while : Lemma
  for(init, b_ex, expr, body) =
    (init ##
      if_else(b_ex, body ## catch_continue ## while(expr ## b_ex, body) ##
        catch_break, skip))

for(max : nat, init, b_ex, expr, body) : [State -> StmtResult[State]] =
  for(init, b_ex, expr, body)

% proof status :-)
for_unroll : Lemma
  Forall (max : nat) :
    for(max, init, b_ex, expr, body) =
      (init ##
        if_else(b_ex, body ## catch_continue ##
          while(max, expr ## b_ex, body) ## catch_break, skip))

```

END IterationStatements2

IterationStatements3[State, Ord : **Type**] : **Theory**
Begin

Importing IterationStatements1

b_ex : **VAR** [State → ExprResult[State, bool]]
body : **VAR** [State → StmtResult[State]]

% data invariant (P = abnormal termination, Q = normal termination)
% termination invariant R
P, Q, R : **VAR** [State → PRED[StmtResult[State]]]

% variant
variant : **VAR** [State → [State → Ord]]
< : **VAR** PRED[[Ord, Ord]]

% while with embedded loop invariants

while(b_ex, body)(P, Q, R, variant, <) : [State → StmtResult[State]] =
while(b_ex, body)

do_while(body, b_ex)(P, Q, R, variant, <) : [State → StmtResult[State]] =
do_while(body, b_ex)

% proof status :-)
do_while_inv_unroll : **Lemma**
do_while(body, b_ex)(P, Q, R, variant, <) =
(body ## catch_continue ## while(b_ex, body)(P, Q, R, variant, <) ##
catch_break)

End IterationStatements3

IterationStatements4[State, ExprData, Ord : **Type**] : **Theory**
Begin

Importing IterationStatements2, IterationStatements3

b_ex : **VAR** [State → ExprResult[State, bool]]
body : **VAR** [State → StmtResult[State]]

init: **VAR** [State → StmtResult[State]]
expr: **VAR** [State → ExprResult[State, ExprData]]

% data invariant (P = abnormal termination, Q = normal termination)
% termination invariant R
P, Q, R : **VAR** [State → PRED[StmtResult[State]]]

% variant
variant : **VAR** [State → [State → Ord]]

```

< : VAR PRED[[Ord, Ord]]

% while with embedded loop invariants

for(init, b_ex, expr, body)(P, Q, R, variant, <) :
  [State -> StmtResult[State]] =
  for(init, b_ex, expr, body)

% proof status :-)
for_inv_unroll : Lemma
  for(init, b_ex, expr, body)(P, Q, R, variant, <) =
  (init ##
   if_else(b_ex, body ## catch_continue ##
           while(expr ## b_ex, body)(P, Q, R, variant, <) ##
           catch_break, skip))

```

End IterationStatements4

```

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% 6.6: Jump statements (break, continue, return, goto)
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

```

JumpStatements1[State : Type] : **THEORY**
BEGIN

IMPORTING State_Transformer

```

s : VAR State
% label : VAR string

```

```

% 6.6.1
break : [State -> StmtResult[State]] =
  Lambda (s : State) : Break(s)

```

```

% 6.6.2
continue : [State -> StmtResult[State]] =
  Lambda (s : State) : Continue(s)

```

```

% 6.6.3
return_void : [State -> StmtResult[State]] =
  return_result

```

```

% 6.6.4
% goto: not modeled
% goto(label)(s) : StmtResult[State] = Goto(s, label)

```

END JumpStatements1

JumpStatements2[State, Data : **Type**] : **THEORY**

BEGIN

IMPORTING Cpp_Types,
ExpressionStatements, Abstract_Read_Write_Plain

dt : **VAR** (interpreted_data_type?[Data])

addr : **VAR** Address

pm : **VAR** Plain_Memory[State]

% % 6.6.3

% *MV: this is bogus as we have to potentially call the copy constructor*

% *for compound objects; don't do it but generate an assignment to ret_addr*

% *instead*

return(pm, dt, addr)(ex : ET[State, Data]) : [State -> StmtResult[State]] =

(

 e2s(

 ex ## **lambda** (e : Data) :

 write_data(pm, dt)(addr, e)

) ##

 return_result

)

END JumpStatements2

%%%

% 6.7: Declaration statement

%

%%%

DeclarationStatements1[State : **Type**] : **THEORY**

BEGIN

IMPORTING Abstract_Read_Write_Plain, CatchAux

fstmt : **VAR** [Address -> [State -> StmtResult[State]]]

addr : **Var** Address

uidt : **Var** Uninterpreted_data_type % *to determine size*

% *cf. 6.7.2, 6.6.2*

% *MV >> Use cpp type instead of dt*

% *This way we can allocate with proper alignment*

% *MV >> do we have non-fundamental ExprResults anyway*

allocate_stack(uidt) : [State -> ExprResult[State, Address]] =

 ok_result(Mem(0)) % *TODO: return proper stack address ;*

 % *allocate size(uidt) bytes*

deallocate_stack(uidt, addr) : [State -> StmtResult[State]] =

B PVS Theory Sources

```
Lambda (s : State) : OK(s) % TODO: modify stack (release memory again)
```

```
with_new_stackvar(uidt)(fstmt) : [State -> StmtResult[State]] =  
(  
  allocate_stack(uidt) ## Lambda (addr : Address) :  
  fstmt(addr) ##  
  lift_destructor(deallocate_stack(uidt, addr))  
)
```

END DeclarationStatements1

```
DeclarationStatements2[State, Data : Type] : THEORY  
BEGIN
```

```
IMPORTING DeclarationStatements1
```

```
fstmt : VAR [Address -> [State -> StmtResult[State]]]  
addr : Var Address  
uidt : Var Uninterpreted_data_type % to determine size  
dt : Var (interpreted_data_type?[Data])  
pm : Var Plain_Memory[State]
```

```
with_new_returnvar(pm, dt)(fstmt) : [State -> ExprResult[State, Data]] =  
(  
  allocate_stack(uidt(dt)) ## Lambda (addr : Address) :  
  fstmt(addr) ##  
  catch_return ##  
  read_data(pm, dt)(addr)  
)
```

END DeclarationStatements2

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%  
%  
% Assembler statement (this is NOT a C++ statement)  
%  
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
AssemblerStatements[State: Type] : THEORY  
BEGIN
```

```
IMPORTING SkipStatements, State_Transformer
```

```
str : VAR string
```

```
% We do not model assembler statements. In fact, we can't model them in  
% general, unless we know a lot more about the machine state. This  
% definition is provided only so that the semantics compiler can translate  
% assembler statements to type-correct (but unverifiable) PVS code.
```

```
asm(str) : [State -> StmtResult[State]] =
  Lambda (s : State) : Fatal
```

```
END AssemblerStatements
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% Statements: imports everything
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
Statements[State : Type] : THEORY
BEGIN
```

```
  IMPORTING LabeledStatements,
            % SkipStatements,
            % ExpressionStatements,
            % SelectionStatements,
            % IterationStatements1,
            % IterationStatements2,
            % IterationStatements3,
            IterationStatements4,
            JumpStatements1,
            JumpStatements2,
            % DeclarationStatements1,
            DeclarationStatements2,
            AssemblerStatements
```

```
END Statements
```

B.31 state-transformer.pvs

```
% $Id: state-transformer.pvs,v 1.61.6.1.4.2 2008/05/14 14:25:00 tews Exp $
%
% Author: Hendrik Tews
% Author: Marcus Voelp
% Author: Tjark Weber, (c) 2007 Radboud University
%
% Lifting and composition of state transformers.
```

```
Transformer_Super_Embedding_Expr[State, Data : Type] : THEORY
% map expression and statement transformers into super transformers
% utility lemmas to break up the super transformers
```

```
BEGIN
```

```
  IMPORTING Super_Embedding
```

```
  s : VAR State
  expr : VAR [State -> ExprResult[State, Data]]
```

```
  expr_2_super(expr)(s) : SuperResult[State] =
    expr_2_super_res(expr(s))
```


B PVS Theory Sources

```
% proof status :-)
ok_expr_2_super : LEMMA
  OK?(expr_2_super(expr)(s)) = OK?(expr(s))

% proof status :-)
has_next_state_expr_2_super : LEMMA
  has_next_state(expr_2_super(expr)(s)) = has_next_state(expr(s))

% state_expr_2_super_TCC1

% state_expr_2_super_TCC2

% proof status :-)
% state_expr_2_super_TCC1 : TCC Obligation

% proof status :-)
% state_expr_2_super_TCC2 : TCC Obligation

% proof status :-)
state_expr_2_super : LEMMA
  has_next_state(expr(s)) implies
  state(expr_2_super(expr)(s)) = state(expr(s))

End Transformer_Super_Embedding_Expr

Transformer_Super_Embedding_Stmt[State : Type] : THEORY
% map expression and statement transformers into super transformers
% utility lemmas to break up the super transformers
BEGIN
  IMPORTING Super_Embedding

  s : VAR State
  stmt : VAR [State -> StmtResult[State]]

  stmt_2_super(stmt)(s) : SuperResult[State] =
    stmt_2_super_res(stmt(s))

% proof status :-)
ok_stmt_2_super : LEMMA
  OK?(stmt_2_super(stmt)(s)) = OK?(stmt(s))

% proof status :-)
has_next_state_stmt_2_super : LEMMA
  has_next_state(stmt_2_super(stmt)(s)) = has_next_state(stmt(s))

% state_stmt_2_super_TCC1

% state_stmt_2_super_TCC2

% proof status :-)
% state_stmt_2_super_TCC1 : TCC Obligation
```

```

% proof status :-)
% state_stmt_2_super_TCC2 : TCC Obligation

% proof status :-)
state_stmt_2_super : LEMMA
  has_next_state(stmt(s)) implies
    state(stmt_2_super(stmt)(s)) = state(stmt(s))

End Transformer_Super_Embedding_Stmt

Transformer_Super_Embedding : Theory
Begin
  Importing Transformer_Super_Embedding_Stmt, Transformer_Super_Embedding_Expr

End Transformer_Super_Embedding

State_Transformer_Lift_Expr[State, Data : Type] : THEORY
BEGIN

  IMPORTING Transformer_Super_Embedding

  s : VAR State

  % lift data to ExprResult
  ok_result(data : Data)(s) : ExprResult[State, Data] =
    OK(s, data)

  % proof status :-)
  ok_result_ok : LEMMA Forall(data : Data) :
    OK?(ok_result(data)(s))

  % proof status :-)
  % ok_result_data_TCC1 : TCC Obligation

  % proof status :-)
  ok_result_data : LEMMA Forall (data : Data) :
    data(ok_result(data)(s)) = data

  % proof status :-)
  ok_result_state : Lemma Forall (data : Data) :
    state(ok_result(data)(s)) = s

  % proof status :-)
  expr_2_super_ok_result : LEMMA Forall(data : Data) :
    OK?(expr_2_super(ok_result(data))(s))

  % lift a constant of type "lift[Data]" to "ExprResult[State, Data]"
  ok_lift(d : lift[Data])(s) : ExprResult[State, Data] =
    CASES d OF
      bottom : Fatal,

```

```

    up(b) : OK(s, b)
ENDCASES

% proof status :-)
ok_lift_ok : LEMMA Forall(d : lift[Data]) :
  up?(d) implies ok_lift(d)(s) = OK(s, down(d))

% lift exception to ExprResult
exception_result(ex : Exception_type)(s) : ExprResult[State, Data] =
  Exception(ex, s)

fatal_result(s) : ExprResult[State, Data] =
  Fatal

% proof status :-)
has_next_state_ok_result : Lemma Forall (data : Data) :
  has_next_state(ok_result(data)(s))

% proof status :-)
has_next_state_fatal_result : Lemma
  Not has_next_state(fatal_result(s))

% proof status :-)
expr_2_super_fatal_result : Lemma
  expr_2_super(fatal_result)(s) = Bottom

% type abbreviations
ET : TYPE = [State -> ExprResult[State, Data]]

END State_Transformer_Lift_Expr

State_Transformer_Lift_Stmt[State : Type] : THEORY
BEGIN

  IMPORTING Transformer_Super_Embedding

  s : VAR State

  % define lifting for statements

  % "lift" lifts a function "stmt"
  % [State -> StmtResult[State]]
  % to
  % [StmtResult[State] -> StmtResult[State]]
  %
  % If the argument is OK, then "stmt" is applied, else "stmt" is skipped. Note
  % that input and output type must be the same.

  stmt : VAR [State -> StmtResult[State]]
  sres : VAR StmtResult[State]

```

```

lift(stmt)(sres) : StmtResult[State] =
  CASES sres OF
    OK(state) : stmt(state)
  ELSE sres
ENDCASES

% "lift_destructor" lifts a function "stmt"
% [State -> StmtResult[State]]
% to
% [StmtResult[State] -> StmtResult[State]]
%
% If the argument is OK, Break, Continue, or Return, then "stmt" is applied,
% else "stmt" is skipped. If "stmt" returns an OK state, the previous result
% constructor (Break/Continue/Return) is then used to replace the OK
% constructor. (Useful for C++ destructors, which may need to be called
% after a break/continue/return statement.) Note that input and output type
% must be the same.

% TODO: what about exceptions?

lift_destructor(stmt)(sres) : StmtResult[State] =
  CASES sres OF
    OK(state) : stmt(state),
    Break(state) :
      CASES stmt(state) OF
        OK(newstate) : Break(newstate)
      ELSE stmt(state)
      ENDCASES,
    Continue(state) :
      CASES stmt(state) OF
        OK(newstate) : Continue(newstate)
      ELSE stmt(state)
      ENDCASES,
    Return(state) :
      CASES stmt(state) OF
        OK(newstate) : Return(newstate)
      ELSE stmt(state)
      ENDCASES
  ELSE sres
ENDCASES

% switch results
switch_result(n : int)(s) : StmtResult[State] =
  Switch(s, n)

default_result(s) : StmtResult[State] =
  Default(s)

hang_result(s) : StmtResult[State] =
  Hang

return_result(s) : StmtResult[State] =

```

B PVS Theory Sources

```
Return(s)

% type abbreviations
ST : TYPE = [State -> StmtResult[State]]

END State_Transformer_Lift_Stmt

State_Transformer_Lift : THEORY
BEGIN
  IMPORTING State_Transformer_Lift_Expr, State_Transformer_Lift_Stmt

END State_Transformer_Lift

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% Composition of Statements and Expressions
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% The composition operator ## will be heavily overloaded.
% we artificially split the different definitions into
% several theories, such that eg show-expanded-sequent will
% tell which of the many ## has been used.

Composition_Simple_Statement_Simple_Statement[State : Type] : THEORY
BEGIN

  IMPORTING State_Transformer_Lift

  s : VAR State

  stmt_1 : VAR [State -> StmtResult[State]]
  stmt_2 : VAR [State -> StmtResult[State]]

  % Note that ## is left-associative, which is important in cases like
  % (s1 ## s2) ## catch_continue
  % vs.
  % s1 ## (s2 ## catch_continue)
  % where the treatment of a "Continue" raised in s1 depends on the
  % parentheses!

  ##(stmt_1, stmt_2)(s) : StmtResult[State] =
    lift(stmt_2)(stmt_1(s))

END Composition_Simple_Statement_Simple_Statement

Composition_Simple_Statement_Lifted_Statement[State : Type] : THEORY
BEGIN
```

IMPORTING StmtResult

s : **VAR** State
 stmt_1 : **VAR** [State → StmtResult[State]]
 lift_stmt_2 : **VAR** [StmtResult[State] → StmtResult[State]]

% Example: whilebody ## catch_continue
 ##(stmt_1, lift_stmt_2)(s) : StmtResult[State] =
 lift_stmt_2(stmt_1(s))

END Composition_Simple_Statement_Lifted_Statement

Composition_Lifted_Statement_Simple_Statement[State : **Type**] : **THEORY**
BEGIN

IMPORTING State_Transformer_Lift

sres : **VAR** StmtResult[State]
 lift_stmt_1 : **VAR** [StmtResult[State] → StmtResult[State]]
 stmt_2 : **VAR** [State → StmtResult[State]]

% Example: (case 1) ## (if ...)
 ##(lift_stmt_1, stmt_2)(sres) : StmtResult[State] =
 lift(stmt_2)(lift_stmt_1(sres))

END Composition_Lifted_Statement_Simple_Statement

Composition_Lifted_Statement_Lifted_Statement[State : **Type**] : **THEORY**
BEGIN

IMPORTING StmtResult

sres : **VAR** StmtResult[State]
 lift_stmt_1 : **VAR** [StmtResult[State] → StmtResult[State]]
 lift_stmt_2 : **VAR** [StmtResult[State] → StmtResult[State]]

% Example: ((case 1) ## ...) ## (case 2)
 ##(lift_stmt_1, lift_stmt_2)(sres) : StmtResult[State] =
 lift_stmt_2(lift_stmt_1(sres))

END Composition_Lifted_Statement_Lifted_Statement

Composition_Simple_Expression_Data_Expression[State, Data1, Data2 : **Type**] :
THEORY
BEGIN

IMPORTING Transformer_Super_Embedding

s : **VAR** State

B PVS Theory Sources

```
expr : VAR [State -> ExprResult[State, Data1]]
fexpr : VAR [Data1 -> [State -> ExprResult[State, Data2]]]

% expression ## (lambda data: expression)

##(expr, fexpr)(s) : ExprResult[State, Data2] =
  CASES expr(s) OF
    OK(state, data) : fexpr(data)(state),
    Exception(ex_type, state) : Exception(ex_type, state),
    Fatal : Fatal,
    Hang : Hang
    % TODO: other expression results?
  ENDCASES

% super_comp_expr_expr_TCC1

% super_comp_expr_expr_TCC1

% proof status :- )
% super_comp_expr_expr_TCC1 : TCC Obligation

% proof status :- )
% super_comp_expr_expr_TCC2 : TCC Obligation

% proof status :- )
super_comp_expr_expr : Lemma
  expr_2_super(expr ## fexpr)(s) =
    CASES expr_2_super(expr)(s) OF
      OK(state) : expr_2_super(fexpr(data(expr(s))))
                    (state(expr_2_super(expr)(s)))
      ELSE expr_2_super(expr)(s)
    ENDCASES

% deprecated now, but kept for compatibility
eval_if_ok(expr, fexpr) : MACRO [State -> ExprResult[State, Data2]] =
  (expr ## fexpr)

END Composition_Simple_Expression_Data_Expression
```

Composition_Simple_Expression_Simple_Expression[State, Data1, Data2 : **Type**] :

THEORY

BEGIN

IMPORTING Composition_Simple_Expression_Data_Expression

s : **VAR** State

expr1 : **VAR** [State -> ExprResult[State, Data1]]

expr2 : **VAR** [State -> ExprResult[State, Data2]]

% expression ## expression (forgetful composition)

```

##(expr1, expr2)(s) : ExprResult[State, Data2] =
  (expr1 ## (lambda (d : Data1): expr2))(s)

% proof status :-)
comp_expr_forget_expr : LEMMA
  (expr1 ## expr2)(s) =
    CASES expr1(s) OF
      OK(state, data) : expr2(state),
      Exception(ex_type, state) : Exception(ex_type, state),
      Fatal : Fatal,
      Hang : Hang
      % TODO: other expression results?
    ENDCASES

% proof status :-)
super_comp_expr_forget_expr : LEMMA
  expr_2_super(expr1 ## expr2)(s) =
    CASES expr_2_super(expr1)(s) OF
      OK(state) : expr_2_super(expr2)(state)
      ELSE expr_2_super(expr1)(s)
    ENDCASES

% proof status :-)
% comp_simple_expr_simple_expr_TCC1 : TCC Obligation

% proof status :-)
comp_simple_expr_simple_expr : Lemma
  OK?(expr1(s)) Implies
    expr2(state(expr1(s))) = (expr1 ## expr2)(s)

% proof status :-)
% comp_simple_expr_simple_expr_ok_TCC1 : TCC Obligation

% proof status :-)
comp_simple_expr_simple_expr_ok : Lemma
  OK?((expr1 ## expr2)(s)) Implies
    OK?(expr2(state(expr1(s))))

% proof status :-)
% comp_simple_expr_simple_expr_data_TCC1 : TCC Obligation

% proof status :-)
comp_simple_expr_simple_expr_data : Lemma
  OK?((expr1 ## expr2)(s)) Implies
    data(expr2(state(expr1(s)))) = data((expr1 ## expr2)(s))

% proof status :-)
% comp_simple_expr_simple_expr_state_TCC1 : TCC Obligation

% proof status :-)
comp_simple_expr_simple_expr_state : Lemma
  OK?((expr1 ## expr2)(s)) Implies

```


B PVS Theory Sources

```
state(expr2(state(expr1(s)))) = state((expr1 ## expr2)(s))
```

```
END Composition_Simple_Expression_Simple_Expression
```

```
Composition_Simple_Expression_Data_Statement[State, Data : Type] :
```

```
  THEORY
```

```
  BEGIN
```

```
    IMPORTING ExprResult, StmtResult
```

```
    s : VAR State
```

```
    expr : VAR [State -> ExprResult[State, Data]]
```

```
    fstmt : VAR [Data -> [State -> StmtResult[State]]]
```

```
    % expression ## (lambda data: statement)
```

```
    ##(expr, fstmt)(s) : StmtResult[State] =
```

```
      CASES expr(s) OF
```

```
        OK(state, data) : fstmt(data)(state),
```

```
        Exception(ex_type, state) : Exception(ex_type, state),
```

```
        Fatal : Fatal,
```

```
        Hang : Hang
```

```
        % TODO: other expression results?
```

```
      ENDCASES
```

```
END Composition_Simple_Expression_Data_Statement
```

```
Composition_Simple_Statement_Simple_Expression[State, Data : Type] :
```

```
  THEORY
```

```
  BEGIN
```

```
    IMPORTING StmtResult, ExprResult
```

```
    s : VAR State
```

```
    stmt : VAR [State -> StmtResult[State]]
```

```
    expr : VAR [State -> ExprResult[State, Data]]
```

```
    % define lifting for expressions
```

```
    sres : VAR StmtResult[State]
```

```
    lift(expr)(sres) : ExprResult[State, Data] =
```

```
      CASES sres OF
```

```
        OK(state) : expr(state),
```

```
        Hang : Hang,
```

```
        Exception(ex_type, state) : Exception(ex_type, state)
```

```
      ELSE
```

```
        Fatal
```

```
      ENDCASES;
```

```

% Example: stmt ## read_data(...)
##(stmt, expr)(s) : ExprResult[State, Data] =
  lift(expr)(stmt(s))

% useful for Hoare logic, to append expressions in the postcondition
% (e.g. read_data(...)) to the program

% comp_simple_stmt_simple_expr_TCC1

% proof status :-)
% comp_simple_stmt_simple_expr_TCC1 : TCC Obligation

% proof status :-)
comp_simple_stmt_simple_expr : LEMMA
  OK?(stmt(s)) implies
    expr(state(stmt(s))) = (stmt ## expr)(s)

% proof status :-)
% comp_simple_stmt_simple_expr_ok_TCC1 : TCC Obligation

% proof status :-)
comp_simple_stmt_simple_expr_ok : Lemma
  OK?((stmt ## expr)(s)) Implies
    OK?(expr(state(stmt(s))))

% proof status :-)
% comp_simple_stmt_simple_expr_data_TCC1 : TCC Obligation

% proof status :-)
comp_simple_stmt_simple_expr_data : Lemma
  OK?((stmt ## expr)(s)) Implies
    data(expr(state(stmt(s)))) = data((stmt ## expr)(s))

% proof status :-)
% comp_simple_stmt_simple_expr_state_TCC1 : TCC Obligation

% proof status :-)
comp_simple_stmt_simple_expr_state : Lemma
  OK?((stmt ## expr)(s)) Implies
    state(expr(state(stmt(s)))) = state((stmt ## expr)(s))

```

END Composition_Simple_Statement_Simple_Expression

Composition_Simple_Statement_Lifted_Expression[State, Data : **Type**] :

THEORY

BEGIN

IMPORTING StmtResult, ExprResult

s : **VAR** State

stmt : **VAR** [State \rightarrow StmtResult[State]]

B PVS Theory Sources

```
lift_expr : VAR [StmtResult[State] -> ExprResult[State, Data]]
```

```
% Example: function_body ## catch_return  
##(stmt, lift_expr)(s) : ExprResult[State, Data] =  
  lift_expr(stmt(s))
```

```
END Composition_Simple_Statement_Lifted_Expression
```

```
Composition_Lifted_Statement_Simple_Expression[State, Data : Type] :  
  THEORY  
BEGIN
```

```
  IMPORTING Composition_Simple_Statement_Simple_Expression
```

```
  sres : VAR StmtResult[State]  
  lift_stmt_1 : VAR [StmtResult[State] -> StmtResult[State]]  
  expr_2 : VAR [State -> ExprResult[State, Data]]
```

```
  ##(lift_stmt_1, expr_2)(sres) : ExprResult[State, Data] =  
    lift(expr_2)(lift_stmt_1(sres))
```

```
END Composition_Lifted_Statement_Simple_Expression
```

```
Composition_Lifted_Statement_Lifted_Expression[State, Data : Type] :  
  THEORY  
BEGIN
```

```
  IMPORTING Composition_Simple_Statement_Simple_Expression
```

```
  sres : VAR StmtResult[State]  
  lift_stmt : VAR [StmtResult[State] -> StmtResult[State]]  
  lift_expr : VAR [StmtResult[State] -> ExprResult[State, Data]]
```

```
  ##(lift_stmt, lift_expr)(sres) : ExprResult[State, Data] =  
    lift_expr(lift_stmt(sres))
```

```
END Composition_Lifted_Statement_Lifted_Expression
```

```
Statement_Composition_Rewrites[State: Type] : THEORY  
BEGIN
```

```
  IMPORTING Composition_Simple_Statement_Simple_Statement,  
    Composition_Simple_Statement_Lifted_Statement,  
    Composition_Lifted_Statement_Simple_Statement,  
    Composition_Lifted_Statement_Lifted_Statement
```

```
  s : VAR State  
  sres : VAR StmtResult[State]  
  stmt_1 : VAR [State -> StmtResult[State]]
```

```

stmt_2 : Var [State -> StmtResult[State]]
stmt_3 : Var [State -> StmtResult[State]]
lift_stmt_1 : VAR [StmtResult[State] -> StmtResult[State]]
lift_stmt_2 : VAR [StmtResult[State] -> StmtResult[State]]
lift_stmt_3 : VAR [StmtResult[State] -> StmtResult[State]]

% For symbolic evaluation, we want a composition operator that associates
% to the right: e.g.
% ((s1 ## s2) ## catch_continue)(s)
% must be rewritten to
% (s1 ## (lift(s2) ## catch_continue))(s)
% before evaluation can be performed, starting with s1.
%
% With 3 statements, each one either lifted or not, there are 8 cases to
% consider.
%
% Essentially: when stmt_2 is not lifted, but stmt_3 is, then stmt_2 must be
% lifted.

% proof status :-)
composition_assoc_rewrite_1 : LEMMA
  (stmt_1 ## (stmt_2 ## stmt_3)) =
  ((stmt_1 ## stmt_2) ## stmt_3)

% proof status :-)
composition_assoc_rewrite_2 : LEMMA
  (stmt_1 ## (lift(stmt_2) ## lift_stmt_3)) =
  ((stmt_1 ## stmt_2) ## lift_stmt_3)

% proof status :-)
composition_assoc_rewrite_3 : LEMMA
  (stmt_1 ## (lift_stmt_2 ## stmt_3)) =
  ((stmt_1 ## lift_stmt_2) ## stmt_3)

% proof status :-)
composition_assoc_rewrite_4 : LEMMA
  (stmt_1 ## (lift_stmt_2 ## lift_stmt_3)) =
  ((stmt_1 ## lift_stmt_2) ## lift_stmt_3)

% proof status :-)
composition_assoc_rewrite_5 : LEMMA
  (lift_stmt_1 ## (stmt_2 ## stmt_3)) =
  ((lift_stmt_1 ## stmt_2) ## stmt_3)

% proof status :-)
composition_assoc_rewrite_6 : LEMMA
  equalities[[StmtResult[State] -> StmtResult[State]]].= (
    (lift_stmt_1 ## (lift(stmt_2) ## lift_stmt_3)),
    ((lift_stmt_1 ## stmt_2) ## lift_stmt_3))

% proof status :-)
composition_assoc_rewrite_7 : LEMMA

```

```

(lift_stmt_1 ## (lift_stmt_2 ## stmt_3)) =
((lift_stmt_1 ## lift_stmt_2) ## stmt_3)

% proof status :-)
composition_assoc_rewrite_8 : LEMMA
  equalities[[StmtResult[State] -> StmtResult[State]]].= (
    (lift_stmt_1 ## (lift_stmt_2 ## lift_stmt_3)),
    ((lift_stmt_1 ## lift_stmt_2) ## lift_stmt_3))

END Statement_Composition_Rewrites

Expression_Composition_Rewrites[State, Data1, Data2 : Type] : THEORY
BEGIN

  IMPORTING Composition_Simple_Expression_Simple_Expression

  s : VAR State
  expr : VAR [State -> ExprResult[State, Data1]]
  fexpr : VAR [Data1 -> [State -> ExprResult[State, Data2]]]

  % proof status :-)
  comp_eval_if_ok_fexpr : LEMMA
    OK?(expr(s)) implies
      (expr ## fexpr)(s) = (expr ## fexpr(data(expr(s))))(s)

END Expression_Composition_Rewrites

Ok_Result_Rewrite[State, Data1, Data2 : Type] : Theory
Begin

  Importing State_Transformer_Lift,
    Composition_Simple_Expression_Data_Statement,
    Expression_Composition_Rewrites,
    Composition_Lifted_Statement_Simple_Expression

  s : Var State
  data : Var Data1
  data2 : Var Data2
  q : Var [State -> ExprResult[State, Data2]]
  states : Var PRED[State]

  % ok_result
  %-----
  % proof status :-)
  ok_result_q_ok : Lemma
    OK?[State, Data1]((q ## ok_result(data))(s)) = OK?(q(s))

  % proof status :-)

```

```

% ok_result_q_state_TCC1 : TCC Obligation

% proof status :-)
% ok_result_q_state_TCC2 : TCC Obligation

% proof status :-)
ok_result_q_state : Lemma
  OK?(q(s)) Implies
    state[State, Data1]((q ## ok_result(data))(s)) = state(q(s))

% proof status :-)
% ok_result_q_data_TCC1 : TCC Obligation

% proof status :-)
ok_result_q_data : Lemma
  OK?(q(s)) Implies
    data((q ## ok_result(data))(s)) = data

% proof status :-)
ok_result_elimination_1 : Lemma
  (ok_result[State, Data1](data) ## q) = q

End Ok_Result_Rewrite

Transformer_Invariant[State : Type] : Theory
Begin
  IMPORTING Super_Result_Util

  transformers : Var PRED[[State -> SuperResult[State]]]
  states : Var PRED[State]

% proof status :-)
% result_pred_TCC1 : TCC Obligation

result_pred(states) : PRED[SuperResult[State]] =
  Lambda(res : SuperResult[State]) :
    has_next_state(res) Implies states(state(res))

% invariants for state transformers
transformer_invariant?(states, transformers) : bool =
  Forall(s : State, q : [State -> SuperResult[State]]) :
    states(s) AND transformers(q) IMPLIES
      result_pred(states)(q(s))

% proof status :-)
% super_transformer_invariant_next_ok_TCC1 : TCC Obligation

% invariant lemma
% proof status :-)
super_transformer_invariant_next_ok : Lemma

```

```

Forall(s : State, q : [State -> SuperResult[State]]) :
  transformer_invariant?(states, transformers) AND
  states(s) AND transformers(q) AND
  has_next_state(q(s))
IMPLIES
  states(state(q(s)))

% Monotonicity
% proof status :- )
transformer_invariant_mono_transformers : Lemma
Forall(transformers_1, transformers_2 :
  PRED[[State -> SuperResult[State]]]) :
  subset?(transformers_1, transformers_2) AND
  transformer_invariant?(states, transformers_2)
IMPLIES
  transformer_invariant?(states, transformers_1)

% union of transformers
% proof status :- )
transformer_invariant_union_transformers : Lemma
Forall(transformers_1, transformers_2 :
  PRED[[State -> SuperResult[State]]]) :
  transformer_invariant?(states, transformers_1) AND
  transformer_invariant?(states, transformers_2)
IMPLIES
  transformer_invariant?(states, union(transformers_1, transformers_2))

% the full set is always an invariant
% proof status :- )
transformer_invariant_truth : Lemma
  transformer_invariant?(fullset[State], transformers)

% proof status :- )
transformer_invariant_all_transformers : Lemma
  (Forall (q : [State -> SuperResult[State]]) : transformers(q) Implies
    transformer_invariant?(states, singleton(q)))
Implies
  transformer_invariant?(states, transformers)

% all transformers return OK
transformers_ok?(states, transformers) : bool =
  Forall(s : State, q : [State -> SuperResult[State]]) :
    states(s) AND transformers(q)
IMPLIES
  OK?(q(s))

% access lemma
% proof status :- )
super_transformers_ok_ok : Lemma
Forall(s : State, q : [State -> SuperResult[State]]) :
  transformers_ok?(states, transformers) AND

```

```

states(s) AND transformers(q) IMPLIES
  OK?(q(s))

% Monotonicity
% proof status :-)
transformers_ok_mono_transformers : Lemma
  Forall(transformers_1, transformers_2 :
    PRED[[State -> SuperResult[State]]]) :
    subset?(transformers_1, transformers_2) AND
    transformers_ok?(states, transformers_2)
    IMPLIES
      transformers_ok?(states, transformers_1)

% proof status :-)
transformers_ok_mono_states : Lemma
  Forall(states_1, states_2 : PRED[State]) :
    subset?(states_1, states_2) AND
    transformers_ok?(states_2, transformers)
    IMPLIES
      transformers_ok?(states_1, transformers)

% lemma for union
% proof status :-)
transformers_ok_union_transformers : Lemma
  Forall(transformers_1, transformers_2 :
    PRED[[State -> SuperResult[State]]]) :
    transformers_ok?(states, transformers_1) AND
    transformers_ok?(states, transformers_2)
    IMPLIES
      transformers_ok?(states, union(transformers_1, transformers_2))

% proof status :-)
transformers_ok_all_transformers : Lemma
  (Forall (q : [State -> SuperResult[State]]) : transformers(q) Implies
    transformers_ok?(states, singleton(q)))
  Implies
    transformers_ok?(states, transformers)

```

End Transformer_Invariant

```

Transformer_Invariant_2[State, Data : Type] : Theory
% express some of above lemmas for non-super transformers
Begin
  Importing Transformer_Invariant, Transformer_Super_Embedding

  transformers : Var PRED[[State -> SuperResult[State]]]
  states : Var PRED[State]

% proof status :-)
% expr_transformer_invariant_next_ok_TCC1 : TCC Obligation

```



```

% proof status :-)
expr_transformer_invariant_next_ok : Lemma
  Forall(s : State, q : [State -> ExprResult[State, Data]]) :
    transformer_invariant?(states, transformers) AND
    states(s) AND transformers(expr_2_super(q)) AND
    has_next_state(q(s))
    IMPLIES
      states(state(q(s)))

% proof status :-)
expr_transformers_ok_ok : Lemma
  Forall(s : State, q : [State -> ExprResult[State, Data]]) :
    transformers_ok?(states, transformers) AND
    states(s) AND transformers(expr_2_super(q)) IMPLIES
      OK?(q(s))

% proof status :-)
% expr_result_pred_next_state_TCC1 : TCC Obligation

% proof status :-)
expr_result_pred_next_state : Lemma
  Forall(s : State, q : [State -> ExprResult[State, Data]]) :
    has_next_state(q(s)) And
    result_pred(states)(expr_2_super(q)(s)) Implies
      states(state(q(s)))

End Transformer_Invariant_2

Transformer_Invariant_3[State, Data1, Data2 : Type] : Theory
Begin
  Importing Transformer_Invariant_2, Expression_Composition_Rewrites

  states : Var PRED[State]

  expr1 : Var [State -> ExprResult[State, Data1]]
  expr2 : Var [State -> ExprResult[State, Data2]]
  fexpr : Var [Data1 -> [State -> ExprResult[State, Data2]]]

  % proof status :-)
  fexpr_composition_transformers_ok : Lemma
    Forall (P : PRED[Data1]) :
      transformers_ok?(states, singleton(expr_2_super(expr1))) And
      transformer_invariant?(states, singleton(expr_2_super(expr1))) And
      (Forall (s : (states)) : OK?(expr1(s)) Implies P(data(expr1(s)))) And
      (Forall (d : (P)) : transformers_ok?(states, singleton(expr_2_super(fexpr(d)))))
      Implies
        transformers_ok?(states, singleton(expr_2_super(expr1 ## fexpr)))

  % proof status :-)

```

```

expr_composition_transformers_ok : Lemma
  transformers_ok?(states, singleton(expr_2_super(expr1))) And
  transformer_invariant?(states, singleton(expr_2_super(expr1))) And
  transformers_ok?(states, singleton(expr_2_super(expr2)))
Implies
  transformers_ok?(states, singleton(expr_2_super(expr1 ## expr2)))

% proof status :-)
fexpr_composition_transformer_invariant : Lemma
Forall (P : PRED[Data1]) :
  transformer_invariant?(states, singleton(expr_2_super(expr1))) And
  (Forall (s : (states)) : OK?(expr1(s)) Implies P(data(expr1(s)))) And
  (Forall (d : (P)) : transformer_invariant?(states, singleton(expr_2_super(fexpr(d))))))
Implies
  transformer_invariant?(states, singleton(expr_2_super(expr1 ## fexpr)))

% proof status :-)
expr_composition_transformer_invariant : Lemma
  transformer_invariant?(states, singleton(expr_2_super(expr1))) And
  transformer_invariant?(states, singleton(expr_2_super(expr2)))
Implies
  transformer_invariant?(states, singleton(expr_2_super(expr1 ## expr2)))

End Transformer_Invariant_3

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% State_Transformer: imports everything
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

State_Transformer : Theory
BEGIN

IMPORTING
  % Transformer_Super_Embedding,
  % State_Transformer_Lift,
  % Composition_Simple_Statement_Simple_Statement,
  % Composition_Simple_Statement_Lifted_Statement,
  % Composition_Lifted_Statement_Simple_Statement,
  % Composition_Lifted_Statement_Lifted_Statement,
  % Composition_Simple_Expression_Data_Expression,
  % Composition_Simple_Expression_Simple_Expression,
  % Composition_Simple_Expression_Data_Statement,
  % Composition_Simple_Statement_Simple_Expression,
  Composition_Simple_Statement_Lifted_Expression,
  Composition_Lifted_Statement_Lifted_Expression,
  Statement_Composition_Rewrites,
  % Expression_Composition_Rewrites,
  Ok_Result_Rewrite,
% Make_Transformer,

```

```
% Transformer_Invariant,  
%Transformer_Invariant_2,  
Transformer_Invariant_3
```

END State_Transformer

B.32 types.pvs

```
% $Id: types.pvs,v 1.16.4.2.2.4.2.4 2008/05/14 14:25:01 tews Exp $  
%  
% Author: Tjark Weber; Hendrik Tews; Marcus Völz et al.  
% (c) 2007 Radboud University  
%  
% A formalization of C++ types.
```

```
% Section numbers in comments throughout this file refer to  
% "Programming languages – C++", ISO/IEC 14882:1998(E)
```

```
% Open Issues / To Do  
%=====  
% Open issue: can we reap benefit from this?  
% 3.9.1 pt 7 representation of value of integral through pure binary system  
% integral value = Sum (0 <= p <(=?=) n) b[p] * 2^p with bits b[p]  
%  
% Can we conclude from this that the max_value_bits are the bits in the  
% value representation, or more precisely that:  
% [bits in object representation] >= [bits in value representation]  
% size(uidt(unsigned)) * bits_per_byte >= max_value_bits(unsigned)
```

```
% Open issue:  
% The underlying data model with its to_byte/from_byte encoding functions  
% does not really allow to model individual bits which are in a type's  
% object representation, but NOT in its value representation.
```

```
% Open issue: const, volatile  
% Is it wise to formalise const, volatile as Cpp_Type; there seems to be  
% no easy way to define that expressions on cv(t) behave like expressions  
% on t. Should we use typ(c, v : bool) instead? Note that this implies  
% dt(t) = dt(volatile(t)) and thus that hidden bits cannot indicate a  
% volatile access.
```

```
% MV: To do:  
% – classes are not CV qualified by their members however a cv class  
% behave like all nonstatic members being cv qualified ; use the semantics  
% compiler to generate appropriate member types  
% – cv(array) = array(cv)  
% – cv void only as function return parameter  
% – cv function affects cv of this => use semantics compiler to type this
```

```
% => define dt(*) only for interpreted types with domain  
% Number x Address x Semantics Pointer
```

```

% -> generic handling of dt
% -> generic behaviour of const, volatile
% -> Define CV_Arithmetic
% -> dt_const(t) = dt(t)
% -> dt_volatile(t) = dt(t) except for hidden bits

% Not modelled (yet):
% - Pointer-to-function
% - const/volatile specifiers
% - float, double, long double
% - Conversion between integral types and bitsets
% - Conversion between integral types and enumeration types

% - wird kein every mehr für Datentypen definiert?
% - dt_exists TCC durch Instanziierung mit konkretem Modell beweisen
% 3.9.1.1: "In any particular implementation, a plain char object can
% take on either the same values as a signed char or an
% unsigned char; which one is implementation-defined." -- We
% take this to also mean that the value representation is the
% same for char and (either) signed char or unsigned char. (Why?)

% Todo:
% - should we encode arg_type as list[Cpp_Type]
% or better as members, below[members] -> CppType
% - variable arguments
% - semantic compiler generates var_args object to contain arguments
% - var_args object is passed to function
% - default values
% - semantics compiler initialises all uninitialised values with defaults.

% See abstract_data.pvs for the underlying data type model.
%
% Naming conventions:
% - theories are named "Cpp_<type>",
% - value ranges are named "range(<type>)",
% - data types are named "dt_<type>",
% where <type> is one of
% - uchar, schar, char,
% - short, int, long, longlong,
% - ushort, uint, ulong, ulonglong,
% - wchar_t,
% - bool,
% - void,
% - pointer,
% - reference.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% Fundamental Types
%
```

B PVS Theory Sources

%%%

% Deviations from the C++ standard:

%

% The underlying data model with its to_byte/from_byte encoding functions

% does not really allow to model individual bits which are in a type's

% object representation, but NOT in its value representation.

% 3.9.1 pt 1 Unsigned Char

%=====

Cpp_uchar : Theory

Begin

Importing finite_sets@finite_sets_minmax[nat, <=], More_More_List_Props

% Alignment

%-----

align_uchar : posnat

% proof status :-)

% range_uchar_TCC1 : TCC Obligation

% Range of values

%-----

range_uchar : non_empty_finite_set[nat]

% (Un)interpreted Datatype

%-----

Importing Interpreted_Data[(range_uchar)]

dt_uchar_exists : **Axiom**

Exists (dt : (pod_data_type?[(range_uchar)])) : **true**

% proof status :-)

% dt_uchar_TCC1 : TCC Obligation

dt_uchar : (pod_data_type?[(range_uchar)])

% 5.3.3 pt 1 "The size_of operator yields the number of bytes in the object

% representation of its operand."

% "size_of(unsigned char) [is] 1"

dt_uchar_size : **Axiom**

size(uidt(dt_uchar)) = 1

% 3.9.1 pt 1: "For unsigned character types, all possible bit patterns of the

% value representation represent numbers."

% (TODO: This seems to follow from the arithmetic-mod-2^n requirement

% (3.9.1 pt 4) already -- maybe we could prove it?!)

dt_uchar_valid : **Axiom**

Forall (l : list[Byte], a : Address):

length(l) = size(uidt(dt_uchar)) **Implies** valid?(uidt(dt_uchar))(l, a)

```

% Bits in the value representation
%-----
max_value_bits_uchar : posnat

% proof status :-)
% value_bitmask_uchar_TCC1 : TCC Obligation

% a bitmask in which all bits are set that are affected by a bit
% operation on this type
value_bitmask_uchar : {l : list[Byte] | length(l) = size(uidt(dt_uchar))}

% C Standard, 5.2.4.2.1: UCHAR_MAX >= 255 (2^8 - 1)
% 3.9.1 pt 4 "Unsigned integers [...] shall obey the laws of arithmetic
% modulo 2^n where n is the number of bits in the value
% representation"
min_max_value_bits_uchar : Axiom
  max_value_bits_uchar >= 8

% 3.9 p4: "For POD types, the value representation is the set of bits in
% the object representation that determines a value [...]"
% 3.9.1 pt1: "For character types, all bits of the object representation
% participate in the value representation."
dt_uchar_bits : Axiom
  max_value_bits_uchar = size(uidt(dt_uchar)) * bits_per_byte

% max_value(uchar) >= 255
max_uchar : Axiom
  max(range_uchar) >= (2^max_value_bits_uchar) - 1

% 3.9.1 pt 4 "Unsigned integers [...] shall obey the laws of arithmetic
% modulo 2^n where n is the number of bits in the value
% representation"
binary_range_uchar : Axiom
  Forall (n : nat) :
    range_uchar(n) IFF n <= max(range_uchar)

% => max_value_bits(uchar) >= min_bits_per_byte
% => max_value(uchar) = 2^bits_per_byte - 1

End Cpp_uchar

% 3.9.1 pt 1 Signed Char
%=====
Cpp_schar : Theory
Begin

  Importing Cpp_uchar, finite_sets@finite_sets_minmax[int, <=]

% Alignment
%-----

```

B PVS Theory Sources

```
% 3.9.1 pt 1 alignment of schar = alignment of uchar
align_schar : posnat = align_uchar

% proof status :-)
% range_schar_TCC1 : TCC Obligation

% Range of values
%-----
range_schar : non_empty_finite_set[int]

% (Un)interpreted Datatype
%-----
Importing Interpreted_Data[(range_schar)]

dt_schar_exists : Axiom
  Exists (dt : (pod_data_type?[(range_schar)])) : true

% proof status :-)
% dt_schar_TCC1 : TCC Obligation

dt_schar : (pod_data_type?[(range_schar)])

% 3.9.1 pt 1 size_of(schar) = size_of(uchar)
dt_schar_size : Axiom
  size(uidt(dt_schar)) = size(uidt(dt_uchar))

% 3.9.1 pt 3 "The range of nonnegative values of a signed integer type
% is the range of the corresponding unsigned integer type,
% and the value representation of each corresponding
% signed / unsigned type shall be the same."
schar_nonneg_is_uchar : Axiom
  Forall (i : (range_schar)) : i >= 0 Implies range_uchar(i)

% proof status :-)
% schar_value_eq_uchar_value_TCC1 : TCC Obligation

schar_value_eq_uchar_value : Axiom
  Forall (i : (range_schar), a : Address) : i >= 0 Implies
    to_byte(dt_schar)(i, a) = to_byte(dt_uchar)(i, a)

max_value_bits_schar : nat

% proof status :-)
% value_bitmask_schar_TCC1 : TCC Obligation

value_bitmask_schar : {l : list[Byte] | length(l) = size(uidt(dt_schar))}

% 3.9 pt 4: "For POD types, the value representation is the set of bits in
% the object representation that determines a value [...]"
% 3.9.1 pt 1: "For character types, all bits of the object representation
% participate in the value representation."
% The highest order bit may however encode the sign of the value.
```

```

% (footnote 49 3.9.1 pt 7)
dt_schar_bits : Axiom
  max_value_bits_schar >= size(uidt(dt_schar)) * bits_per_byte - 1

% C Standard, 5.2.4.2.1: SCHAR_MIN <= -127 (-(2^7 - 1))
% C Standard, 5.2.4.2.1: SCHAR_MAX >= 127 (2^7 - 1)
min_schar : Axiom
  min(range_schar) <= -((2 ^ max_value_bits_schar) - 1)

max_schar : Axiom
  max(range_schar) >= (2 ^ max_value_bits_schar) - 1

schar_binary : Axiom
  Forall (i : int) :
    range_schar(i) IFF (min(range_schar) <= i And i <= max(range_schar))

% => min_value(schar) <= -127
% => max_value(schar) >= 127
% => max_value_bits(schar) >= min_bits_per_byte - 1

```

End Cpp_schar

```

% 3.9.1 pt 1 Char
%=====
Cpp_char : Theory
Begin

  Importing Cpp_schar

  % Alignment
  %-----
  % 3.9.1 pt 1 alignment of char = alignment of uchar
  align_char : posnat = align_uchar

  % Range of values
  %-----
  range_char : non_empty_finite_set[int]

  % 3.9.1.1: "In any particular implementation, a plain char object can
  % take on either the same values as a signed char or an
  % unsigned char; which one is implementation-defined."
  char_is_uchar_or_schar : Axiom
    range_char = range_schar OR range_char = range_uchar

  % (Un)interpreted Datatype
  %-----
  Importing Interpreted_Data[(range_char)]

  dt_char_exists : Axiom
    Exists (dt : (pod_data_type?[(range_char)])) : true

```


B PVS Theory Sources

```
% proof status :-)
% dt_char_TCC1 : TCC Obligation

dt_char : (pod_data_type?[(range_char)])

% 3.9.1 pt 1 size_of(schar) = size_of(uchar)
dt_char_size : Axiom
  size(uidt(dt_char)) = size(uidt(dt_uchar))

max_value_bits_char : nat

% proof status :-)
% value_bitmask_char_TCC1 : TCC Obligation

value_bitmask_char : {l : list[Byte] | length(l) = size(uidt(dt_char))}

% 3.9 pt 4: "For POD types, the value representation is the set of bits in
% the object representation that determines a value [...]"
% 3.9.1 pt 1: "For character types, all bits of the object representation
% participate in the value representation."
% The highest order bit may however encode the sign of the value.
% (footnote 49 3.9.1 pt 7)
dt_char_bits : Axiom
  max_value_bits_char >= size(uidt(dt_char)) * bits_per_byte - 1

% proof status :-)
min_char : Lemma
  min(range_char) <= 0

% proof status :-)
max_char : Lemma
  max(range_char) >= 127

% proof status :-)
char_binary : Lemma
  Forall (i : int) :
    range_char(i) IFF (min(range_char) <= i And i <= max(range_char))
```

End Cpp_char

```
% 3.9.1 pt 2 Signed Short Int
%-----
Cpp_short : Theory
Begin

  Importing Cpp_schar, finite_sets@finite_sets_minmax[int, <=]

  % Alignment
  %-----
  align_short : posnat
```

```

% proof status :-)
% range_short_TCC1 : TCC Obligation

% Range of values
%-----
range_short : non_empty_finite_set[int]

% (Un)interpreted Datatype
%-----
Importing Interpreted_Data[(range_short)]

dt_short_exists : Axiom
  Exists (dt : (pod_data_type?[(range_short)])) : true

% proof status :-)
% dt_short_TCC1 : TCC Obligation

dt_short : (pod_data_type?[(range_short)])

% 3.9.1.2: "There are four signed integer types: signed char,
% short int, int, and long int. In this list, each type
% provides at least as much storage as those preceding it
% in the list." -- We take this to mean that both the value
% range and the size of the object representation must not
% be smaller.
dt_short_size : Axiom
  size(uidt(dt_short)) >= size(uidt(dt_schar))

short_value_supset_schar_value : Axiom
  Forall (i : (range_schar)) : range_short(i)

max_value_bits_short : nat

% proof status :-)
% value_bitmask_short_TCC1 : TCC Obligation

value_bitmask_short : {l : list[Byte] | length(l) = size(uidt(dt_short))}

% C Standard, 5.2.4.2.1: SHRT_MIN <= -32767 (-(2^15 - 1))
min_short : Axiom
  min(range_short) <= -32767

% C Standard, 5.2.4.2.1: SHRT_MAX >= 32767 (2^15 - 1)
max_short : Axiom
  max(range_short) >= 32767

short_binary : Axiom
  FORALL (i: int):
    range_short(i) IFF (min(range_short) <= i And i <= max(range_short))

End Cpp_short

```

```

% 3.9.1 pt 2 Signed Int
%-----
Cpp_int : Theory
Begin

    Importing Cpp_short, finite_sets@finite_sets_minmax[int, <=]

% Alignment
%-----
align_int : posnat

% proof status :-)
% range_int_TCC1 : TCC Obligation

% Range of values
%-----
range_int : non_empty_finite_set[int]

% (Un)interpreted Datatype
%-----
Importing Interpreted_Data[(range_int)]

dt_int_exists : Axiom
    Exists (dt : (pod_data_type?[(range_int)])) : true

% proof status :-)
% dt_int_TCC1 : TCC Obligation

dt_int : (pod_data_type?[(range_int)])

% 3.9.1.2: "There are four signed integer types: signed char,
% short int, int, and long int. In this list, each type
% provides at least as much storage as those preceding it
% in the list." -- We take this to mean that both the value
% range and the size of the object representation must not
% be smaller.
dt_int_size : Axiom
    size(uidt(dt_int)) >= size(uidt(dt_short))

int_value_supset_short_value : Axiom
    Forall (i : (range_short)) : range_int(i)

max_value_bits_int : nat

% proof status :-)
% value_bitmask_int_TCC1 : TCC Obligation

value_bitmask_int : {l : list[Byte] | length(l) = size(uidt(dt_int))}

% C Standard, 5.2.4.2.1: INT_MIN <= -32767 (-(2^15 - 1))
min_int : Axiom

```

```

min(range_int) <= -32767

% C Standard, 5.2.4.2.1: INT_MAX >= 32767 (2^15 - 1)
max_int : Axiom
  max(range_int) >= 32767

int_binary : Axiom
FORALL (i: int):
  range_int(i) IFF (min(range_int) <= i And i <= max(range_int))

```

End Cpp_int

% 3.9.1 pt 2 Signed Long

%-----

Cpp_long : **Theory**

Begin

Importing Cpp_int, finite_sets@finite_sets_minmax[int, <=]

% Alignment

%-----

align_long : posnat

% proof status :-)

% range_long_TCC1 : TCC Obligation

% Range of values

%-----

range_long : non_empty_finite_set[int]

% (Un)interpreted Datatype

%-----

Importing Interpreted_Data[(range_long)]

dt_long_exists : **Axiom**

Exists (dt : (pod_data_type?[(range_long)])) : **true**

% proof status :-)

% dt_long_TCC1 : TCC Obligation

dt_long : (pod_data_type?[(range_long)])

% 3.9.1 pt 2: "There are four signed integer types: signed char,

% short int, int, and long int. In this list, each type

% provides at least as much storage as those preceding it

% in the list." -- We take this to mean that both the value

% range and the size of the object representation must not

% be smaller.

dt_long_size : **Axiom**

size(uidt(dt_long)) >= size(uidt(dt_int))

B PVS Theory Sources

```
long_value_supset_int_value : Axiom
  Forall (i : (range_int)) : range_long(i)

max_value_bits_long : nat

% proof status : -)
% value_bitmask_long_TCC1 : TCC Obligation

value_bitmask_long : {l : list[Byte] | length(l) = size(uidt(dt_long))}

% C Standard, 5.2.4.2.1: LONG_MIN <= -2147483647 (-(2^31 - 1))
min_long : Axiom
  min(range_long) <= -2147483647

% C Standard, 5.2.4.2.1: LONG_MAX >= 2147483647 (2^31 - 1)
max_long : Axiom
  max(range_long) >= 2147483647

long_binary : Axiom
  FORALL (i: int):
    range_long(i) IFF (min(range_long) <= i And i <= max(range_long))
```

End Cpp_long

```
% Signed Long Long
% (not in the mentioned version of the C++ standard)
%-----
```

```
Cpp_longlong : Theory
Begin
```

```
  Importing Cpp_long, finite_sets@finite_sets_minmax[int, <=]
```

```
  % Alignment
```

```
  %-----
  align_longlong : posnat
```

```
  % proof status : -)
```

```
  % range_longlong_TCC1 : TCC Obligation
```

```
  % Range of values
```

```
  %-----
  range_longlong : non_empty_finite_set[int]
```

```
  % (Un)interpreted Datatype
```

```
  %-----
  Importing Interpreted_Data[(range_longlong)]
```

```
  dt_longlong_exists : Axiom
```

```
    Exists (dt : (pod_data_type?[(range_longlong)])) : true
```

```
  % proof status : -)
```

```

% dt_longlong_TCC1 : TCC Obligation

dt_longlong : (pod_data_type?[(range_longlong)])

% Following 3.9.1 pt 2 in principle we define:
dt_longlong_size : Axiom
  size(uidt(dt_longlong)) >= size(uidt(dt_long))

longlong_value_supset_long_value : Axiom
  Forall (i : (range_long)) : range_longlong(i)

max_value_bits_longlong : nat

% proof status :-)
% value_bitmask_longlong_TCC1 : TCC Obligation

value_bitmask_longlong : {l : list[Byte] | length(l) =
  size(uidt(dt_longlong))}

% C Standard, 5.2.4.2.1: LLONG_MIN <= -922337203685477807 (-(2^63 - 1))
min_longlong : Axiom
  min(range_longlong) <= -922337203685477807

% C Standard, 5.2.4.2.1: LLONG_MAX >= 922337203685477807 (2^63 - 1)
max_longlong : Axiom
  max(range_longlong) >= 922337203685477807

longlong_binary : Axiom
  FORALL (i: int):
    range_longlong(i) IFF
      (min(range_longlong) <= i And i <= max(range_longlong))

End Cpp_longlong

% 3.9.1 pt 3 Unsigned Short Integer
%=====
Cpp_ushort : Theory
Begin

  Importing Cpp_short, finite_sets@finite_sets_minmax[nat, <=]

  % Alignment
  %-----
  % 3.9.1 pt 3 unsigned integers shall have the same alignment as the
  % corresponding signed integer
  align_ushort : posnat = align_short

  % proof status :-)
  % range_ushort_TCC1 : TCC Obligation

  % Range of values

```

B PVS Theory Sources

```
%-----  
range_ushort : non_empty_finite_set[nat]  
  
% (Un)interpreted Datatype  
%-----  
Importing Interpreted_Data[(range_ushort)]  
  
dt_ushort_exists : Axiom  
  Exists (dt : (pod_data_type?[(range_ushort)])) : true  
  
% proof status :-)   
% dt_ushort_TCC1 : TCC Obligation  
  
dt_ushort : (pod_data_type?[(range_ushort)])  
  
% 3.9.1 pt 3 unsigned integers shall have the same size as the  
% corresponding signed integer  
dt_ushort_size : Axiom  
  size(uidt(dt_ushort)) = size(uidt(dt_short))  
  
% 3.9.1 pt 3 "The range of nonnegative values of a signed integer type is a  
% subrange of the corresponding unsigned integer type, and the  
% value representation of each corresponding signed / unsigned  
% type shall be the same."  
ushort_value_supset_short_value : Axiom  
  Forall (i : (range_short)) : i >= 0 Implies range_ushort(i)  
  
% proof status :-)   
% ushort_represenation_same_for_positive_short_TCC1 : TCC Obligation  
  
ushort_represenation_same_for_positive_short : Axiom  
  Forall (i : (range_short), a : Address) : i >= 0 Implies  
    to_byte(dt_ushort)(i, a) = to_byte(dt_short)(i, a)  
  
% Bits in the value representation  
%-----  
max_value_bits_ushort : posnat  
  
% proof status :-)   
% value_bitmask_ushort_TCC1 : TCC Obligation  
  
value_bitmask_ushort : {l : list[Byte] | length(l) = size(uidt(dt_ushort))}  
  
% C Standard, 5.2.4.2.1: USHRT_MAX >= 65535 (2^16 - 1)  
max_ushort_bits : Axiom  
  max_value_bits_ushort >= 16  
  
% 3.9.1 pt 4 "Unsigned integers [...] shall obey the laws of arithmetic  
% modulo 2^n where n is the number of bits in the value  
% representation ..."  
binary_range_ushort : Axiom  
  Forall (n : nat) :
```

```
range_ushort(n) IFF n <= max(range_ushort)
```

```
End Cpp_ushort
```

```
% 3.9.1 pt 3 Unsigned Integer
```

```
%=====
```

```
Cpp_uint : Theory
```

```
Begin
```

```
Importing Cpp_int, finite_sets@finite_sets_minmax[nat, <=]
```

```
% Alignment
```

```
%-----
```

```
% 3.9.1 pt 3 unsigned integers shall have the same alignment as the  
% corresponding signed integer
```

```
align_uint : posnat = align_int
```

```
% proof status :—)
```

```
% range_uint_TCC1 : TCC Obligation
```

```
% Range of values
```

```
%-----
```

```
range_uint : non_empty_finite_set[nat]
```

```
% (Un)interpreted Datatype
```

```
%-----
```

```
Importing Interpreted_Data[(range_uint)]
```

```
dt_uint_exists : Axiom
```

```
  Exists (dt : (pod_data_type?[(range_uint)])) : true
```

```
% proof status :—)
```

```
% dt_uint_TCC1 : TCC Obligation
```

```
dt_uint : (pod_data_type?[(range_uint)])
```

```
% 3.9.1 pt 3 unsigned integers shall have the same size as the
```

```
% corresponding signed integer
```

```
dt_uint_size : Axiom
```

```
  size(uidt(dt_uint)) = size(uidt(dt_int))
```

```
% 3.9.1 pt 3 "The range of nonnegative values of a signed integer type is a
```

```
% subrange of the corresponding unsigned integer type, and the
```

```
% value representation of each corresponding signed / unsigned
```

```
% type shall be the same."
```

```
uint_value_supset_int_value : Axiom
```

```
  Forall (i : (range_int)) : i >= 0 Implies range_uint(i)
```

```
% proof status :—)
```

```
% uint_represenation_same_for_positive_int_TCC1 : TCC Obligation
```


B PVS Theory Sources

```

uint_representation_same_for_positive_int : Axiom
  Forall (i : (range_int), a : Address) : i >= 0 Implies
    to_byte(dt_uint)(i, a) = to_byte(dt_int)(i, a)

% Bits in the value representation
%-----
max_value_bits_uint : posnat

% proof status :-)
% value_bitmask_uint_TCC1 : TCC Obligation

value_bitmask_uint : {l : list[Byte] | length(l) = size(uidt(dt_uint))}

% C Standard, 5.2.4.2.1: UINT_MAX >= 65535 (2^16 - 1)
max_uint_bits : Axiom
  max_value_bits_uint >= 16

max_uint : Axiom
  max(range_uint) >= (2^max_value_bits_uint) - 1

% 3.9.1 pt 4 "Unsigned integers [...] shall obey the laws of arithmetic
% modulo 2^n where n is the number of bits in the value
% representation ..."
binary_range_uint : Axiom
  Forall (n : nat) :
    range_uint(n) IFF n <= max(range_uint)

End Cpp_uint

% 3.9.1 pt 3 Unsigned Long
%=====
Cpp_ulong : Theory
Begin

  Importing Cpp_long, finite_sets@finite_sets_minmax[nat, <=]

  % Alignment
  %-----
  % 3.9.1 pt 3 unsigned integers shall have the same alignment as the
  % corresponding signed integer
  align_ulong : posnat = align_long

  % proof status :-)
  % range_ulong_TCC1 : TCC Obligation

  % Range of values
  %-----
  range_ulong : non_empty_finite_set[nat]

  % (Un)interpreted Datatype
  %-----

```

```

Importing Interpreted_Data[(range_ulong)]

dt_ulong_exists : Axiom
  Exists (dt : (pod_data_type?[(range_ulong)])) : true

% proof status :-)
% dt_ulong_TCC1 : TCC Obligation

dt_ulong : (pod_data_type?[(range_ulong)])

% 3.9.1 pt 3 unsigned integers shall have the same size as the
% corresponding signed integer
dt_ulong_size : Axiom
  size(uidt(dt_ulong)) = size(uidt(dt_long))

% 3.9.1 pt 3 "The range of nonnegative values of a signed integer type is a
% subrange of the corresponding unsigned integer type, and the
% value representation of each corresponding signed / unsigned
% type shall be the same."
ulong_value_supset_long_value : Axiom
  Forall (i : (range_long)) : i >= 0 Implies range_ulong(i)

% proof status :-)
% ulong_representation_same_for_positive_long_TCC1 : TCC Obligation

ulong_representation_same_for_positive_long : Axiom
  Forall (i : (range_long), a : Address) : i >= 0 Implies
    to_byte(dt_ulong)(i, a) = to_byte(dt_long)(i, a)

% Bits in the value representation
% -----
max_value_bits_ulong : posnat

% proof status :-)
% value_bitmask_ulong_TCC1 : TCC Obligation

value_bitmask_ulong : {l : list[Byte] | length(l) = size(uidt(dt_ulong))}

% C Standard, 5.2.4.2.1: ULONG_MAX >= 4294967295 (2^32 - 1)
max_ulong_bits : Axiom
  max_value_bits_ulong >= 32

max_ulong : Axiom
  max(range_ulong) >= (2^max_value_bits_ulong) - 1

% 3.9.1 pt 4 "Unsigned integers [...] shall obey the laws of arithmetic
% modulo 2^n where n is the number of bits in the value
% representation ..."
binary_range_ulong : Axiom
  Forall (n : nat) :
    range_ulong(n) IFF n <= max(range_ulong)

```

End Cpp_ulong

% 3.9.1 pt 3 Unsigned Long Long

%=====

Cpp_ulonglong : **Theory**

Begin

Importing Cpp_longlong, finite_sets@finite_sets_minmax[nat, <=]

% Alignment

%-----

% 3.9.1 pt 3 unsigned integers shall have the same alignment as the

% corresponding signed integer

align_ulonglong : posnat = align_longlong

% proof status : -)

% range_ulonglong_TCC1 : TCC Obligation

% Range of values

%-----

range_ulonglong : non_empty_finite_set[nat]

% (Un)interpreted Datatype

%-----

Importing Interpreted_Data[(range_ulonglong)]

dt_ulonglong_exists : **Axiom**

Exists (dt : (pod_data_type?[(range_ulonglong)])) : **true**

% proof status : -)

% dt_ulonglong_TCC1 : TCC Obligation

dt_ulonglong : (pod_data_type?[(range_ulonglong)])

% 3.9.1 pt 3 unsigned integers shall have the same size as the

% corresponding signed integer

dt_ulonglong_size : **Axiom**

size(uidt(dt_ulonglong)) = size(uidt(dt_longlong))

% 3.9.1 pt 3 "The range of nonnegative values of a signed integer type is a

% subrange of the corresponding unsigned integer type, and the

% value representation of each corresponding signed / unsigned

% type shall be the same."

ulonglong_value_supset_longlong_value : **Axiom**

Forall (i : (range_longlong)) : i >= 0 **Implies** range_ulonglong(i)

% proof status : -)

% ulonglong_represenation_same_for_positive_longlong_TCC1 : TCC Obligation

ulonglong_represenation_same_for_positive_longlong : **Axiom**

Forall (i : (range_longlong), a : Address) : i >= 0 **Implies**

to_byte(dt_ulonglong)(i, a) = to_byte(dt_longlong)(i, a)

```

% Bits in the value representation
%-----
max_value_bits_ulonglong : posnat

% proof status :-)
% value_bitmask_ulonglong_TCC1 : TCC Obligation

value_bitmask_ulonglong : {l : list[Byte] | length(l) =
  size(uidt(dt_ulonglong))}

% C Standard, 5.2.4.2.1: ULLONG_MAX >= 18446744073709551615 (2^64 - 1)
max_ulonglong_bits : Axiom
  max_value_bits_ulonglong >= 64

max_ulonglong : Axiom
  max(range_ulonglong) >= (2^max_value_bits_ulonglong) - 1

% 3.9.1 pt 4 "Unsigned integers [...] shall obey the laws of arithmetic
% modulo 2^n where n is the number of bits in the value
% representation ..."
binary_range_ulonglong : Axiom
  Forall (n : nat) :
    range_ulonglong(n) IFF n <= max(range_ulonglong)

```

End Cpp_ulonglong

```

% 3.9 pt 8 Float
%=====
Cpp_float : Theory
Begin

  Importing Extended_Real

  % Alignment
  %-----
  align_float : posnat

  % Value Range
  %-----
  % subset of number so that we can eventually formalise not-a-thing
  % and +-infinity values

  % proof status :-)
  % range_float_TCC1 : TCC Obligation

  range_float : non_empty_finite_set[extended_real]

  % (Un)interpreted Datatype
  %-----
  Importing Interpreted_Data[(range_float)]

```

B PVS Theory Sources

```
dt_float_exists : Axiom
  Exists (dt_float : (pod_data_type?[(range_float)])) : True

% proof status :-)
% dt_float_TCC1 : TCC Obligation

dt_float : (pod_data_type?[(range_float)])

End Cpp_float

% 3.9 pt 8 Double
%=====
Cpp_double : Theory
Begin

  Importing Cpp_float, Extended_Real

% Alignment
%-----
align_double : posnat

% Value Range
%-----
% subset of number so that we can eventually formalise not-a-thing
% and +-infinity values

range_double : non_empty_finite_set[extended_real]

% (Un)interpreted Datatype
%-----
Importing Interpreted_Data[(range_double)]

dt_double_exists : Axiom
  Exists (dt_double : (pod_data_type?[(range_double)])) : True

% proof status :-)
% dt_double_TCC1 : TCC Obligation

dt_double : (pod_data_type?[(range_double)])

% 3.9.1 pt 8 precision float <= double <= long double
% float values is subset of double values is subset of
% long double values
double_precision : Axiom
  Forall (f : (range_float)) : real?(f) Implies range_double(f)

End Cpp_double

% 3.9 pt 8 Long Double
```

```

%=====
Cpp_longdouble : Theory
Begin

  Importing Cpp_double, Extended_Real

  % Alignment
  %-----
  align_longdouble : posnat

  % Value Range
  %-----
  % subset of number so that we can eventually formalise not-a-thing
  % and +-infinity values

  range_longdouble : non_empty_finite_set[extended_real]

  % (Un)interpreted Datatype
  %-----
  Importing Interpreted_Data[(range_longdouble)]

  dt_longdouble_exists : Axiom
    Exists (dt_longdouble : (pod_data_type?[(range_longdouble)])) : True

  % proof status :-)
  % dt_longdouble_TCC1 : TCC Obligation

  dt_longdouble : (pod_data_type?[(range_longdouble)])

  % 3.9.1 pt 8 precision float <= double <= long double
  % float values is subset of double values is subset of
  % long double values
  longdouble_precision : Axiom
    Forall (f : (range_double)) : real?(f) Implies range_longdouble(f)

End Cpp_longdouble

% 3.9.1 pt 6 Bool
%=====
Cpp_bool : Theory
Begin

  Importing Cpp_Deep_Types

  % Alignment
  %-----
  align_bool : posnat

  % !! To do:
  % I (Marcus) read the standard as follows; you have bool objects which
  % are at least byte aligned. Then there are bitfields where single bits

```

B PVS Theory Sources

```
% can represent a bool. The above is only for the objects of type bool.
% Compilers that pack multiple bools in one byte make use of bitfields.

% proof status :-)
% range_bool_TCC1 : TCC Obligation

% Value Range
%-----
range_bool : non_empty_finite_set[bool] = fullset[bool]

% (Un)interpreted Datatype
%-----
Importing Interpreted_Data[bool]

dt_bool_exists : Axiom
  Exists (dt_bool : (pod_data_type?[bool])) : True

% proof status :-)
% dt_bool_TCC1 : TCC Obligation

dt_bool : (pod_data_type?[bool])

% 3.9.1 pt 6 bool
% bool values behaves as integral type (true < false)
min_value(typ : (bool?)) : bool = true
max_value(typ : (bool?)) : bool = false

End Cpp_bool

% 3.9.1 pt 5 Wchar_t
%=====
Cpp_wchar_t : Theory
Begin

  Importing Cpp_char, Cpp_longlong

  % Alignment
  %-----
  align_wchar_t : posnat

  % 3.9.1 pt 5 "Type wchar_t shall have the same size, signedness and alignment
  % requirements as one other the other integral types, called its
  % underlying type."
  % signed types have the same alignment as their corresponding unsigned types
  wchar_t_alignment : Axiom
    align_wchar_t = align_char Or
    align_wchar_t = align_short Or
    align_wchar_t = align_int Or
    align_wchar_t = align_long Or
    align_wchar_t = align_longlong
```

```

% proof status :-)
% range_wchar_t_TCC1 : TCC Obligation

% Value Range
%-----
range_wchar_t : non_empty_finite_set[int]

% (Un)interpreted Datatype
%-----
Importing Interpreted_Data[(range_wchar_t)]

dt_wchar_t_exists : Axiom
  Exists (dt_wchar_t : (pod_data_type?[(range_wchar_t)])) : True

% proof status :-)
% dt_wchar_t_TCC1 : TCC Obligation

dt_wchar_t : (pod_data_type?[(range_wchar_t)])

% proof status :-)
% value_bitmask_wchar_t_TCC1 : TCC Obligation

value_bitmask_wchar_t : {l : list[Byte] | length(l) = size(uidt(dt_wchar_t))}

% 3.9.1 pt 5 "Type wchar_t shall have the same size, signedness and alignment
% requirements as one other the other integral types, called its
% underlying type."
% signed types have the same size as their corresponding unsigned types
dt_wchar_t_size : Axiom
  size(uidt(dt_wchar_t)) = size(uidt(dt_char)) Or
  size(uidt(dt_wchar_t)) = size(uidt(dt_short)) Or
  size(uidt(dt_wchar_t)) = size(uidt(dt_int)) Or
  size(uidt(dt_wchar_t)) = size(uidt(dt_long)) Or
  size(uidt(dt_wchar_t)) = size(uidt(dt_longlong))

End Cpp_wchar_t

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% A Deep Embedding of non union; non class C++ Types
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% The primary uses of the following deep embedding of C++ datatypes are
% – the polymorphic specificaiton of expressions on fundamental types,
% – the polymorphic specification of pointers and of references
%
% The below deep embedding does not formalise class and union members as these
% give rise to unlimited recursion. The following example illustrates this
% recursion:
%
```


B PVS Theory Sources

```
% struct A {  
% A* pt_A;  
% };
```

Cpp_Deep_Types : **THEORY**
BEGIN

Importing More_List_Props

Access_Specifier : **Type** = {public, private, protected}

Cpp_Type_ : **DATATYPE**
BEGIN

% fundamental types (3.9.1)

%-----

uchar : uchar?

schar : schar?

char : char?

short : short?

int : int?

long : long?

longlong : longlong?

ushort : ushort?

uint : uint?

ulong : ulong?

ulonglong : ulonglong?

wchar_t : wchar_t?

bool : bool?

float : float?

double : double?

longdouble : longdouble?

void : void?

% compound types (3.9.2)

%-----

% array types (8.3.4)

% 8.3.4 pt 1 [...] If the constant-expression is present, it shall be an

% integral constant expression and its value shall be greater

% than zero. [...]

% Thus, it is possible to encode arrays of unknown size as arrays of size 0

array_type(typ : Cpp_Type_, size : nat) : **array?**

% function types (8.3.5)

function_type(ret_typ : Cpp_Type_,
 args_typ : list[Cpp_Type_],

```

        var_args : bool) : function?
% member_function : bool) : function?

% pointer types (8.3.1)
pointer(typ : Cpp_Type_) : pointer?

% reference types (8.3.2)
reference(typ : Cpp_Type_) : reference?

% classes (clause 9)
class(name : string,
      abstract : bool) : class?

% unions (9.5)
union(name : string) : union?
% members : posnat,
% mutable : [below[members] -> bool],
% access_spec : [below[members] -> Access_Specifier]) : union?

% 3.9.2 pt 1. static members
% "Static class members are objects or functions, and pointers to them
% are ordinary pointers to objects or functions"
% => Static members are allocated at an object-independent address.
% Thus, there is no need to include static members into the deep
% embedding of classes and unions

% Bit fields are addressed relative to an allocation unit. An Address
% (e.g., a const reference to a bit field) identifies this allocation unit,
% the bit_offset addresses the first bits within this allocation unit and
% the bit_size the size of the bitfield. Therefore, size(uidt(dt_bitf))
% corresponds to the size of the allocation unit. We modify a bitfield by
% reading its allocation unit, setting the modified bits (as determined by
% the mask obtained from to_byte(dt_bitf) and by writing the allocation
% unit. Reading and writing of the allocation unit is by read list; reading
% may be omitted if the mask modifies all bits in the allocation unit.
% Allocation units may overlap. We translate unnamed bit fields and padding
% into "holes" in the respective allocation units.

% bit field (9.6)
bitfield (typ : Cpp_Type_, % integral or enum
         bit_offset : nat, bit_size : posnat) : bitfield?

% enumerations (7.2)
enum(name : string,
     typ : Cpp_Type_, % integral
     members : posnat,
     constant_value : [below[members] -> int]) : enum?

% proof status :-)
% Cpp_Type_pointer_to_member_eta_TCC1 : TCC Obligation

```

B PVS Theory Sources

```

% proof status :-)
% Cpp_Type_pointer_to_member_eta_TCC2 : TCC Obligation

% proof status :-)
% Cpp_Type_typ_pointer_to_member_TCC1 : TCC Obligation

% pointer-to-member (8.3.3)
pointer_to_member(typ : (class?),
                 member_typ : Cpp_Type_) : pointer_to_member?

% CV qualifiers (3.9.3)
%-----
% 3.9.3 pt 1 const and volatile qualified types define distinct C++ types
% ...
const (typ : Cpp_Type_) : const? % not class, union, array or
                                % function
volatile (typ : Cpp_Type_) : volatile? % -> semantic compiler sets cv
                                        % for member accesses from
                                        % within a cv function

END Cpp_Type_

% Properties of the subterm relation
%-----
% proof status :-)
subterm_less : Lemma
  Forall (t1, t2 : Cpp_Type_) :
    subterm(t1, t2) And t1 /= t2 Implies t1 << t2

% proof status :-)
subterm_transitive : Lemma
  Forall (t1, t2, t3 : Cpp_Type_) :
    subterm(t1, t2) And subterm(t2, t3) Implies subterm(t1, t3)

% Classification of Types
%=====

% proof status :-)
% c_TCC1 : TCC Obligation

% a possibly cv qualified type
c(p : PRED[Cpp_Type_])(typ : Cpp_Type_) : bool =
  p(typ) Or (const?(typ) And p(typ(typ)))

% proof status :-)
% v_TCC1 : TCC Obligation

v(p : PRED[Cpp_Type_])(typ : Cpp_Type_) : bool =
  p(typ) Or (volatile?(typ) And p(typ(typ)))

% proof status :-)
% cv_TCC1 : TCC Obligation

```

```

% proof status :-)
% cv_TCC2 : TCC Obligation

cv(p : PRED[Cpp_Type_])(typ : Cpp_Type_) : bool =
  c(p)(typ) Or v(p)(typ) Or
  (const?(typ) And volatile?(typ(typ)) And p(typ(typ(typ))))

% proof status :-)
cv_implied : Lemma
  Forall (p1, p2 : PRED[Cpp_Type_], typ : Cpp_Type_) :
  subset?(p1, p2) And cv(p1)(typ) Implies cv(p2)(typ)

% incomplete types (3.9 pt 6)
incomplete_type?(typ : Cpp_Type_) : bool =
% a class may become complete while the type remains the same
% (class?(typ) And size(typ) = 0) Or
  (array?(typ) And size(typ) = 0) Or
  void?(typ)

% object_types (3.9 pt 9)
object_type?(typ : Cpp_Type_) : bool =
  Not cv(function?)(typ) And Not cv(reference?)(typ) And Not cv(void?)(typ)

% Fundamental Types
%-----

% signed integer types (3.9.1 pt 2)
signed_integer?(typ : Cpp_Type_) : bool =
  schar?(typ) Or short?(typ) Or int?(typ) Or long?(typ) Or longlong?(typ)

% unsigned integer types (3.9.1 pt 3)
unsigned_integer?(typ : Cpp_Type_) : bool =
  uchar?(typ) Or ushort?(typ) Or uint?(typ) Or ulong?(typ) Or ulonglong?(typ)

% integral types (3.9.1 pt 7)
% we handle bool separately because the standard defines several
% special cases for bool expressions.
non_bool_integral?(typ : Cpp_Type_) : bool =
  char?(typ) Or wchar_t?(typ) Or
  signed_integer?(typ) Or unsigned_integer?(typ)

integral?(typ : Cpp_Type_) : bool =
  bool?(typ) Or non_bool_integral?(typ)

% floating point types (3.9.1 pt 8)
floating_point?(typ : Cpp_Type_) : bool =
  float?(typ) Or double?(typ) Or longdouble?(typ)

% arithmetic types (3.9.1 pt 8)
arithmetic?(typ : Cpp_Type_) : bool =
  integral?(typ) Or floating_point?(typ)

```

```

fundamental?(typ : Cpp_Type_) : bool =
  arithmetic?(typ) Or void?(typ)

% scalar types (3.9 p10)
scalar?(typ : Cpp_Type_) : bool =
  arithmetic?(typ) Or enum?(typ) Or
  pointer?(typ) Or pointer_to_member?(typ)

% pod struct / union types (3.9 pt 10)
pod_struct?(typ : (class?)) : bool
pod_union?(typ : (class?)) : bool

% proof status :-)
% pod?_TCC1 : TCC Obligation

% pod types (3.9 pt 10)
pod?(t : Cpp_Type_) : bool =
  scalar?(t) Or
  (class?(t) And (pod_struct?(t) Or pod_union?(t))) Or
  (array?(t) And (scalar?(typ(t)) Or
    (class?(typ(t)) Implies (pod_struct?(typ(t)) Or pod_union?(typ(t))))))

% Further Classifications
%-----

non_bool_integral_enum?(t : Cpp_Type_) : bool =
  non_bool_integral?(t) Or enum?(t)

% cpp types for which we define an interpreted data type
% - void, functions, arrays, classes and unions remain uninterpreted
interpreted?(t : Cpp_Type_) : bool =
  scalar?(t) Or reference?(t) Or bitfield?(t)

% Range of Values
%=====

% We use the PVS type number (more precisely predicates of this type)
% to encode the value range of fundamental types. Reals do not suffice
% because bool is an integral type and we do not want to define
% a real constant for true and false and because floating point types
% may encode positive and negative infinity and not-a-thing values

% 3.9.1 pt 9 void
range_void : finite_set[int] = emptyset[int]

Importing Uninterpreted_Data

% proof status :-)
% uidt_void_TCC1 : TCC Obligation

uidt_void : (uninterpreted_data_type?) =

```

```
(# size := 0, valid? := Lambda (l : list[Byte], a : Address) : False #)
```

```
End Cpp_Deep_Types
```

```
% To do: don't we need a theory instantiation of the above theories rather  
% than their importing?
```

```
Cpp_Integral_Types : Theory
```

```
Begin
```

```
Importing Interpreted_Data,  
          Cpp_ushort, Cpp_uint, Cpp_ulong, Cpp_ulonglong,  
          Cpp_longdouble, Cpp_bool, Cpp_wchar_t,  
          finite_sets@finite_sets_minmax[real, <=]
```

```
% proof status :-)  
% range_integral_TCC1 : TCC Obligation
```

```
% proof status :-)  
% range_integral_TCC2 : TCC Obligation
```

```
% proof status :-)  
% range_integral_TCC3 : TCC Obligation
```

```
% proof status :-)  
% range_integral_TCC4 : TCC Obligation
```

```
% proof status :-)  
% range_integral_TCC5 : TCC Obligation
```

```
% proof status :-)  
% range_integral_TCC6 : TCC Obligation
```

```
range_integral(typ : (non_bool_integral?)) : non_empty_finite_set[int] =
```

```
Cases typ Of
```

```
  uchar : range_uchar,  
  schar : range_schar,  
  char : range_char,
```

```
  short : range_short,  
  int : range_int,  
  long : range_long,  
  longlong : range_longlong,
```

```
  ushort : range_ushort,  
  uint : range_uint,  
  ulong : range_ulong,  
  ulonglong : range_ulonglong,
```

```
  wchar_t : range_wchar_t
```

```
EndCases
```

B PVS Theory Sources

```
% proof status :-)
% range_floating_point_TCC1 : TCC Obligation

range_floating_point(typ : (floating_point?)) :
  non_empty_finite_set[extended_real] =
Cases typ Of
  float : range_float,
  double : range_double,
  longdouble : range_longdouble
EndCases

% proof status :-)
% dt_integral_TCC1 : TCC Obligation

% proof status :-)
% dt_integral_TCC2 : TCC Obligation

% proof status :-)
% dt_integral_TCC3 : TCC Obligation

% proof status :-)
% dt_integral_TCC4 : TCC Obligation

% proof status :-)
% dt_integral_TCC5 : TCC Obligation

% proof status :-)
% dt_integral_TCC6 : TCC Obligation

% proof status :-)
% dt_integral_TCC7 : TCC Obligation

% proof status :-)
% dt_integral_TCC8 : TCC Obligation

% proof status :-)
% dt_integral_TCC9 : TCC Obligation

% proof status :-)
% dt_integral_TCC10 : TCC Obligation

% proof status :-)
% dt_integral_TCC11 : TCC Obligation

% proof status :-)
% dt_integral_TCC12 : TCC Obligation

dt_integral(typ : (non_bool_integral?)) :
  (pod_data_type?[(range_integral(typ))]) =
Cases typ Of
  uchar : dt_uchar,
```

```

schar : dt_schar,
char : dt_char,

short : dt_short,
int : dt_int,
long : dt_long,
longlong : dt_longlong,

ushort : dt_ushort,
uint : dt_uint,
ulong : dt_ulong,
ulonglong : dt_ulonglong,

wchar_t : dt_wchar_t
EndCases

% proof status :-)
% dt_floating_point_TCC1 : TCC Obligation

% proof status :-)
% dt_floating_point_TCC2 : TCC Obligation

% proof status :-)
% dt_floating_point_TCC3 : TCC Obligation

dt_floating_point(typ : (floating_point?)) :
  (pod_data_type?[(range_floating_point(typ))]) =
Cases typ Of
  float : dt_float,
  double : dt_double,
  longdouble : dt_longdouble
EndCases

% proof status :-)
% range_integral_unsigned_TCC1 : TCC Obligation

% proof status :-)
range_integral_unsigned : Judgement
  range_integral(typ : (unsigned_integer?)) Has_Type
  non_empty_finite_set[nat]

% proof status :-)
% max_value_TCC1 : TCC Obligation

% proof status :-)
% max_value_TCC2 : TCC Obligation

% 3.9.1 pt 8 "... numeric_limits shall specify the maximum and minimum values
% for each arithmetic type ..."
max_value(typ : (non_bool_integral?)) : int =
  max(range_integral(typ))

```


B PVS Theory Sources

```
% proof status :-)
% min_value_TCC1 : TCC Obligation

min_value(typ : (non_bool_integral?)) : int =
  min(range_integral(typ))

% proof status :-)
max_value_unsigned : Judgement
  max_value(typ : (unsigned_integer?)) Has_Type nat

% proof status :-)
min_value_unsigned : Judgement
  min_value(typ : (unsigned_integer?)) Has_Type nat

% proof status :-)
% max_value_bits_TCC1 : TCC Obligation

% proof status :-)
% max_value_bits_TCC2 : TCC Obligation

% proof status :-)
% max_value_bits_TCC3 : TCC Obligation

% proof status :-)
% max_value_bits_TCC4 : TCC Obligation

% proof status :-)
% max_value_bits_TCC5 : TCC Obligation

% proof status :-)
% max_value_bits_TCC6 : TCC Obligation

max_value_bits(typ : (cv(unsigned_integer?))) : Recursive posnat =
  Cases typ Of
    uchar : max_value_bits_uchar,
    ushort : max_value_bits_ushort,
    uint : max_value_bits_uint,
    ulong : max_value_bits_ulong,
    ulonglong : max_value_bits_ulonglong,
    const(t) : max_value_bits(t),
    volatile(t) : max_value_bits(t)
  EndCases
  Measure typ by <<

End Cpp_Integral_Types

Cpp_Deep_Type_Wellformedness : Theory
Begin

  Importing Cpp_Integral_Types
```

```

% Constraints on Recursive Definition of Types
%=====

% proof status :-)
% no_pointers_to_bitfield?_TCC1 : TCC Obligation

% Pointers
%-----
% 8.3.1 pt 4 no pointers to bitfields
no_pointers_to_bitfield?(t : Cpp_Type_) : bool =
  Not (pointer?(t) And cv(bitfield?)(typ(t)))

% 8.3.2 pt 4 no pointers to ref
no_pointers_to_references?(t : Cpp_Type_) : bool =
  Not (pointer?(t) And reference?(typ(t)))

% References
%-----

% proof status :-)
% no_cv_references?_TCC1 : TCC Obligation

% 8.3.2 pt 1 no cv(ref)
no_cv_references?(t : Cpp_Type_) : bool =
  Not ((const?(t) Or volatile?(t)) And reference?(typ(t)))

% proof status :-)
% no_reference_to_reference?_TCC1 : TCC Obligation

% 8.3.2 pt 4 no ref to ref
no_reference_to_reference?(t : Cpp_Type_) : bool =
  Not (reference?(t) And reference?(typ(t)))

% 9.6 pt 1 "Bit fields are packed into some addressable allocation unit"
% no ref to bitfield
no_reference_to_bitfields?(t : Cpp_Type_) : bool =
  Not (reference?(t) And cv(bitfield?)(typ(t)))

% 8.3.2 pt 4 "A reference shall be initialised to point to a valid object"
% 3.9.1 pt 9 "An object-type is a [...] type that is not [...] a void type."
% We take this to mean no references to void.
%>> 3.9.1 pt 9 declares functions are no objects, however,
% gcc allows int (&bar)(...)
no_reference_to_void?(t : Cpp_Type_) : bool =
  Not (reference?(t) And cv(void?)(typ(t)))

% Pointer To Member
%-----

% 8.3.3 pt 3 no pointer_to_member to static members (trivial)

```

```

% 8.3.3 pt 3 no pointer_to_member to reference
no_pointer_to_member_to_reference?(t : Cpp_Type_) : bool =
  Not (pointer_to_member?(t) And reference?(member_typ(t)))

% 8.3.3 pt 3 no pointer_to_member to cv void
no_pointer_to_member_to_cv_void?(t : Cpp_Type_) : bool =
  Not (pointer_to_member?(t) And cv(void?)(member_typ(t)))

% proof status :-)
% no_array_of_references?_TCC1 : TCC Obligation

% Arrays
%-----
% 8.3.2 pt 4 and 8.3.4 pt 1 no array of ref
no_array_of_references?(t : Cpp_Type_) : bool =
  array?(t) Implies Not reference?(typ(t))

% 8.3.4 pt 1 no array to cv void
no_array_of_cv_void?(t : Cpp_Type_) : bool =
  array?(t) Implies Not cv(void?)(typ(t))

% 8.3.4 pt 1 no array to function
no_array_of_function?(t : Cpp_Type_) : bool =
  array?(t) Implies Not function?(typ(t))

% 8.3.4 pt 1 no array of abstract class type
no_array_of_abstract_class?(t : Cpp_Type_) : bool =
  array?(t) Implies Not (class?(typ(t)) And abstract(typ(t)))

% struct { int [] : 5 }; is ill formed
no_array_of_bitfields?(t : Cpp_Type_) : bool =
  array?(t) Implies Not cv(bitfield?)(typ(t))

% Functions
%-----

% 8.3.5 pt 4 cv(function) only for nonstatic member functions
% Cv member functions access members as if they were cv qualified
% we handle this situation in the semantics compiler by declar
% members accessed within a cv function to have cv type.
% We can therefore forbid cv(function)
no_cv_function?(t : Cpp_Type_) : bool =
  Not ((const?(t) Or volatile?(t)) And function?(typ(t)))

% To do: semantics compiler: members accessed within cv function must
% have cv type

% 8.3.5 pt 2 no void function parameter
% f(void) is equivalent to f() with an empty list
no_cv_void_parameter?(t : Cpp_Type_) : bool =
  function?(t) Implies
    every(Lambda (p : Cpp_Type_) : not cv(void?)(p))(args_typ(t))

```

```

% proof status :-)
% no_pointer_or_ref_to_incomplete_array_parameter?_TCC1 : TCC Obligation

% proof status :-)
% no_pointer_or_ref_to_incomplete_array_parameter?_TCC2 : TCC Obligation

% proof status :-)
% no_pointer_or_ref_to_incomplete_array_parameter?_TCC3 : TCC Obligation

% proof status :-)
% no_pointer_or_ref_to_incomplete_array_parameter?_TCC4 : TCC Obligation

% proof status :-)
% no_pointer_or_ref_to_incomplete_array_parameter?_TCC5 : TCC Obligation

% proof status :-)
% no_pointer_or_ref_to_incomplete_array_parameter?_TCC6 : TCC Obligation

% 8.3.5 pt 6 no function parameters of type:
% pointer to array of unknown bound; reference to array of unknown bound
no_pointer_or_ref_to_incomplete_array_parameter?(t : Cpp_Type_) : bool =
  function?(t) Implies every(Lambda (p : Cpp_Type_) :
    Not ((pointer?(p) Or reference?(p)) And
      (array?(typ(p)) And size(typ(p)) > 0)) And
    Not ((const?(p) Or volatile?(p)) And pointer?(typ(p)) And
      (array?(typ(typ(p))) And size(typ(typ(p))) > 0)) And
    Not (const?(p) And volatile?(typ(p)) And pointer?(typ(typ(p))) And
      (array?(typ(typ(typ(p)))) And size(typ(typ(typ(p)))) > 0))
  )(args_typ(t))

% 8.3.5 pt 6 no return type array or function
no_array_return_type?(t : Cpp_Type_) : bool =
  Not (function?(t) And array?(ret_typ(t)))

no_function_return_type?(t : Cpp_Type_) : bool =
  Not (function?(t) And function?(ret_typ(t)))

% To do:
% 8.3.5 pt 6 no incomplete types in return type or function
% (except for member functions)

% Classes and Unions
%-----
% no restrictions on element types

% Bit Fields
%-----

% 9.6 pt 3 bit-field shall have integral or enumeration type

% proof status :-)

```

B PVS Theory Sources

```
% bitfield_underlying_integral_or_enum_type?_TCC1 : TCC Obligation

%>> we deviate from the standard by disallowing bools to be stored
% directly in a bitfield; use bool_to_int conversions instead
bitfield_underlying_integral_or_enum_type?(t : Cpp_Type_) : bool =
  bitfield?(t) Implies (non_bool_integral_enum?(typ(t)))

% Enums
%-----
% 7.2 "The underlying type [...] is an integral type that can
% represent all the [enum] values"

% proof status :-)
% enum_underlying_integral?_TCC1 : TCC Obligation

enum_underlying_integral?(t : Cpp_Type_) : bool =
  enum?(t) Implies non_bool_integral?(typ(t))

% proof status :-)
% enum_constants?_TCC1 : TCC Obligation

% proof status :-)
% enum_constants?_TCC2 : TCC Obligation

enum_constants?(e : (enum_underlying_integral?)) : bool =
  enum?(e) Implies
  Forall (m : below[members(e)]) :
    range_integral(typ(e))(constant_value(e)(m))

% CV Qualifiers
%-----

% proof status :-)
% const_volatile?_TCC1 : TCC Obligation

% 3.9.3 pt 1 const(volatile(typ)) = volatile(const(typ))
% (rule out the latter)
const_volatile?(t : Cpp_Type_) : bool =
  volatile?(t) Implies Not const?(typ(t))

% proof status :-)
% const_stutter?_TCC1 : TCC Obligation

const_stutter?(t : Cpp_Type_) : bool =
  const?(t) Implies Not const?(typ(t))

volatile_stutter?(t : Cpp_Type_) : bool =
  volatile?(t) Implies Not volatile?(typ(t))

% proof status :-)
% cv_array?_TCC1 : TCC Obligation
```

```

% 3.9.3 pt 2 cv(array(typ)) = array(cv(typ)) rule out the first
cv_array?(t : Cpp_Type_) : bool =
  (volatile?(t) Or const?(t)) Implies Not array?(typ(t))

% 3.9.3 pt 3 cv(class(t1, ..., tn)) =>
% cv(t1) .. cv(tn) provided ti is not a reference
% 3.9.3 pt 3 const(class(t1, ..., tn)) =>
% const(t1) .. const(tn) provided ti is not mutable
% We handle this case in the semantics compiler.
no_cv_class?(t : Cpp_Type_) : bool =
  Not ((const?(t) Or volatile?(t)) And class?(typ(t)))

no_cv_union?(t : Cpp_Type_) : bool =
  Not ((const?(t) Or volatile?(t)) And union?(typ(t)))

no_cv_void?(t : Cpp_Type_) : bool =
  Not ((const?(t) Or volatile?(t)) And void?(typ(t)))

% To do: semantics compiler cv(class(t1,...tn)) => cv(ti)

% Combination of individual constraints in a single predicative subtype
%-----

Cpp_Type?(typ : Cpp_Type_) : bool =
  Forall (t : Cpp_Type_) : subterm(t, typ) Implies
    no_pointers_to_bitfield?(t) And no_pointers_to_references?(t) And
    no_cv_references?(t) And no_reference_to_reference?(t) And
    no_reference_to_bitfields?(t) And
    no_pointer_to_member_to_reference?(t) And
    no_pointer_to_member_to_cv_void?(t) And no_cv_void_parameter?(t) And
    no_array_of_references?(t) And no_array_of_cv_void?(t) And
    no_array_of_function?(t) And no_array_of_abstract_class?(t) And
    no_pointer_or_ref_to_incomplete_array_parameter?(t) And
    no_array_return_type?(t) And no_function_return_type?(t) And
    bitfield_underlying_integral_or_enum_type?(t) And cv_array?(t) And
    enum_underlying_integral?(t) And enum_constants?(t) And
    const_volatile?(t) And
    const_stutter?(t) And volatile_stutter?(t) And no_cv_class?(t) And
    no_cv_union?(t) And no_cv_function?(t) And no_reference_to_void?(t) And
    no_cv_void?(t) And
    no_array_of_bitfields?(t)

Cpp_Type : Type = (Cpp_Type?)

Cpp_Subtype(P : [Cpp_Type_ -> bool]) : Type = {typ : Cpp_Type | P(typ)}

% proof status :-)
% cv_base_TCC1 : TCC Obligation

% proof status :-)
% cv_base_TCC2 : TCC Obligation

```

B PVS Theory Sources

```

% proof status :-)
% cv_base_TCC3 : TCC Obligation

% proof status :-)
% cv_base_TCC4 : TCC Obligation

% proof status :-)
% cv_base_TCC5 : TCC Obligation

% Reduce cv qualifiedness from a type
%-----
cv_base(typ : Cpp_Type) : Recursive Cpp_Type =
  Cases typ Of
    const(t) : cv_base(t),
    volatile(t) : cv_base(t)
  Else
    typ
  EndCases
Measure typ by <<

% proof status :-)
subterm_cv_base : Lemma
  Forall (typ : Cpp_Type) :
    subterm(cv_base(typ), typ)

% proof status :-)
cv_base_result : Lemma
  Forall (P : {Q : PRED[Cpp_Type_] | subset?(Q, interpreted?)},
    typ : Cpp_Subtype(cv(P))) :
    P(cv_base(typ))

% proof status :-)
cv_base_result2 : Lemma
  Forall (typ : Cpp_Type) :
    Not const?(typ) And Not volatile?(typ)
  Implies
    cv_base(typ) = typ

% proof status :-)
cv_base_not_const : Lemma
  Forall (typ : Cpp_Type) :
    Not const?(cv_base(typ))

% proof status :-)
cv_base_not_volatile : Lemma
  Forall (typ : Cpp_Type) :
    Not volatile?(cv_base(typ))

% proof status :-)
cv_base_cv : Lemma
  Forall (P : PRED[Cpp_Type_], typ : Cpp_Type) :

```

```

subset?(P, interpreted?) Implies
  (P(cv_base(typ)) IFF cv(P)(typ))

% Layout Compatibility
%=====

% proof status :-)
% layout_compatible?_TCC1 : TCC Obligation

% (3.9 pt 11) [reflexive] same types are layout compatible
% symmetric, transitive follows from nature of layout_compatibility?
layout_compatible? : equivalence[Cpp_Type]

% proof status :-)
% layout_compatible_enums_TCC1 : TCC Obligation

% proof status :-)
% layout_compatible_enums_TCC2 : TCC Obligation

% 7.2 pt 7 "Two enumeration types are layout compatible if they have the
% same underlying type."
layout_compatible_enums : Axiom
  Forall (e1, e2 : Cpp_Subtype(enum?)) :
    layout_compatible?(e1, e2) IFF typ(e1) = typ(e2)

% 3.9.3 pt 1 "The cv-qualified or cv-unqualified version of a type [...]
% shall have the same representation and alignment requirement"
% We take this to mean they are layout compatible.
layout_compatible_cv : Axiom
  Forall (t : Cpp_Type) :
    (Cpp_Type?(const(t)) Implies layout_compatible?(t, const(t))) And
    (Cpp_Type?(volatile(t)) Implies layout_compatible?(t, volatile(t))) And
    (Cpp_Type?(const(volatile(t))) Implies
      layout_compatible?(t, const(volatile(t))))

End Cpp_Deep_Type_Wellformedness

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% Compound Types
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% 7.2 Enumerations
%=====
Cpp_enum : Theory
Begin
  Importing Cpp_Deep_Type_Wellformedness

  e : Var Cpp_Subtype(enum?)

% Alignment

```


B PVS Theory Sources

```
%-----  
align_enum(e) : posnat  
  
% Value Range  
%-----  
  
% proof status :-)  
% valid_range?_TCC1 : TCC Obligation  
  
% proof status :-)  
% valid_range?_TCC2 : TCC Obligation  
  
% 7.2 pt 5 "The underlying type of an enumeration is an integral type that  
% can represent all the enumerator values"  
valid_range?(e)(r : non_empty_finite_set[int]) : bool =  
  subset?(r, range_integral(typ(e))) And  
  Forall (member : below[members(e)]) : r(constant_value(e)(member))  
  
% To do: ?  
% Currently the smallest range of an enum is comprised only of the discrete  
% enum constants. However, 7.2 pt 6 allows all values between emin and emax  
% or more precisely in the surrounding bitfield bmin, bmax.  
% 7.2 pt 6 bitfield  
% bmin <= emin <= emax <= bmax for a bitfield b  
  
% proof status :-)  
% range_enum_TCC1 : TCC Obligation  
  
range_enum(e) : (valid_range?(e))  
  
% (Un-)interpreted Datatype  
%-----  
  
dt_enum_exists : Axiom  
  Exists (dt : (pod_data_type?[(range_enum(e))])) : true  
  
% Remark: the datatype of the underlying type ---- dt(typ(e)) ---- should be  
% a valid model for dt(e) and all possible ranges.  
  
% proof status :-)  
% dt_enum_TCC1 : TCC Obligation  
  
dt_enum(e) : (pod_data_type?[(range_enum(e))])  
  
% proof status :-)  
% value_bitmask_enum_TCC1 : TCC Obligation  
  
value_bitmask_enum(e) : {l : list[Byte] | length(l) = size(uidt(dt_enum(e)))}  
  
% proof status :-)  
% enum_size_TCC1 : TCC Obligation
```

```

% proof status :-)
% enum_size_TCC2 : TCC Obligation

% 7.2 pt 5 "The value of sizeof() applied to an enumeration type [...] is
% the value of sizeof() applied to the underlying type. "
enum_size : Axiom
  size(uidt(dt_enum(e))) = size(uidt(dt_integral(typ(e))))

```

End Cpp_enum

Cpp_pointer : **Theory**
Begin

Importing Cpp_Deep_Type_Wellformedness

```

p : Var Cpp_Subtype(pointer?)
i : Var int

```

```

% Alignment
%-----
align_ptr(p) : posnat

```

```

% Value Range
%-----
Pointer_Base_Type : Type

```

```

% Alternative models:
% ~~~~~~
% 1) Pointer_Base_Type = Memory_Address
% - no bounds checking; + no hidden bits required ;
% + size = l2(Max_Address) possible
%
% 2) Pointer_Base_Type =
% [# base : Memory_Address, size : posnat, offset : int #]
% + bounds checking possible ;
% - requires hidden bits or size > l2(Max_Address)

```

% To do: implement the above two models

```

% 3.9.2 pt 3 pointer value is address of a byte in memory or a null pointer
range_pointer(p) : PRED[lift[Pointer_Base_Type]]

```

```

null?(ptr_val : lift[Pointer_Base_Type]) : bool = bottom?(ptr_val)

```

```

% we assert that the null pointer is in the semantics range of any pointer
null_every_range : Axiom
  Forall (p) : range_pointer(p)(bottom)

```

```

% (Un-)interpreted Datatype
%=====

```

B PVS Theory Sources

```
dt_pointer_exists : Axiom
  Exists (dt : (pod_data_type?[(range_pointer(p))])) : true

% proof status :-)
% dt_pointer_TCC1 : TCC Obligation

dt_pointer(p) : (pod_data_type?[(range_pointer(p))])

% proof status :-)
% pointers_to_layout_compatible_range_TCC1 : TCC Obligation

% proof status :-)
% pointers_to_layout_compatible_range_TCC2 : TCC Obligation

% proof status :-)
% pointers_to_layout_compatible_range_TCC3 : TCC Obligation

% proof status :-)
% pointers_to_layout_compatible_range_TCC4 : TCC Obligation

% 3.9.2 pt 3 pointers to layout-compatible (possibly cv-qualified) types
% shall have the same value representation and alignment
% requirements
pointers_to_layout_compatible_range : Axiom
  Forall (p1, p2 : Cpp_Subtype(pointer?)) :
    layout_compatible?(typ(p1), typ(p2)) Implies
      range_pointer(p1) = range_pointer(p2)

% proof status :-)
% pointers_to_layout_compatible_dt_TCC1 : TCC Obligation

pointers_to_layout_compatible_dt : Axiom
  Forall (p1, p2 : Cpp_Subtype(pointer?)) :
    layout_compatible?(typ(p1), typ(p2)) Implies
      dt_pointer(p1) = dt_pointer(p2)

pointers_to_layout_compatible_alignment : Axiom
  Forall (p1, p2 : Cpp_Subtype(pointer?)) :
    layout_compatible?(typ(p1), typ(p2)) Implies
      align_ptr(p1) = align_ptr(p2)

% 3.9.2 pt 4 "cv? void* and void* shall have the same representation
% and alignment requirements as a cv? char*"
void_pointer_char_pointer_range : Axiom
  Forall (p1, p2 : Cpp_Subtype(pointer?)) :
    void?(typ(p1)) And char?(typ(p2))
  Implies
    range_pointer(p1) = range_pointer(p2)

% proof status :-)
% void_pointer_char_pointer_dt_TCC1 : TCC Obligation
```

```

void_pointer_char_pointer_dt : Axiom
  Forall (p1, p2 : Cpp_Subtype(pointer?)) :
    void?(typ(p1)) And char?(typ(p2))
  Implies
    dt_pointer(p1) = dt_pointer(p2)

void_pointer_char_pointer_alignment : Axiom
  Forall (p1, p2 : Cpp_Subtype(pointer?)) :
    void?(typ(p1)) And char?(typ(p2))
  Implies
    align_ptr(p1) = align_ptr(p2)

% Pointer Arithmetic
%=====

% Address of Pointer and Arithmetic
%-----

ptyp : Var Cpp_Subtype(cv(pointer?))

not_null?(ptyp)(ptr_val : (range_pointer(cv_base(ptyp)))) : bool = Not null?(ptr_val)

address_of(ptyp)(ptr_val : (range_pointer(cv_base(ptyp)))) : lift[Memory_Address]

address_of_spec : Axiom
  Forall (ptr_val : (range_pointer(cv_base(ptyp)))) : not_null?(ptyp)(ptr_val)
  Implies up?(address_of(ptyp)(ptr_val))

address_of_spec2 : Axiom
  Forall (ptr_val : (range_pointer(cv_base(ptyp)))) :
    null?(ptr_val) Implies bottom?(address_of(ptyp)(ptr_val))

% proof status :-)
% add_TCC1 : TCC Obligation

% Integer addition for pointers
add(ptyp)(ptr_val : (range_pointer(cv_base(ptyp))), i : int) : (range_pointer(cv_base(ptyp)))

add_spec : Axiom
  Forall (ptr_val : (range_pointer(cv_base(ptyp)))) :
    not_null?(ptyp)(ptr_val) Iff not_null?(ptyp)(add(ptyp)(ptr_val, i))

% proof status :-)
% add_spec2_TCC1 : TCC Obligation

% proof status :-)
% add_spec2_TCC2 : TCC Obligation

add_spec2 : Axiom
  Forall (ptr_val : (range_pointer(cv_base(ptyp)))) :
    not_null?(ptyp)(ptr_val)
  Implies

```

```

    down(address_of(ptyp)(add(ptyp)(ptr_val, i))) =
      down(address_of(ptyp)(ptr_val)) + i

% proof status :-)
add_simplification : Lemma
  Forall (ptr_val : (range_pointer(cv_base(ptyp))), i, j : int) :
    address_of(ptyp)(add(ptyp)(add(ptyp)(ptr_val, i), j)) =
      address_of(ptyp)(add(ptyp)(ptr_val, i + j))

% Pointer subtraction
% 5.7 pt 6 defines subtraction only for pointers of the same array object.
% 5.9 pt 2 pointers to the same array are comparable

same_array(ptyp)(ptr_val1, ptr_val2 : (range_pointer(cv_base(ptyp)))) : bool

same_array_spec : Axiom
  Forall (ptr_val1, ptr_val2 : (range_pointer(cv_base(ptyp)))) :
    not_null?(ptyp)(ptr_val1) And not_null?(ptyp)(ptr_val2)
  Implies
    (same_array(ptyp)(ptr_val1, ptr_val2) IFF ptr_val1 = ptr_val2)

same_array_spec2 : Axiom
  Forall (ptr_val1, ptr_val2 : (range_pointer(cv_base(ptyp)))) :
    null?(ptr_val1) Or null?(ptr_val2)
  Implies
    same_array(ptyp)(ptr_val1, ptr_val2) = false

% Array Bounds Checking (Requires Model 2)
% ~~~~~
check_bounds(ptyp)(ptr_val : (range_pointer(cv_base(ptyp)))) : bool

check_bounds_spec : Axiom
  Forall (ptr_val : (range_pointer(cv_base(ptyp)))) :
    null?(ptr_val)
  Implies
    check_bounds(ptyp)(ptr_val) = false

% Respective model (2) implementations of these functions are:
% ~~~~~
%
% - add(p)(ptr_val, i) = 'offset += i
% - address_of(p)(ptr_val) = base + offset
% - check_bounds(p)(ptr_val) = 0 <= offset < size

% proof status :-)
% void_pointer_other_pointer_value_TCC1 : TCC Obligation

% proof status :-)
% void_pointer_other_pointer_value_TCC2 : TCC Obligation

% 3.9.2 pt 4 "A void* shall be able to hold hold any object pointer."
void_pointer_other_pointer_value : Axiom

```

```

Forall (p1, p2 : Cpp_Subtype(cv(pointer?)), ptr_val1 : (range_pointer(cv_base(p1)))) :
  void?(typ(cv_base(p1))) And not_null?(p1)(ptr_val1)

```

```

Implies

```

```

  Exists (ptr_val2 : (range_pointer(cv_base(p2)))) :
    not_null?(p2)(ptr_val2) And
    down(address_of(p1)(ptr_val1)) = down(address_of(p2)(ptr_val2))

```

```

End Cpp_pointer

```

```

% 8.3.2 Reference

```

```

%=====

```

```

Cpp_reference : Theory

```

```

Begin

```

```

Importing Cpp_Deep_Type_Wellformedness

```

```

r : Var Cpp_Subtype(reference?)

```

```

% Alignment

```

```

%-----

```

```

align_ref(r) : posnat

```

```

% Value Range

```

```

%-----

```

```

address_range(r) : PRED[Memory_Address]

```

```

% 8.3.2 pt 3 "It is unspecified whether or not a reference requires storage"

```

```

Reference_Type : Datatype

```

```

Begin

```

```

  % References stored in an allocated reference variable

```

```

  alloc (ref_variable : Address) : alloc?

```

```

  % References that are not allocated

```

```

  nonalloc (target : Memory_Address) : nonalloc?

```

```

End Reference_Type

```

```

range_reference(r) : PRED[Reference_Type] =

```

```

  Lambda (rt : Reference_Type) :

```

```

    Cases rt Of

```

```

      alloc(ref_var) : divides(align_ref(r), offset(ref_var)),

```

```

      nonalloc(target) : address_range(r)(target)

```

```

    EndCases

```

```

alloc_range(r)(rt : Reference_Type) : bool =

```

```

  range_reference(r)(rt) And alloc?(rt)

```

```

% To do: alignment of references

```

```

% 3.9 pt 5 objects are allocated at an address that meets the alignment

```

```

% requirements

```

```

% reference_to_aligned_object : Axiom

```

B PVS Theory Sources

```
% Forall (r : Reference, addr : (address_range(r))) :
% divides?(alignment(typ(r)), offset(addr))

% (Un-)interpreted Datatype
%-----

dt_ref_exists : Axiom
  Exists (dt : (pod_data_type?[(alloc_range(r))])) : true

% proof status :-)
% dt_reference_TCC1 : TCC Obligation

dt_reference(r) : (pod_data_type?[(alloc_range(r))])

End Cpp_reference

% 9 Class / Union
%=====
Cpp_class : Theory
Begin

  Importing Cpp_Deep_Type_Wellformedness

  c : Var Cpp_Subtype(class?)

  % Alignment
  %-----
  align_class(c) : posnat

  % Uninterpreted Datatype
  %-----
  % We define no semantic domain for classes and unions

  % proof status :-)
  % uidt_class_TCC1 : TCC Obligation

  uidt_class(c) : {uidt : (uninterpreted_data_type?) | size(uidt) > 0}

End Cpp_class

Cpp_union : Theory
Begin

  Importing Cpp_Deep_Type_Wellformedness

  u : Var Cpp_Subtype(union?)

  % Alignment
  %-----
  align_union(u) : posnat

  % Uninterpreted Datatype
```

```

%-----
% We define no semantic domain for classes and unions

% proof status :-)
% uidt_union_TCC1 : TCC Obligation

uidt_union(u) : {uidt : (uninterpreted_data_type?) | size(uidt) > 0}

```

End Cpp_union

```

% 8.3.4 Array

```

```

%=====

```

```

Cpp_array : Theory

```

```

Begin

```

```

Importing Cpp_Deep_Type_Wellformedness

```

```

a : Var Cpp_Subtype(array?)

```

```

% Alignment

```

```

%-----

```

```

align_arr(a) : posnat

```

```

% Uninterpreted Datatype

```

```

%-----

```

```

% We define no semantic domain for arrays

```

```

% proof status :-)

```

```

% uidt_array_TCC1 : TCC Obligation

```

```

uidt_array(a) : (uninterpreted_data_type?)

```

```

% 8.3.4 pt 6 subscript  $E1[E2] = *((E1) + (E2))$ 

```

End Cpp_array

```

% 8.3.5 Functions

```

```

%=====

```

```

Cpp_function : Theory

```

```

Begin

```

```

Importing Cpp_Deep_Type_Wellformedness

```

```

% Functions

```

```

%-----

```

```

% 8.3.5 pt 2 variable number of arguments  $f(\text{const char}^*, [], \dots)$ 

```

```

% To do :

```

```

% handle in semantics compiler by creating and passing var args object

```


B PVS Theory Sources

```
% 8.3.6
% We handle default arguments in the semantics compiler by generating
% appropriate call stubs

% To do:
% - pointer to function
% - recursive functions

% proof status :-)
% uidt_function_TCC1 : TCC Obligation

uidt_function(f : Cpp_Subtype(function?)) : (uninterpreted_data_type? =
  (# size := 0, valid? := Lambda (l : list[Byte], a : Address) : False #)

% Virtual Functions
%-----
c : Var Cpp_Subtype(class?)

Vtable : Type+

vtable(c) : Vtable

vtable_idt_exists : Axiom
  Exists (idt : (interpreted_data_type?[(singleton(vtable(c)))])) : true

% proof status :-)
% dt_vtable_TCC1 : TCC Obligation

dt_vtable(c) : (interpreted_data_type?[(singleton(vtable(c)))]))

End Cpp_function

% 9.6 Bit Fields
%=====
%
% We do not consider bit fields to directly participate in
% expressions. The integral promotion (§ 4.5 pt 3) and integral
% conversion (§ 4.7 pt 2 and pt 3) rules for bit fields allow bit fields
% to be regarded as signed or unsigned integers for the purpose of
% arithmetic operations.

Cpp_bitfield : Theory
Begin

  Importing Cpp_enum

  b : Var Cpp_Subtype(bitfield?)

  % Alignment
  %-----
  align_bitfield(b) : posnat
```

```

% 9.6 pt 1 "Alignment of bit-fields is implementation-defined."

% proof status :-)
% range_bitfield_TCC1 : TCC Obligation

% Value Range
%-----
% 9.6 pt 3 "It is implementation defined whether a plain [...] char, short,
% int or long bit-field is signed or unsigned."
%
% We take this to mean that bit fields taking viewer bits than the value
% representantion of the type can be signed or unsigned. Thus, we can define
% range(b) to be a subset of range(typ(b)).
range_bitfield(b) : non_empty_finite_set[int]

% proof status :-)
% bitfield_range_underlying_TCC1 : TCC Obligation

% proof status :-)
% bitfield_range_underlying_TCC2 : TCC Obligation

% proof status :-)
% bitfield_range_underlying_TCC3 : TCC Obligation

bitfield_range_underlying : Axiom
  If enum?(typ(b)) Then
    subset?(range_bitfield(b), range_enum(typ(b)))
  Else
    subset?(range_bitfield(b), range_integral(typ(b)))
  Endif

% (Un-)interpreted Datatype
%-----

dt_bitfield_exists : Axiom
  Exists (idt : (interpreted_data_type?[(range_bitfield(b))])) : True

% proof status :-)
% dt_bitfield_TCC1 : TCC Obligation

dt_bitfield(b) : (interpreted_data_type?[(range_bitfield(b))])

% the bitfield is contained in the allocation unit
bitfield_bits : Axiom
  bit_offset(b) + bit_size(b) <= size(uidt(dt_bitfield(b))) * bits_per_byte

% To do:
% - bit_masked(mask, n)
% - mask(to_byte(dt))
% - write_data(with_mask)
% - axioms asserting that non-bitfield_masks are full

```

B PVS Theory Sources

```
% only bits in the bitfield are masked
bitfield_mask : Axiom
  Forall (n : nat, data : (range_bitfield(b)), a : Address) :
    n < size(uidt(dt_bitfield(b))) * bits_per_byte And
    bit_set?(n, to_mask(dt_bitfield(b))(data, a))
  Implies
    (bit_offset(b) <= n And n < bit_offset(b) + bit_size(b))

End Cpp_bitfield

% 8.3.3 Pointer to Member
%=====
Cpp_pointer_to_member : Theory
Begin

  Importing Cpp_Deep_Type_Wellformedness

  p : Var Cpp_Subtype(pointer_to_member?)

  % Alignment
  %-----
  align_pointer_to_member(p) : posnat

  % Value Range
  %-----
  % The offsets of all class members of that type. Emptyset if no such member
  % is contained in the class. The pointer can at most be initialised with a
  % member of a base class but not with a member of a derived class.
  range_ptm(p) : finite_set[nat]

  % To do:
  % class members provided we could define an Axiom along the lines:
  % range(p) = {n : nat | Exists (m : member(typ(p))) :
  % member_typ(p) = typ(p)(m) Implies n = offset(m)}

  % (Un-)interpreted Datatype
  %-----

  dt_pointer_to_member_exists : Axiom
    Exists (dt : (pod_data_type?[(range_ptm(p))])) : True

  % proof status :-)
  % dt_ptm_TCC1 : TCC Obligation

  dt_ptm(p) : (pod_data_type?[(range_ptm(p))])

End Cpp_pointer_to_member

Cpp_const_volatile[Data : Type] : Theory
```

Begin**Assuming**

Importing Cpp_Deep_Type_Wellformedness

idt_exists : **Assumption**

Exists (idt : (interpreted_data_type?[Data])) : **True**

EndAssuming

```

% Volatile Datatypes
%=====
% 3.9.3 pt 1 "The cv-qualified or cv-unqualified versions of a type ...
% shall have the same representation and alignment requirements"
%
% In the present specification the behaviour of all write accesses is that
% of volatile accesses.
%
% The approach currently under discussion is to add a store-buffer like
% unit --- the compiler layer for volatile accesses --- as an additional
% memory layer. Non volatile writes may be buffered in this layer. Because
% of 3.9.3 pt 1 non volatile reads of volatile written data should succeed
% and return the same value. Therefore, we "have to (???) use hidden bits
% to signal a volatile access to the store buffer.
%
% Because hidden bits are not architecturally justified, we provide two
% models:
%
% - Model 1: which cannot distinguish volatile from non-volatile accesses
% and which needs no hidden bits, and,
% - Model 2: in which volatile accesses can be distinguished from
% non-volatile accesses using hidden bits

% proof status :-)
% dt_volatile_TCC1 : TCC Obligation

dt_volatile(idt : (interpreted_data_type?[Data])) :
  (interpreted_data_type?[Data])

dt_volatile_size : Axiom
Forall (idt : (interpreted_data_type?[Data])) :
  size(uidt(dt_volatile(idt))) = size(uidt(idt))

dt_volatile_pod : Axiom
Forall (idt : (pod_data_type?[Data])) :
  pod_data_type?[Data](dt_volatile(idt))

dt_volatile_spec_model_1 : Axiom
Forall (idt : (interpreted_data_type?[Data])) :
  dt_volatile(idt) = idt

```

```

%>> MV: The side effect cannot change the object representation from
% dt(volatile(int)) to dt(int) because the type information is
% lost at this level. We can only add or remove hidden bits
% that are set / reset in dt(volatile(int)) and not in dt(int)
% and which do not participate in the value representation.
dt_volatile_spec_model_2 : Axiom
  Forall (idt : (interpreted_data_type?[Data])) :
    dt_volatile(idt) /= idt And bits_per_byte > min_bits_per_byte

%>> shouldn't we instead use something like:
%>> - member(volatile_bit, hidden_bits_per_byte) and
%>> - bits_per_byte = min_bits_per_byte + card(hidden_bits_per_byte)

% Const Datatypes
%=====

dt_const(idt : (interpreted_data_type?[Data])) :
  (interpreted_data_type?[Data])

dt_const_size : Axiom
  Forall (idt : (interpreted_data_type?[Data])) :
    size(uidt(dt_const(idt))) = size(uidt(idt))

dt_const_pod : Axiom
  Forall (idt : (pod_data_type?[Data])) :
    pod_data_type?[Data](dt_const(idt))

dt_const_spec_model_1 : Axiom
  Forall (idt : (interpreted_data_type?[Data])) :
    dt_const(idt) = idt

dt_const_spec_model_2 : Axiom
  Forall (idt : (interpreted_data_type?[Data])) :
    dt_const(idt) /= idt And bits_per_byte > min_bits_per_byte

End Cpp_const_volatile

Cpp_Types : Theory
Begin

  Importing Cpp_class, Cpp_union, Min_Max_Extension,
    Cpp_function, Cpp_array, Cpp_bitfield, Cpp_pointer_to_member,
    Cpp_reference, Cpp_pointer, Cpp_const_volatile

  % Const Volatile Qualifiers
  % 3.9.3 pt 1 "The cv-qualified or cv-unqualified versions of a type ...
  % shall have the same representation and alignment requirements"

  % Alignment (3.9 pt 5)
  %=====

```

```

% proof status :-)
% alignment_TCC1 : TCC Obligation

% proof status :-)
% alignment_TCC2 : TCC Obligation

% proof status :-)
% alignment_TCC3 : TCC Obligation

% proof status :-)
% alignment_TCC4 : TCC Obligation

% proof status :-)
% alignment_TCC5 : TCC Obligation

% proof status :-)
% alignment_TCC6 : TCC Obligation

% proof status :-)
% alignment_TCC7 : TCC Obligation

% proof status :-)
% alignment_TCC8 : TCC Obligation

% proof status :-)
% alignment_TCC9 : TCC Obligation

% proof status :-)
% alignment_TCC10 : TCC Obligation

% proof status :-)
% alignment_TCC11 : TCC Obligation

% proof status :-)
% alignment_TCC12 : TCC Obligation

% proof status :-)
% alignment_TCC13 : TCC Obligation

alignment(typ : Cpp_Type) : Recursive nat =
Cases typ Of
  uchar : align_uchar,
  schar : align_schar,
  char : align_char,
  short : align_short,
  int : align_int,
  long : align_long,
  longlong : align_longlong,
  ushort : align_ushort,
  uint : align_uint,
  ulong : align_ulong,
  ulonglong : align_ulonglong,

```

B PVS Theory Sources

```
wchar_t : align_wchar_t,
bool : align_bool,
float : align_float,
double : align_double,
longdouble : align_longdouble,
void : 0,

array_type(t, s) : align_arr(typ),
function_type(r, a, v) : 0,
pointer(t) : align_ptr(typ),
reference(t) : align_ref(typ),
class(n, a) : align_class(typ),
union(n) : align_union(typ),
bitfield(t, bo, s) : align_bitfield(typ),
enum(n, t, m, c) : align_enum(typ),
pointer_to_member(t, m) : align_pointer_to_member(typ),

const (t) : alignment(t),
volatile (t) : alignment(t)
EndCases
Measure typ By <<

% proof status :- )
% pointer_to_aligned_object_TCC1 : TCC Obligation

% proof status :- )
% pointer_to_aligned_object_TCC2 : TCC Obligation

% proof status :- )
% pointer_to_aligned_object_TCC3 : TCC Obligation

% Pointer
%-----
% 3.9 pt 5 Objects are allocated at an address that meets the alignment
% requirements
pointer_to_aligned_object : Axiom
  Forall (p : Cpp_Subtype(cv(pointer?)), ptr_val : (range_pointer(cv_base(p)))) :
    not_null?(p)(ptr_val) Implies
      divides(alignment(typ(cv_base(p))), offset(down(address_of(p)(ptr_val))))

% proof status :- )
% align_array_of_type_TCC1 : TCC Obligation

% proof status :- )
% align_array_of_type_TCC2 : TCC Obligation

% Arrays
%-----
align_array_of_type : Axiom
  Forall (a : Cpp_Subtype(array?)) :
    divides(alignment(a), alignment(typ(a)))
```

```

% Small Super Type for Integral C++ Types
%=====

% proof status :-)
% range_TCC1 : TCC Obligation

% proof status :-)
% range_TCC2 : TCC Obligation

% proof status :-)
% range_TCC3 : TCC Obligation

% proof status :-)
% range_TCC4 : TCC Obligation

% proof status :-)
% range_TCC5 : TCC Obligation

% proof status :-)
% range_TCC6 : TCC Obligation

% proof status :-)
% range_TCC7 : TCC Obligation

% this range does not include the not allocated references
range(typ : Cpp_Subtype(cv(non_bool_integral_enum?))) : Recursive
  non_empty_finite_set[int] =
Cases typ Of
  enum(n, t, m, c) : range_enum(typ),
  const(t) : range(t),
  volatile(t) : range(t)
Else
  range_integral(typ)
EndCases
Measure typ By <<

% proof status :-)
% range_cv_base_TCC1 : TCC Obligation

% proof status :-)
range_cv_base : Lemma
Forall (typ : Cpp_Subtype(cv(non_bool_integral_enum?))) :
  range(typ) = range(cv_base(typ))

% (Un-)interpreted Datatype
%=====
Importing Interpreted_Data_Lift

% To do:
% – we probably need hidden bits to detect const modification through
% an alias and to distinguish volatile from non-volatile accesses
% – Hendrik's idea: use separate datatype for t and const(t);

```


B PVS Theory Sources

```

% semantics compiler ensures no assignments with const(t)

% proof status :-)
% dt_non_bool_integral_enum_TCC1 : TCC Obligation

% proof status :-)
% dt_non_bool_integral_enum_TCC2 : TCC Obligation

% proof status :-)
% dt_non_bool_integral_enum_TCC3 : TCC Obligation

% proof status :-)
% dt_non_bool_integral_enum_TCC4 : TCC Obligation

% Interpreted Datatype
%-----
dt_non_bool_integral_enum(typ : Cpp_Subtype(non_bool_integral_enum?)) :
  (interpreted_data_type?[(range(typ))]) =
  Cases typ Of
    enum(n, t, m, c) : dt_enum(typ)
  Else
    dt_integral(typ)
  EndCases

% proof status :-)
dt_non_bool_integral_enum_pod : Judgement
  dt_non_bool_integral_enum(typ : Cpp_Subtype(non_bool_integral_enum?))
    Has_Type (pod_data_type?[(range(typ))])

% proof status :-)
% dt_TCC1 : TCC Obligation

% proof status :-)
% dt_TCC2 : TCC Obligation

% proof status :-)
% dt_TCC3 : TCC Obligation

% proof status :-)
% dt_TCC4 : TCC Obligation

% proof status :-)
% dt_TCC5 : TCC Obligation

% Generic Datatype Model
%=====
dt(typ : Cpp_Subtype(cv(non_bool_integral_enum?))) : Recursive
  (interpreted_data_type?[(range(typ))]) =
  Cases typ Of
    const(t) : dt_const(dt(t)),
    volatile(t) : dt_volatile(dt(t))
  Else

```

```

    dt_non_bool_integral_enum(typ)
EndCases
Measure typ by <<

% proof status :-)
dt_pod : Judgement
  dt(typ : Cpp_Subtype(cv(non_bool_integral_enum?))) Has_Type
    (pod_data_type?[(range(typ))])

% proof status :-)
% dt_cv_pointer_TCC1 : TCC Obligation

% proof status :-)
% dt_cv_pointer_TCC2 : TCC Obligation

% proof status :-)
% dt_cv_pointer_TCC3 : TCC Obligation

% proof status :-)
% dt_cv_pointer_TCC4 : TCC Obligation

% proof status :-)
% dt_cv_pointer_TCC5 : TCC Obligation

% proof status :-)
% dt_cv_pointer_TCC6 : TCC Obligation

% proof status :-)
% dt_cv_pointer_TCC7 : TCC Obligation

% proof status :-)
% dt_cv_pointer_TCC8 : TCC Obligation

% proof status :-)
% dt_cv_pointer_TCC9 : TCC Obligation

% proof status :-)
% dt_cv_pointer_TCC10 : TCC Obligation

% proof status :-)
% dt_cv_pointer_TCC11 : TCC Obligation

% proof status :-)
% dt_cv_pointer_TCC12 : TCC Obligation

% proof status :-)
% dt_cv_pointer_TCC13 : TCC Obligation

% proof status :-)
% dt_cv_pointer_TCC14 : TCC Obligation

% proof status :-)

```

B PVS Theory Sources

```
% dt_cv_pointer_TCC15 : TCC Obligation

dt_cv_pointer(typ : Cpp_Subtype(cv(pointer?))) : Recursive
  (interpreted_data_type?[(range_pointer(cv_base(typ)))] =
  Cases typ Of
    const(t) : dt_const(dt_cv_pointer(t)),
    volatile(t) : dt_volatile(dt_cv_pointer(t))
  Else
    dt_pointer(typ)
  EndCases
  Measure typ by <<

% proof status : -)
% dt_cv_bitfield_TCC1 : TCC Obligation

% proof status : -)
% dt_cv_bitfield_TCC2 : TCC Obligation

% proof status : -)
% dt_cv_bitfield_TCC3 : TCC Obligation

% proof status : -)
% dt_cv_bitfield_TCC4 : TCC Obligation

% proof status : -)
% dt_cv_bitfield_TCC5 : TCC Obligation

% proof status : -)
% dt_cv_bitfield_TCC6 : TCC Obligation

% proof status : -)
% dt_cv_bitfield_TCC7 : TCC Obligation

% proof status : -)
% dt_cv_bitfield_TCC8 : TCC Obligation

% proof status : -)
% dt_cv_bitfield_TCC9 : TCC Obligation

% proof status : -)
% dt_cv_bitfield_TCC10 : TCC Obligation

% proof status : -)
% dt_cv_bitfield_TCC11 : TCC Obligation

% proof status : -)
% dt_cv_bitfield_TCC12 : TCC Obligation

% proof status : -)
% dt_cv_bitfield_TCC13 : TCC Obligation

% proof status : -)
```

```

% dt_cv_bitfield_TCC14 : TCC Obligation

% proof status :-)
% dt_cv_bitfield_TCC15 : TCC Obligation

dt_cv_bitfield(typ : Cpp_Subtype(cv(bitfield?))) : Recursive
  (interpreted_data_type?[(range_bitfield(cv_base(typ)))] =
  Cases typ Of
    const(t) : dt_const(dt_cv_bitfield(t)),
    volatile(t) : dt_volatile(dt_cv_bitfield(t))
  Else
    dt_bitfield(typ)
  EndCases
Measure typ by <<

% proof status :-)
% dt_cv_ptm_TCC1 : TCC Obligation

% proof status :-)
% dt_cv_ptm_TCC2 : TCC Obligation

% proof status :-)
% dt_cv_ptm_TCC3 : TCC Obligation

% proof status :-)
% dt_cv_ptm_TCC4 : TCC Obligation

% proof status :-)
% dt_cv_ptm_TCC5 : TCC Obligation

% proof status :-)
% dt_cv_ptm_TCC6 : TCC Obligation

% proof status :-)
% dt_cv_ptm_TCC7 : TCC Obligation

% proof status :-)
% dt_cv_ptm_TCC8 : TCC Obligation

% proof status :-)
% dt_cv_ptm_TCC9 : TCC Obligation

% proof status :-)
% dt_cv_ptm_TCC10 : TCC Obligation

% proof status :-)
% dt_cv_ptm_TCC11 : TCC Obligation

% proof status :-)
% dt_cv_ptm_TCC12 : TCC Obligation

% proof status :-)

```

B PVS Theory Sources

```
% dt_cv_ptm_TCC13 : TCC Obligation

% proof status :-)
% dt_cv_ptm_TCC14 : TCC Obligation

% proof status :-)
% dt_cv_ptm_TCC15 : TCC Obligation

dt_cv_ptm(typ : Cpp_Subtype(cv(pointer_to_member?))) : Recursive
  (interpreted_data_type?[(range_ptm(cv_base(typ)))] =
  Cases typ Of
    const(t) : dt_const(dt_cv_ptm(t)),
    volatile(t) : dt_volatile(dt_cv_ptm(t))
  Else
    dt_ptm(typ)
  EndCases
  Measure typ by <<

% proof status :-)
% dt_cv_float_TCC1 : TCC Obligation

% proof status :-)
% dt_cv_float_TCC2 : TCC Obligation

% proof status :-)
% dt_cv_float_TCC3 : TCC Obligation

% proof status :-)
% dt_cv_float_TCC4 : TCC Obligation

% proof status :-)
% dt_cv_float_TCC5 : TCC Obligation

% proof status :-)
% dt_cv_float_TCC6 : TCC Obligation

% proof status :-)
% dt_cv_float_TCC7 : TCC Obligation

% proof status :-)
% dt_cv_float_TCC8 : TCC Obligation

% proof status :-)
% dt_cv_float_TCC9 : TCC Obligation

% proof status :-)
% dt_cv_float_TCC10 : TCC Obligation

% proof status :-)
% dt_cv_float_TCC11 : TCC Obligation

% proof status :-)
```

```

% dt_cv_float_TCC12 : TCC Obligation

% proof status :-)
% dt_cv_float_TCC13 : TCC Obligation

% proof status :-)
% dt_cv_float_TCC14 : TCC Obligation

% proof status :-)
% dt_cv_float_TCC15 : TCC Obligation

dt_cv_float(typ : Cpp_Subtype(cv(floating_point?))) : Recursive
  (interpreted_data_type?[(range_floating_point(cv_base(typ)))] =
  Cases typ Of
    const(t) : dt_const(dt_cv_float(t)),
    volatile(t) : dt_volatile(dt_cv_float(t))
  Else
    dt_floating_point(typ)
  EndCases
Measure typ by <<

% Dt is pod judgements
%-----
% To do (if required)

% proof status :-)
% uidt_TCC1 : TCC Obligation

% proof status :-)
% uidt_TCC2 : TCC Obligation

% proof status :-)
% uidt_TCC3 : TCC Obligation

% proof status :-)
% uidt_TCC4 : TCC Obligation

% proof status :-)
% uidt_TCC5 : TCC Obligation

% proof status :-)
% uidt_TCC6 : TCC Obligation

% proof status :-)
% uidt_TCC7 : TCC Obligation

% proof status :-)
% uidt_TCC8 : TCC Obligation

% proof status :-)
% uidt_TCC9 : TCC Obligation

```

B PVS Theory Sources

% proof status :-)
% uidt_TCC10 : TCC Obligation

% proof status :-)
% uidt_TCC11 : TCC Obligation

% proof status :-)
% uidt_TCC12 : TCC Obligation

% proof status :-)
% uidt_TCC13 : TCC Obligation

% proof status :-)
% uidt_TCC14 : TCC Obligation

% proof status :-)
% uidt_TCC15 : TCC Obligation

% proof status :-)
% uidt_TCC16 : TCC Obligation

% proof status :-)
% uidt_TCC17 : TCC Obligation

% proof status :-)
% uidt_TCC18 : TCC Obligation

% proof status :-)
% uidt_TCC19 : TCC Obligation

% proof status :-)
% uidt_TCC20 : TCC Obligation

% proof status :-)
% uidt_TCC21 : TCC Obligation

% proof status :-)
% uidt_TCC22 : TCC Obligation

% proof status :-)
% uidt_TCC23 : TCC Obligation

% proof status :-)
% uidt_TCC24 : TCC Obligation

% proof status :-)
% uidt_TCC25 : TCC Obligation

% proof status :-)
% uidt_TCC26 : TCC Obligation

% proof status :-)

```

% uidt_TCC27 : TCC Obligation

% proof status :-)
% uidt_TCC28 : TCC Obligation

% proof status :-)
% uidt_TCC29 : TCC Obligation

% proof status :-)
% uidt_TCC30 : TCC Obligation

% proof status :-)
% uidt_TCC31 : TCC Obligation

% proof status :-)
% uidt_TCC32 : TCC Obligation

% proof status :-)
% uidt_TCC33 : TCC Obligation

% proof status :-)
% uidt_TCC34 : TCC Obligation

% proof status :-)
% uidt_TCC35 : TCC Obligation

% proof status :-)
% uidt_TCC36 : TCC Obligation

% proof status :-)
% uidt_TCC37 : TCC Obligation

% proof status :-)
% uidt_TCC38 : TCC Obligation

% proof status :-)
% uidt_TCC39 : TCC Obligation

% proof status :-)
% uidt_TCC40 : TCC Obligation

% proof status :-)
% uidt_TCC41 : TCC Obligation

% Uninterpreted Datatype
%-----
uidt(typ : Cpp_Type) : Recursive (uninterpreted_data_type?) =
  Cases typ Of
    void : uidt_void,
    array_type(t, s) : uidt_array(typ),
    function_type(r, a, v) : uidt_function(typ),
    class(n, a) : uidt_class(typ),

```



```

union(n) : uidt_union(typ),

% interpreted types
pointer(t) : uidt(dt_cv_pointer(typ)),
reference(t) : uidt(dt_reference(typ)),
pointer_to_member(c, t) : uidt(dt_cv_ptm(typ)),
bitfield(t, bo, s) : uidt(dt_cv_bitfield(typ)),

bool : uidt(dt_bool),

% qualifiers
const(t) :
  Cases t Of
    bool : uidt(dt_bool),
    pointer(t1) : uidt(dt_cv_pointer(typ)),
    pointer_to_member(c, t1) : uidt(dt_cv_ptm(typ)),
    bitfield(t1, bo, s) : uidt(dt_cv_bitfield(typ)),
    volatile(t1) : uidt(t)
  Else
    Cond
      floating_point?(t) -> uidt(dt_cv_float(typ)),
      non_bool_integral_enum?(t) -> uidt(dt(typ))
    EndCond
  EndCases,

volatile(t) :
  Cases t Of
    bool : uidt(dt_bool),
    pointer(t1) : uidt(dt_cv_pointer(typ)),
    pointer_to_member(c, t1) : uidt(dt_cv_ptm(typ)),
    bitfield(t1, bo, s) : uidt(dt_cv_bitfield(typ))
  Else
    Cond
      floating_point?(t) -> uidt(dt_cv_float(typ)),
      non_bool_integral_enum?(t) -> uidt(dt(typ))
    EndCond
  EndCases

Else
  Cond
    floating_point?(typ) -> uidt(dt_cv_float(typ)),
    non_bool_integral_enum?(typ) -> uidt(dt(typ))
  EndCond
EndCases
Measure typ by <<

% Array
%-----
% 8.3.4 pt 1 contiguously allocated set of N objects
array_size : Axiom
Forall (a : Cpp_Subtype(array?)) :
  size(uidt(a)) = size(a) * size(uidt(typ(a)))

```

```

% Size Of (3.9 pt 4)
%-----
size_of(typ : Cpp_Type) : nat = size(uidt(typ))

% Modulo Arithmetic / Floating Point Values
%-----
% arithmetic types (except bool), enums and bitfields may follow the rules
% of arithmetic modulo  $2^n$  where  $n$  is the number of bits in the value
% representation. Floating point values may have to be adjusted to a
% representable higher or lower value. Pointers effectively wrap around,
% although the standard leaves this behaviour undefined:
% (e.g., char * p = &0xffffffff p++ = 0).
%
% We therefore allow value modifications for any interpreted type.
%
% Completing the specification of the below underspecified function
% should allow us to add this functionality. For example
%  $\text{mod}(\text{typ} : \text{cv}(\text{unsigned}))(n) = n \bmod 2^{\text{max\_value\_bits}(\text{typ})}$ 

mod(typ : Cpp_Subtype(cv(non_bool_integral_enum?)))(n : int) : int

mod_range : Axiom
  Forall (typ : Cpp_Subtype(cv(non_bool_integral_enum?)), n : (range(typ))) :
    mod(typ)(n) = n

% proof status :-)
% mod_float_TCC1 : TCC Obligation

mod_float(typ : Cpp_Subtype(cv(floating_point?)))(n : extended_real) :
  extended_real

mod_float_range : Axiom
  Forall (typ : Cpp_Subtype(cv(floating_point?)),
    n : (range_floating_point(cv_base(typ)))) :
    mod_float(typ)(n) = n

% proof status :-)
% value_bitmask_TCC1 : TCC Obligation

% bits in value representation
value_bitmask(typ : Cpp_Subtype(cv(non_bool_integral_enum?))) : Recursive
  list[Byte] =
Cases typ Of
  char : value_bitmask_char,
  schar : value_bitmask_schar,
  uchar : value_bitmask_uchar,
  short : value_bitmask_short,
  ushort : value_bitmask_ushort,
  int : value_bitmask_int,
  uint : value_bitmask_uint,
  long : value_bitmask_long,

```

```

    ulong : value_bitmask_ulong,
    longlong : value_bitmask_longlong,
    ulonglong : value_bitmask_ulonglong,
    wchar_t : value_bitmask_wchar_t,
    enum(n,t,m,c) : value_bitmask_enum(typ),
    const(t) : value_bitmask(t),
    volatile(t) : value_bitmask(t)
EndCases
Measure typ by <<

```

Auto_Rewrite— Cpp_Type?

End Cpp_Types

B.33 vfiasco-prelude.pvs

Unit : **Datatype**

% the unit type (aka semantic of void)

Begin

unit : unit?

End Unit

More_Divides : **Theory**

Begin

n,m : **Var** nat

% proof status :-)

expt_divides : **Lemma Forall**(a : posnat) : a > 1 **And** n <= m **Implies**
 divides(expt(a, n), expt(a, m))

% proof status :-)

divides_expt_gt : **Lemma Forall**(i, j : nat) :
 j <= i **And** divides(expt(2, i), n) **Implies** divides(expt(2, j), n)

End More_Divides

Expt_Lemmas : **Theory**

Begin

m, n : **VAR** nat

n0x : **VAR** nzreal

gt1x, gt1y : **VAR** {r: posreal | r > 1}

% proof status :-)

both_sides_expt_gt1_le : **LEMMA**
 m <= n **IFF** expt(gt1x, m) <= expt(gt1x, n)

% proof status :-)

expt_plus : **Lemma Forall**(n0x : nzreal, m, n : nat) :
 expt(n0x, m) * expt(n0x, n) = expt(n0x, m + n)

End Expt_Lemmas

Posnat_induction : **Theory**

Begin

p: **VAR** pred[posnat]

i, j : **VAR** posnat

% proof status :-)

posnat_induction : **Lemma**

(p(1) **And** (**Forall** j: p(j) **Implies** p(j+1)))
Implies (**Forall** i: p(i))

End Posnat_induction

Number_Props : **Theory**

Begin

Importing More_Divides, Expt_Lemmas

% proof status :-)

number_split : **Lemma** **Forall**(x : int, a : posnat) :

x = a * ndiv(x, a) + rem(a)(x)

% proof status :-)

number_split_less : **Lemma** **Forall**(a, b, p, q : nat) :

a < p **Implies**
(a + b * p < p * q **IFF** b < q)

% proof status :-)

bit_split_less : **Lemma** **Forall**(q, r, j, k : nat) :

r < expt(2, k) **Implies**
(q * expt(2, k) + r < expt(2, k + j) **IFF**
q < expt(2, j))

% proof status :-)

rem_def_pos : **Lemma** **Forall**(b : posnat, x : nat) :

Forall(r : mod(b)): rem(b)(x) = r **IFF** **Exists**(q : nat): x = b * q + r

% proof status :-)

rem_rem : **Lemma** **Forall**(a, b : posnat, x : int) :

divides(a, b) **Implies**
rem(a)(rem(b)(x)) = rem(a)(x)

% proof status :-)

divides_times : **Lemma** **Forall**(i : posnat, j, k : int) :

divides(i * j, k) **IFF** divides(i, k) **And** divides(j, ndiv(k, i))

% proof status :-)

```

ndiv_floor : Lemma Forall(a : int, b : posnat) : ndiv(a,b) = floor(a/b)

% proof status :-)
ndiv_self : Lemma Forall(a : posnat) : ndiv(a,a) = 1

% proof status :-)
ndiv_plus_1 : Lemma Forall(a, b : int, c : posnat) :
  divides(c, a) Implies
    ndiv(a + b, c) = ndiv(a,c) + ndiv(b,c)

% proof status :-)
ndiv_plus_2 : Lemma Forall(a, b : int, c : posnat) :
  divides(c, b) Implies
    ndiv(a + b, c) = ndiv(a,c) + ndiv(b,c)

% proof status :-)
ndiv_plus_mod : Lemma Forall(a : int, c : posnat, b : mod(c)) :
  divides(c, a) Implies
    ndiv(a + b, c) = ndiv(a, c)

% proof status :-)
ndiv_plus_mod_2 : Lemma Forall(a : int, c : posnat, b1, b2 : mod(c)) :
  divides(c, a) Implies
    ndiv(a + b1, c) = ndiv(a + b2, c)

% proof status :-)
ndiv_minus_2 : Lemma Forall(a, b : int, c : posnat) :
  divides(c, b) Implies
    ndiv(a - b, c) = ndiv(a,c) - ndiv(b,c)

% ndiv_minus_1 is much harder, because it might
% ndiv(a - b, c) = ndiv(a,c) - ndiv(b,c) - 1
% if rem(c)(b) > 0 or might it not??

% proof status :-)
ndiv_times_divident_1 : Lemma Forall(a, b : int, c : posnat) :
  divides(c, a) Implies
    ndiv(a * b, c) = ndiv(a,c) * b

% proof status :-)
ndiv_times_divident_2 : Lemma Forall(a, b : int, c : posnat) :
  divides(c, b) Implies
    ndiv(a * b, c) = a * ndiv(b,c)

% proof status :-)
rem_prod_3 : Lemma Forall(b1, b2 : posnat, x : int) :
  rem(b1 * b2)(x) = rem(b1)(x) + rem(b2)(ndiv(x, b1)) * b1

% proof status :-)
ndiv_times_2 : Lemma Forall(a : int, b, c : posnat) :
  ndiv(a, b * c) = ndiv(ndiv(a, b), c)

```

```

% proof status :-)
both_sides_ndiv_lt1 : Lemma Forall(a, b : int, c : posnat) :
  a < b And divides(c, b) Implies
  ndiv(a,c) < ndiv(b,c)

% proof status :-)
rem_ndiv : Lemma Forall(a : posnat, b : int, c : posnat) :
  rem(a)(ndiv(b, c)) = ndiv(rem(c * a)(b), c)

% proof status :-)
mod_rem : Lemma Forall(i : int, b : posnat) :
  mod(i, b) = rem(b)(i)

% proof status :-)
% ndiv_rem_divisible_TCC1 : TCC Obligation

% proof status :-)
ndiv_rem_divisible : Lemma Forall(a, c : posnat, b : int) :
  divides(c, a) And c <= a Implies
  ndiv(rem(a)(b), c) = rem(ndiv(a, c))(ndiv(b,c))

% proof status :-)
expt_2_8 : Lemma expt(2, 8) = 256

% proof status :-)
ndiv_0 : Lemma Forall(b : posnat) : ndiv(0, b) = 0

% proof status :-)
ndiv_1 : Lemma Forall(i : int) : ndiv(i, 1) = i

% proof status :-)
ndiv_bigger : Lemma Forall(n : nat, m : posnat) :
  n < m Implies ndiv(n, m) = 0

% proof status :-)
ndiv_reduce : Lemma Forall(a : int, b : posnat, c : int, d : posnat) :
  a * d = b * c Implies
  ndiv(a,b) = ndiv(c,d)

% proof status :-)
% ndiv_expt_expt_TCC1 : TCC Obligation

% proof status :-)
ndiv_expt_expt : Lemma Forall(a : posnat, n, m : nat) :
  a > 1 Implies
  ndiv(expt(a, n), expt(a, m)) =
  IF n < m Then 0
  Else expt(a, n - m)
  Endif

% Floor : x - rem(x / b)
%=====

```

B PVS Theory Sources

```
floor(b : posnat)(x : int) : int =
  x - rem(b)(x)

% proof status :-)
% min_plus1_TCC1 : TCC Obligation

% proof status :-)
min_plus1 : Lemma
  Forall (P : (nonempty?[nat])) :
    Not P(0) Implies
      min(P) = min(Lambda (n : nat) : P(n + 1)) + 1

% proof status :-)
min_member : Lemma
  Forall (P : (nonempty?[nat]), n : nat) :
    min(P) = n Implies P(n)

End Number_Props

Extended_Real : Theory
Begin

% The purpose of this theory is to define arithmetic on real numbers
% that can possibly contain infinity values (as e.g., in  $x / \infty = 0$ ),
% not-a-thing values (e.g.,  $x / 0 = \text{NAT}$ ) and exception values (not
% currently supported, i.e., no exception will be generated).

extended_real? : PRED[number]

extended_real_superset_of_number_fields : Axiom
  Forall (n : number_field) : extended_real?(n)

extended_real : Type = (extended_real?)

extended_real_number : Judgement
  extended_real Subtype_Of number

% proof status :-)
number_field_extended : Judgement
  number_field Subtype_Of extended_real

% extended real ops
%-----
x, y : Var extended_real

% proof status :-)
% er_plus_TCC1 : TCC Obligation

% proof status :-)
% er_neg_TCC1 : TCC Obligation
```

```

er_plus(x, y) : extended_real
er_minus(x, y) : extended_real
er_times(x, y) : extended_real
er_div(x, y) : extended_real
er_neg(x) : extended_real

% on number fields these ops behave like the number field ops
%-----
a, b : Var number_field

er_plus_ax : Axiom
  er_plus(a, b) = number_fields.+(a, b)

er_minus_ax : Axiom
  er_minus(a, b) = number_fields.-(a, b)

er_times_ax : Axiom
  er_times(a, b) = number_fields.*(a, b)

er_div_ax : Axiom
  b /= 0 Implies er_div(a, b) = number_fields./(a, b)

er_neg_ax : Axiom
  er_neg(a) = number_fields.-(a)

% Judgements:
%-----

% number_field
% ~~~~~
% proof status :-)
er_plus_number_field : Judgement
  er_plus(a, b : number_field) Has_Type number_field

% proof status :-)
er_minus_number_field : Judgement
  er_minus(a, b : number_field) Has_Type number_field

% proof status :-)
er_times_number_field : Judgement
  er_times(a, b : number_field) Has_Type number_field

% proof status :-)
er_div_number_field : Judgement
  er_div(a : number_field, b : nznum) Has_Type number_field

% proof status :-)
er_neg_number_field : Judgement
  er_neg(a : number_field) Has_Type number_field

% real

```


B PVS Theory Sources

```
%~~~~~  
% proof status :-)  
er_plus_real : Judgement  
  er_plus(a, b : real) Has_Type real  
  
% proof status :-)  
er_minus_real : Judgement  
  er_minus(a, b : real) Has_Type real  
  
% proof status :-)  
er_times_real : Judgement  
  er_times(a, b : real) Has_Type real  
  
% proof status :-)  
er_div_real : Judgement  
  er_div(a : real, b : nzreal) Has_Type real  
  
% proof status :-)  
er_neg_real : Judgement  
  er_neg(a : real) Has_Type real  
  
% rational  
%~~~~~  
% proof status :-)  
er_plus_rational : Judgement  
  er_plus(a, b : rational) Has_Type rational  
  
% proof status :-)  
er_minus_rational : Judgement  
  er_minus(a, b : rational) Has_Type rational  
  
% proof status :-)  
er_times_rational : Judgement  
  er_times(a, b : rational) Has_Type rational  
  
% proof status :-)  
er_div_rational : Judgement  
  er_div(a : rational, b : nzrat) Has_Type rational  
  
% proof status :-)  
er_neg_rational : Judgement  
  er_neg(a : rational) Has_Type rational  
  
% integers  
%~~~~~  
% proof status :-)  
er_plus_integer : Judgement  
  er_plus(a, b : int) Has_Type int  
  
% proof status :-)  
er_minus_integer : Judgement  
  er_minus(a, b : int) Has_Type int
```

```

% proof status :-)
er_times_integer : Judgement
  er_times(a, b : int) Has_Type int

```

```

% proof status :-)
er_neg_integer : Judgement
  er_neg(a : int) Has_Type int

```

End Extended_Real

More_Sets_Lemmas[T: TYPE]: THEORY

BEGIN

```

a, b, c : VAR set[T]

```

```

%% (auto-rewrite
%% "subset_equal" "union_subset1" "union_subset3"
%% "union_left" "union_right" "subset_singleton"
%% "subset_bigger_union_right" "subset_bigger_union_left"
%% "union_left" "union_right" )

```

```

% proof status :-)
subset_equal : Lemma
  a = b Implies subset?(a, b)

```

```

% proof status :-)
union_subset3: Lemma subset?(a, union(b, a))

```

```

% proof status :-)
subset_bigger_union_left : Lemma
  subset?(a, b) Implies subset?(a, union(b, c))

```

```

% proof status :-)
subset_bigger_union_right : Lemma
  subset?(a, c) Implies subset?(a, union(b, c))

```

```

% proof status :-)
union_empty_left : Lemma
  union(emptyset, a) = a

```

```

% proof status :-)
union_left : Lemma Forall (e : T) :
  a(e) Implies union(a, b)(e)

```

```

% proof status :-)
union_right : Lemma Forall (e : T) :
  b(e) Implies union(a, b)(e)

```

```

% proof status :-)
subset_of_differences : Lemma
  subset?(a, b) Implies subset?(difference(c, b), difference(c, a))

```

B PVS Theory Sources

```
% proof status :-)
difference_disjoint_3: LEMMA disjoint?(difference(b, a), a)

% proof status :-)
subset_singleton : Lemma Forall(e : T) :
  subset?(singleton(e), a) = a(e)

% proof status :-)
subset_member : Lemma Forall(e : T) :
  a(e) And subset?(a, b) Implies b(e)

% proof status :-)
disjoint_symmetric : Lemma
  disjoint?(a, b) = disjoint?(b, a)

% proof status :-)
disjoint_mono : Lemma Forall(a1, a2, b1, b2 : set[T]) :
  subset?(a1, a2) And subset?(b1, b2) And
  disjoint?(a2, b2) Implies
  disjoint?(a1, b1)

% proof status :-)
disjoint_subset_difference : Lemma
  subset?(a, b) And disjoint?(a, c) Implies
  subset?(a, difference(b, c))

End More_Sets_Lemmas

Set_Lift[Data1, Data2 : Type] : Theory
Begin

  set_lift(P : PRED[Data1], inj : [Data1 -> Data2]) : PRED[Data2] =
    Lambda (d2 : Data2) : Exists (d1 : (P)) : inj(d1) = d2

End Set_Lift

Alignment : Theory
Begin
  Importing Number_Props

  n,m : Var nat

  aligned?(n) : PRED[nat] = { a : nat | divides(expt(2,n), a) }

  aligned(n) : Type = (aligned?(n))

  % proof status :-)
  aligned_zero : Lemma aligned?(n)(0)
```

```

% proof status :-)
zero_aligned : Lemma aligned?(0)(n)

% proof status :-)
aligned_bigger : Lemma Forall(n1, n2 : nat) :
  n2 <= n1 And aligned?(n1)(m) Implies aligned?(n2)(m)

% aligned_add : Judgement +(a:aligned(n), b:aligned(m))
% HAS_TYPE aligned(min(n,m))

% proof status :-)
aligned_plus : Lemma
  Forall(a : aligned(n), b : aligned(m)) : aligned?(min(n,m))(a + b)

% proof status :-)
aligned_plus_n : Lemma
  Forall(a, b : aligned(n)) : aligned?(n)(a + b)

% proof status :-)
% aligned_minus_TCC1 : TCC Obligation

% proof status :-)
aligned_minus : Lemma
  Forall(a : aligned(n), b : aligned(m)) :
    a >= b Implies aligned?(min(n,m))(a - b)

% proof status :-)
aligned_mult_expt_2 : Lemma
  Forall(a : aligned(n)) : aligned?(n + m)(a * expt(2, m))

% proof status :-)
aligned_rem : Lemma Forall(i : nat) :
  aligned?(n)(m) Implies aligned?(n)(rem(expt(2, i))(m))

% aligned_minus_rem : Lemma Forall (i : nat) :
% aligned?(i)(n - rem(expt(2, i))(n))
% >> (use "rem_def2")(expand "aligned?")(smash)

% proof status :-)
aligned_add_below : Lemma Forall (n, m, i, k : nat) :
  i <= k And n < expt(2, k) And aligned?(i)(n) And m < expt(2, i) Implies
  n + m < expt(2, k)

```

End Alignment

More_List_Props[T : Type] : Theory

Begin

```

% proof status :-)
length_is_cons : Lemma Forall(l : list[T]) :
  length(l) > 0 Implies cons?(l)

```

B PVS Theory Sources

```

% proof status :-)
cons_length : Lemma
  Forall(l : (cons?[T])) : length(l) > 0

% proof status :-)
% length_cdr_TCC1 : TCC Obligation

% proof status :-)
length_cdr : Lemma Forall(l : list[T]) :
  length(l) > 0 Implies
    length(cdr(l)) = length(l) - 1

% proof status :-)
conlist_induction: Lemma
  Forall(p: [(cons?[T]) -> boolean]):
    (Forall(t : T) : p(cons(t, null))) And
    (Forall(t : T, tail : (cons?[T])): p(tail) Implies p(cons(t, tail)))
  Implies
    Forall (l : (cons?[T])): p(l)

% proof status :-)
% head_TCC1 : TCC Obligation

% proof status :-)
% head_TCC2 : TCC Obligation

% proof status :-)
% head_TCC3 : TCC Obligation

head(l : list[T], n : upto(length(l))) : Recursive list[T] =
  If n = 0 Then
    null
  Else
    cons(car(l), head(cdr(l), n - 1))
  Endif
Measure n

tail(l : list[T], n : upto(length(l))) : Recursive list[T] =
  If n = 0 Then
    l
  Else
    tail(cdr(l), n - 1)
  Endif
Measure n

% proof status :-)
head_tail : Lemma Forall (l : list[T], n : nat) :
  n <= length(l) Implies
    append(head(l, n), tail(l, n)) = l

% proof status :-)

```

```

length_head : Lemma Forall (l : list[T], n : nat) :
  n <= length(l) Implies length(head(l, n)) = n

% proof status :-)
length_tail : Lemma Forall (l : list[T], n : nat) :
  n <= length(l) Implies length(tail(l, n)) = length(l) - n

% proof status :-)
% nth_head_TCC1 : TCC Obligation

% proof status :-)
% nth_head_TCC2 : TCC Obligation

% proof status :-)
nth_head : Lemma Forall(l : list[T], n, i : nat) :
  n <= length(l) And i < n Implies
  nth(head(l, n), i) = nth(l, i)

% proof status :-)
% nth_tail_TCC1 : TCC Obligation

% proof status :-)
% nth_tail_TCC2 : TCC Obligation

% proof status :-)
nth_tail : Lemma Forall(l : list[T], n, i : nat) :
  n <= length(l) And i + n < length(l) Implies
  nth(tail(l, n), i) = nth(l, n + i)

% proof status :-)
% car_head_TCC1 : TCC Obligation

% proof status :-)
% car_head_TCC2 : TCC Obligation

% proof status :-)
car_head : Lemma Forall(l : list[T], n : upto(length(l))) :
  n > 0 Implies
  car(head(l, n)) = car(l)

% proof status :-)
cons_tail : Lemma Forall(l : list[T], n : upto(length(l))) :
  n < length(l) Implies
  cons?(tail(l, n))

% proof status :-)
% car_tail_TCC1 : TCC Obligation

% proof status :-)
car_tail : Lemma Forall(l : list[T], n : upto(length(l))) :
  n < length(l) Implies
  car(tail(l, n)) = nth(l, n)

```

```

% proof status :-)
% cdr_tail_TCC1 : TCC Obligation

% proof status :-)
cdr_tail : Lemma Forall(l : list[T], n : upto(length(l))) :
  n < length(l) Implies
  cdr(tail(l, n)) = tail(l, n+1)

% proof status :-)
every_implied : Lemma Forall(l : list[T], P, Q : PRED[T]) :
  (Forall (t : T) : P(t) Implies Q(t)) And
  every(P)(l)
Implies
  every(Q)(l)

% proof status :-)
every_conjunct_left : Lemma Forall(l : list[T], P, Q : PRED[T]) :
  every(P)(l) And every(Q)(l) Implies every(lambda(t : T) : P(t) And Q(t))(l)

% proof status :-)
every_iff_forall : Lemma Forall (l : list[T], P : PRED[T]) :
  every(P)(l) IFF (Forall (t : T) : member(t, l) Implies P(t))

% proof status :-)
some_iff_exists : Lemma Forall (l : list[T], P : PRED[T]) :
  some(P)(l) IFF (Exists (t : T) : member(t, l) And P(t))

% proof status :-)
% list_extensionality_TCC1 : TCC Obligation

% proof status :-)
list_extensionality : Lemma Forall(l1, l2 : list[T]) :
  l1 = l2 IFF
  (length(l1) = length(l2) AND
  Forall(i : below(length(l1))) : nth(l1,i) = nth(l2,i))

% proof status :-)
% list_remove_TCC1 : TCC Obligation

% proof status :-)
% list_remove_TCC2 : TCC Obligation

list_remove(x : T, l : list[T]) : Recursive list[T] =
Cases l OF
  null : null,
  cons(car, cdr) :
    IF x = car Then list_remove(x, cdr)
    Else cons(car, list_remove(x, cdr))

```

```

Endif
EndCases
Measure length(l)

% proof status :-)
list_remove_commutative : Lemma
  Forall(x, y : T, l : list[T]) :
    list_remove(x, list_remove(y, l)) = list_remove(y, list_remove(x, l))

% proof status :-)
member_list_remove : Lemma
  Forall(x, y : T, l : list[T]) :
    member(x, list_remove(y, l)) =
      IF x = y then false
      Else member(x, l)
      EndIF

% proof status :-)
nth_member : Lemma
  Forall(x : T, l : list[T]) :
    member(x, l) IFF
      Exists(i : below(length(l))) : nth(l, i) = x

% proof status :-)
% flatten_TCC1 : TCC Obligation

flatten(ll : list[list[T]]) : Recursive list[T] =
  Cases ll OF
    null : null,
    cons(car, cdr) : append(car, flatten(cdr))
  EndCases
Measure length(ll)

% proof status :-)
member_append : Lemma
  Forall(x : T, l1, l2 : list[T]) :
    member(x, append(l1, l2)) IFF
      member(x, l1) OR member(x, l2)

% proof status :-)
member_flatten : Lemma
  Forall(x : T, ll : list[list[T]]) :
    member(x, flatten(ll)) IFF
      Exists(l : list[T]) :
        member(l, ll) And member(x, l)

% proof status :-)

```


B PVS Theory Sources

```
% map2_TCC1 : TCC Obligation

map2(f : [T, T -> T])(l1, l2 : list[T]) : Recursive list[T] =
  Cases l1 Of
    null : null,
    cons(h1, t1) :
      Cases l2 Of
        null : null,
        cons(h2, t2) : cons(f(h1, h2), map2(f)(t1, t2))
      EndCases
    EndCases
  Measure length(l1)

% proof status :-)
map2_property : Lemma
  Forall (P : PRED[T], f : [T, T -> T], l1, l2 : list[T]) :
    (Forall (t1, t2 : T) : P(f(t1, t2))) Implies
      every(P)(map2(f)(l1, l2))

End More_List_Props

More_List_Props_2[A, B : Type] : Theory
Begin

  % proof status :-)
  member_map : Lemma
    Forall(b : B, f : [A -> B], l : list[A]) :
      member(b, map(f)(l)) IFF
        Exists(a : A) : member(a, l) And f(a) = b

End More_List_Props_2

More_More_List_Props [T : Type+] : Theory
Begin

  % proof status :-)
  list_all_length : Lemma Forall (n : nat) :
    Exists (l : list[T]) : length(l) = n

End More_More_List_Props

Even_More_List_Props [T : Type, S : Type From T] : Theory
Begin

  % proof status :-)
  every_extend : Lemma
    Forall (l : list[S], P : PRED[T]) :
      every(Lambda (t : T) : P(t))(l) Implies every(Lambda (s : S) : P(s))(l)

End Even_More_List_Props
```

End Even_More_List_Props

More_Relations[T : **Type**] : **Theory**

Begin

Importing More_List_Props *% for cons_length*

R : **Var** PRED[[T,T]]

% Zorn's lemma (I missed it in the prelude)

% proof status :-)

zorn_lr : **Lemma**

well_founded?(R) **Implies**

Not Exists (f : [nat \rightarrow T]) : **Forall** (n : nat) : R(f(n+1), f(n))

End More_Relations

More_Function[Domain, Range : **Type**] : **Theory**

% some more stuff on functions

Begin

f : **Var** [Domain \rightarrow Range]

% proof status :-)

% restrict_to_image_TCC1 : TCC Obligation

restrict_to_image(f) : [Domain \rightarrow (image(f, fullset[Domain]))] = f

% proof status :-)

surjective_restrict_to_image : **Lemma**

surjective?(restrict_to_image(f))

% proof status :-)

bijjective_restrict_to_image : **Lemma**

injective?(f) **Implies** bijective?(restrict_to_image(f))

End More_Function

Finite_Set_Reduce[T : **Type**] : **Theory**

Begin

Importing More_List_Props *% for nth_member*

S : **Var** finite_set[T]

% proof status :-)

% list_of_finite_set_TCC1 : TCC Obligation

% proof status :-)

% list_of_finite_set_TCC2 : TCC Obligation

```

list_of_finite_set(S) : Recursive list[T] =
  IF empty?(S) Then null
  Else cons(choose(S), list_of_finite_set(rest(S)))
  Endif
Measure card(S)

% proof status :-)
length_list_of_finite_set : Lemma
  length(list_of_finite_set(S)) = card(S)

% proof status :-)
member_list_of_finite_set : Lemma
  Forall(x : T) :
    member(x, S) = member(x, list_of_finite_set(S))

% proof status :-)
unique_member_list_of_finite_set : Lemma
  Forall(i, j : below(length(list_of_finite_set(S)))) :
    nth(list_of_finite_set(S), i) = nth(list_of_finite_set(S), j) Implies
      i = j

End Finite_Set_Reduce

Min_Max_Extension[T : Type, S : Type From T, <= : (total_order?[T])] : Theory
Begin

  Importing finite_sets@finite_sets_minmax

  % proof status :-)
  % max_stable_under_extension_TCC1 : TCC Obligation

  % proof status :-)
  max_stable_under_extension : Lemma
    Forall (P : non_empty_finite_set[S]) :
      max[T, <=](extend[T, S, bool, false](P)) = max[S, <=](P)

  % proof status :-)
  min_stable_under_extension : Lemma
    Forall (P : non_empty_finite_set[S]) :
      min[T, <=](extend[T, S, bool, false](P)) = min[S, <=](P)

End Min_Max_Extension

```

C Proof scripts

Contents for Appendix C

C.1	Proofs for Abstract_Read_Write (abstract_data.pvs)	489
C.1.1	read_data_valid_in_mem	489
C.2	Proofs for Abstract_Read_Write_Plain (plain_memory.pvs)	489
C.3	Proofs for Address_Datatype (paging_data.pvs)	489
C.3.1	address_data_type_TCC1	489
C.4	Proofs for Address_Helpers (constants.pvs)	489
C.4.1	type_add_rewrite	489
C.4.2	offset_add_rewrite	490
C.4.3	type_add_memory	490
C.4.4	address_add_zero	490
C.4.5	address_add_sum	490
C.4.6	reg_size_TCC1	491
C.4.7	reg_size_TCC2	491
C.5	Proofs for Address_Model (paging_data_models.pvs)	491
C.5.1	address_to_byte_TCC1	491
C.5.2	address_to_byte_TCC2	492
C.5.3	address_to_byte_TCC3	492
C.5.4	address_from_byte_TCC1	492
C.5.5	address_from_byte_TCC2	492
C.5.6	address_from_byte_TCC3	493
C.5.7	address_from_byte_TCC4	493
C.5.8	address_from_byte_TCC5	493
C.5.9	address_model_TCC1	495
C.5.10	address_valid_to_byte	495
C.5.11	address_is_pod	495
C.6	Proofs for Address_Util (memory.pvs)	497
C.6.1	address_block_subset_1	497
C.6.2	address_block_subset_2	497
C.6.3	blocks_disjoint_disjoint	497
C.6.4	blocks_disjoint_symmetric	499
C.6.5	blocks_in_larger_set	499
C.6.6	blocks_pairwise_disjoint_add	499
C.6.7	blocks_pairwise_disjoint_remove	503
C.6.8	subset_block_is_free	503
C.6.9	block_is_free_add	504
C.7	Proofs for Alignment (vfiasco_prelude.pvs)	504
C.7.1	aligned_zero	504
C.7.2	zero_aligned	504
C.7.3	aligned_bigger	505
C.7.4	aligned_plus	505
C.7.5	aligned_plus_n	506

C Proof scripts

C.7.6	aligned_minus_TCC1	506
C.7.7	aligned_minus	506
C.7.8	aligned_mult_expt_2	507
C.7.9	aligned_rem	508
C.7.10	aligned_add_below	510
C.8	Proofs for Allocation_Info (allocators.pvs)	511
C.8.1	consistent_allocator_state_change	511
C.8.2	consistent_allocator_read_TCC1	512
C.8.3	consistent_allocator_read	512
C.8.4	consistent_allocator_write_TCC1	512
C.8.5	consistent_allocator_write	513
C.8.6	consistent_allocator_alloc_TCC1	513
C.8.7	consistent_allocator_alloc	514
C.8.8	consistent_allocator_free_TCC1	516
C.8.9	consistent_allocator_free_TCC2	516
C.8.10	consistent_allocator_free	517
C.9	Proofs for Allocation_Table (allocators.pvs)	522
C.9.1	allocation_finite_tree	522
C.9.2	allocation_subnodes_allocated_and_free	522
C.9.3	allocation_subnodes_disjoint	523
C.9.4	allocation_roots_disjoint	523
C.9.5	allocation_private_memory_disjoint	523
C.9.6	allocation_consistent_allocator	524
C.9.7	allocation_allocated_allocation_point	524
C.9.8	allocation_allocation_point_in_memory_pool	524
C.9.9	allocators_different_in_hierachy	525
C.9.10	uppath_memory_pool	525
C.9.11	uppath_memory_pool_allocated_and_free_TCC1	526
C.9.12	uppath_memory_pool_allocated_and_free	527
C.9.13	allocation_memory_pools	527
C.9.14	allocation_info_TCC1	532
C.9.15	member_allocation_info	533
C.9.16	allocation_consistent	533
C.9.17	allocation_alloc_other_private_memory_TCC1	536
C.9.18	allocation_alloc_other_private_memory_TCC2	537
C.9.19	allocation_alloc_other_private_memory	537
C.9.20	allocation_alloc_TCC1	538
C.9.21	allocation_alloc	538
C.10	Proofs for Allocator (allocators.pvs)	550
C.10.1	null_address_TCC1	550
C.10.2	allocator?_TCC1	550
C.10.3	allocator?_TCC2	550
C.10.4	allocator?_TCC3	551
C.10.5	allocator?_TCC4	552
C.10.6	allocator?_TCC5	553
C.10.7	allocator?_TCC6	555
C.10.8	allocator?_TCC7	556
C.10.9	allocator?_TCC8	558
C.10.10	allocator_subset_private_memory_plain_memory	560
C.10.11	allocator_private_memory_in_plain_memory	560
C.10.12	allocator_subset_allocated_memory_pool	560
C.10.13	allocator_alloc_disjoint	561

C.10.14	allocator_allocated_alloc_TCC1	561
C.10.15	allocator_allocated_alloc	561
C.10.16	allocator_memory_pool_alloc	562
C.10.17	allocator_subset_address_block_alloc_allocated	562
C.10.18	allocator_free_freed_size	563
C.10.19	allocator_allocated_free_TCC1	563
C.10.20	allocator_allocated_free_TCC2	563
C.10.21	allocator_allocated_free	563
C.10.22	allocator_memory_pool_free	564
C.10.23	allocator_const_private_memory_memory_pool	564
C.10.24	allocator_const_private_memory_allocated	565
C.10.25	allocator_next_state_alloc_TCC1	565
C.10.26	allocator_next_state_alloc	565
C.10.27	allocator_next_state_free_TCC1	566
C.10.28	allocator_next_state_free	566
C.10.29	allocator_only_changes_alloc_TCC1	566
C.10.30	allocator_only_changes_alloc	567
C.10.31	allocator_only_changes_free_TCC1	567
C.10.32	allocator_only_changes_free	568
C.10.33	allocator_allocated_memory_read_TCC1	568
C.10.34	allocator_allocated_memory_read	568
C.10.35	allocator_memory_pool_memory_read	569
C.10.36	allocator_stays_unchanged_memory_write_private_mem_TCC1	570
C.10.37	allocator_stays_unchanged_memory_write_private_mem_TCC2	570
C.10.38	allocator_stays_unchanged_memory_write_private_mem	571
C.10.39	allocator_allocated_memory_write	571
C.10.40	allocator_memory_pool_memory_write	572
C.10.41	allocator_subset_address_block_alloc_memory_pool	572
C.11	Proofs for ArithmeticExpressions (expressions.pvs)	572
C.11.1	mod_impl_TCC1	572
C.11.2	div_impl_pos_range_TCC1	573
C.11.3	plus_ptr_TCC1	573
C.11.4	plus_ptr_TCC2	574
C.11.5	plus_ptr_TCC3	577
C.11.6	plus_ptr_TCC4	577
C.11.7	plus_ptr_TCC5	578
C.11.8	plus_ptr_TCC6	578
C.11.9	postinc_ptr_TCC1	578
C.11.10	postinc_ptr_TCC2	578
C.11.11	postinc_ptr_TCC3	579
C.11.12	postinc_ptr_TCC4	579
C.11.13	subscript_TCC1	580
C.11.14	subscript_TCC2	580
C.11.15	subscript_TCC3	584
C.11.16	subscript_TCC4	584
C.11.17	ptr_to_nonzero_object_TCC1	584
C.11.18	ptr_to_nonzero_object_TCC2	585
C.11.19	ptr_to_nonzero_object_TCC3	586
C.11.20	ptr_to_nonzero_object	586
C.11.21	minus_ptr_typ_is_Cpp_Type_TCC1	615
C.11.22	minus_ptr_typ_is_Cpp_Type	615
C.11.23	minus_ptr_ptr_TCC1	619

C.11.24	minus_ptr_ptr_TCC2	619
C.11.25	minus_ptr_ptr_TCC3	620
C.11.26	minus_ptr_ptr_TCC4	620
C.11.27	minus_ptr_ptr_TCC5	622
C.11.28	minus_ptr_ptr_TCC6	623
C.11.29	minus_ptr_ptr_TCC7	624
C.11.30	minus_ptr_ptr_TCC8	625
C.11.31	minus_ptr_ptr_TCC9	626
C.11.32	minus_ptr_ptr_TCC10	626
C.11.33	cmp_pointer_TCC1	627
C.11.34	cmp_pointer_TCC2	627
C.12	Proofs for AssemblerStatements (statements.pvs)	628
C.13	Proofs for AssignmentExpressions (expressions.pvs)	628
C.13.1	assign_times_TCC1	628
C.13.2	assign_plus_ptr_TCC1	628
C.13.3	assign_plus_ptr_TCC2	628
C.14	Proofs for AssignmentExpressions2 (expressions.pvs)	629
C.15	Proofs for Bitlist_ops (bits.pvs)	629
C.15.1	binary_not_TCC1	629
C.15.2	binary_and_TCC1	629
C.15.3	binary_or_TCC1	630
C.15.4	binary_xor_TCC1	630
C.15.5	singleton_zero_mask	630
C.15.6	bit_set?_TCC1	635
C.15.7	bit_set?_TCC2	635
C.16	Proofs for Bitops (bits.pvs)	635
C.16.1	bit_set_zero_mask	635
C.16.2	bit_set_ndiv_expt2	636
C.16.3	bit_set_full_mask_TCC1	636
C.16.4	bit_set_full_mask	636
C.16.5	bits_equal	639
C.16.6	bits_equal2	640
C.16.7	bits_equal3	641
C.16.8	bits_equal4	648
C.16.9	clear_bit_TCC1	648
C.16.10	change_bit_single_bit_op	649
C.16.11	change_bit_result	667
C.16.12	change_bits_TCC1	667
C.16.13	change_bits_TCC2	668
C.16.14	change_bits_result	668
C.16.15	change_bits_unchanged	669
C.16.16	change_bits_below_TCC1	670
C.16.17	change_bits_below	670
C.16.18	binary_not_TCC1	672
C.16.19	binary_not_not	672
C.16.20	binary_and_TCC1	673
C.16.21	binary_and_not	673
C.16.22	binary_and_full_TCC1	673
C.16.23	binary_and_full	674
C.16.24	binary_and_zero_TCC1	675
C.16.25	binary_and_zero	675
C.16.26	binary_or_TCC1	676

C.16.27	binary_or_not	676
C.16.28	binary_xor_TCC1	677
C.16.29	binary_xor_self	677
C.16.30	binary_or_full	678
C.16.31	binary_or_zero	679
C.17	Proofs for Bits (bits.pvs)	679
C.17.1	bool_to_nat_below	679
C.17.2	nat_boolIso	680
C.17.3	cut_bit_zero	680
C.17.4	cut_bits_below	680
C.17.5	cut_bits_aligned	680
C.17.6	aligned_cut_bits	681
C.17.7	aligned_add_cut_bits	681
C.17.8	cut_bit_bits	682
C.17.9	cut_bits_zero	683
C.17.10	zero_cut_bits	683
C.17.11	cut_bits_cut_bits_TCC1	683
C.17.12	cut_bits_cut_bits	683
C.17.13	cut_bit_cut_bits	686
C.17.14	shift_bits_left_less_TCC1	687
C.17.15	shift_bits_left_less	687
C.17.16	aligned_shift_bits_ok	688
C.17.17	aligned_shift_bits_reduce_TCC1	688
C.17.18	aligned_shift_bits_reduce	688
C.17.19	overwrite_bits_TCC1	689
C.17.20	overwrite_bits_less	689
C.17.21	overwrite_bit_less	690
C.17.22	overwrite_bool_bit_less	691
C.17.23	overwrite_shift_bits_less	691
C.17.24	aligned_overwrite_shift_bits_more	691
C.17.25	aligned_overwrite_shift_bits_less_TCC1	694
C.17.26	aligned_overwrite_shift_bits_less	695
C.17.27	cut_bits_shift_bits_left_TCC1	698
C.17.28	cut_bits_shift_bits_left_TCC2	698
C.17.29	cut_bits_shift_bits_left	699
C.17.30	cut_bits_shift_bits_left_outside	700
C.17.31	cut_bits_shift_bits_left_overlap_TCC1	701
C.17.32	cut_bits_shift_bits_left_overlap	701
C.17.33	cut_bits_shift_bits_left_in	701
C.17.34	overwrite_shift_bits_zero	702
C.17.35	zero_overwrite_shift_bits	703
C.17.36	bit_split_TCC1	703
C.17.37	bit_split	704
C.17.38	shift_bits_left_cut_bits	705
C.17.39	cut_bits_0_to_below	707
C.17.40	cut_bits_overwrite_bits_disjoined	707
C.17.41	cut_bits_overwrite_bool_bit_disjoined	715
C.17.42	cut_bits_overwrite_bits_contained	715
C.17.43	cut_bits_overwrite_shift_bits_disjoint	724
C.17.44	cut_bits_overwrite_shift_bits_contained_TCC1	724
C.17.45	cut_bits_overwrite_shift_bits_contained	724
C.17.46	cut_bit_overwrite_shift_bits_TCC1	725

C.17.47	cut_bit_overwrite_shift_bits	725
C.17.48	cut_bit_overwrite_bool_bit	726
C.17.49	overwrite_shift_bits_merge	728
C.18	Proofs for BitwiseExpressions (expressions.pvs)	733
C.19	Proofs for Block_Disjoint_Rewrites (plain_memory_rewrites.pvs)	733
C.19.1	different_registers_blocks_disjoint	733
C.20	Proofs for BooleanConversions (conversions.pvs)	733
C.20.1	pointer_to_bool_TCC1	733
C.21	Proofs for CatchAux (statements.pvs)	733
C.22	Proofs for Challenge_Device_Memory (device_memory.pvs)	733
C.22.1	pm_dev_plain	733
C.22.2	pm_dev_states	734
C.22.3	pm_dev_read	734
C.22.4	pm_dev_write	735
C.22.5	pm_dev_read_side_effect	735
C.22.6	pm_dev_write_side_effect	736
C.22.7	pm_dev_ro_rw_addr	736
C.22.8	device_plain_transformers_ok	737
C.22.9	device_plain_unchanged_memory_invariant	738
C.22.10	device_plain_unchanged_memory_invariant_write	758
C.22.11	device_plain_unchanged_memory_write_invariant	762
C.22.12	device_plain_changed_memory_invariant	767
C.22.13	device_plain_read_side_effect_unchanged	768
C.22.14	device_plain_write_side_effect_unchanged	771
C.22.15	device_plain_transformers_ok2	775
C.22.16	device_memory_plain_memory	786
C.23	Proofs for Challenge_Random_Device (random_device.pvs)	788
C.23.1	IMP_Random_Device_TCC1	788
C.23.2	in_blessed_memory_disjoint_device	788
C.23.3	in_blessed_memory_not_device_address	789
C.23.4	random_side_effects_unchanged_invariant	790
C.23.5	random_side_effects_transformers_ok	797
C.23.6	random_side_effects_side_effect_content_unchanged_read	819
C.23.7	random_side_effects_side_effect_content_unchanged_write	824
C.23.8	random_device_plain_memory	829
C.23.9	unaligned_fails_write	830
C.23.10	unaligned_fails_read	831
C.24	Proofs for CommaExpressions (expressions.pvs)	835
C.25	Proofs for Composition_Lifted_Statement_Lifted_Expression (state-transformer.pvs)	835
C.26	Proofs for Composition_Lifted_Statement_Lifted_Statement (state-transformer.pvs)	835
C.27	Proofs for Composition_Lifted_Statement_Simple_Expression (state-transformer.pvs)	835
C.28	Proofs for Composition_Lifted_Statement_Simple_Statement (state-transformer.pvs)	835
C.29	Proofs for Composition_Simple_Expression_Data_Expression (state-transformer.pvs)	836
C.29.1	super_comp_expr_expr_TCC1	836
C.29.2	super_comp_expr_expr_TCC2	836
C.29.3	super_comp_expr_expr	836
C.30	Proofs for Composition_Simple_Expression_Data_Statement (state-transformer.pvs)	837
C.31	Proofs for Composition_Simple_Expression_Simple_Expression (state-transformer.pvs)	837
C.31.1	comp_expr_forget_expr	837
C.31.2	super_comp_expr_forget_expr	837
C.31.3	comp_simple_expr_simple_expr_TCC1	838
C.31.4	comp_simple_expr_simple_expr	838

C.31.5	comp_simple_expr_simple_expr_ok_TCC1	838
C.31.6	comp_simple_expr_simple_expr_ok	839
C.31.7	comp_simple_expr_simple_expr_data_TCC1	839
C.31.8	comp_simple_expr_simple_expr_data	839
C.31.9	comp_simple_expr_simple_expr_state_TCC1	839
C.31.10	comp_simple_expr_simple_expr_state	840
C.32	Proofs for Composition_Simple_Statement_Lifted_Expression (state-transformer.pvs) .	840
C.33	Proofs for Composition_Simple_Statement_Lifted_Statement (state-transformer.pvs) .	840
C.34	Proofs for Composition_Simple_Statement_Simple_Expression (state-transformer.pvs) .	841
C.34.1	comp_simple_stmt_simple_expr_TCC1	841
C.34.2	comp_simple_stmt_simple_expr	841
C.34.3	comp_simple_stmt_simple_expr_ok_TCC1	841
C.34.4	comp_simple_stmt_simple_expr_ok	842
C.34.5	comp_simple_stmt_simple_expr_data_TCC1	842
C.34.6	comp_simple_stmt_simple_expr_data	842
C.34.7	comp_simple_stmt_simple_expr_state_TCC1	843
C.34.8	comp_simple_stmt_simple_expr_state	843
C.35	Proofs for Composition_Simple_Statement_Simple_Statement (state-transformer.pvs) .	843
C.36	Proofs for ConditionalExpression (expressions.pvs)	844
C.37	Proofs for Control_Register_Datatype (paging-data.pvs)	844
C.37.1	cr0_data_type_TCC1	844
C.37.2	cr2_data_type_TCC1	844
C.37.3	cr4_data_type_TCC1	844
C.38	Proofs for Control_Register_Types (paging-data.pvs)	844
C.39	Proofs for Conversions (conversions.pvs)	845
C.40	Proofs for Cpp_Deep_Type_Wellformedness (types.pvs)	845
C.40.1	no_pointers_to_bitfield?_TCC1	845
C.40.2	no_cv_references?_TCC1	845
C.40.3	no_reference_to_reference?_TCC1	845
C.40.4	no_array_of_references?_TCC1	846
C.40.5	no_pointer_or_ref_to_incomplete_array_parameter?_TCC1	846
C.40.6	no_pointer_or_ref_to_incomplete_array_parameter?_TCC2	847
C.40.7	no_pointer_or_ref_to_incomplete_array_parameter?_TCC3	847
C.40.8	no_pointer_or_ref_to_incomplete_array_parameter?_TCC4	848
C.40.9	no_pointer_or_ref_to_incomplete_array_parameter?_TCC5	848
C.40.10	no_pointer_or_ref_to_incomplete_array_parameter?_TCC6	849
C.40.11	bitfield_underlying_integral_or_enum_type?_TCC1	849
C.40.12	enum_underlying_integral?_TCC1	850
C.40.13	enum_constants?_TCC1	850
C.40.14	enum_constants?_TCC2	850
C.40.15	const_volatile?_TCC1	850
C.40.16	const_stutter?_TCC1	851
C.40.17	cv_array?_TCC1	851
C.40.18	cv_base_TCC1	851
C.40.19	cv_base_TCC2	852
C.40.20	cv_base_TCC3	852
C.40.21	cv_base_TCC4	852
C.40.22	cv_base_TCC5	853
C.40.23	subterm_cv_base	853
C.40.24	cv_base_result	858
C.40.25	cv_base_result2	860
C.40.26	cv_base_not_const	860

C.40.27	cv_base_not_volatile	864
C.40.28	cv_base_cv	868
C.40.29	layout_compatible?_TCC1	873
C.40.30	layout_compatible_enums_TCC1	873
C.40.31	layout_compatible_enums_TCC2	873
C.41	Proofs for Cpp_Deep_Types (types.pvs)	874
C.41.1	Cpp_Type_pointer_to_member_eta_TCC1	874
C.41.2	Cpp_Type_pointer_to_member_eta_TCC2	874
C.41.3	Cpp_Type_typ_pointer_to_member_TCC1	874
C.41.4	subterm_less	875
C.41.5	subterm_transitive	881
C.41.6	c_TCC1	885
C.41.7	v_TCC1	886
C.41.8	cv_TCC1	886
C.41.9	cv_TCC2	886
C.41.10	cv_implied	887
C.41.11	pod?_TCC1	888
C.41.12	uidt_void_TCC1	889
C.42	Proofs for Cpp_Examples (cpp-examples.pvs)	889
C.42.1	spec01	889
C.42.2	PRECONDITION_TCC1	890
C.42.3	PRECONDITION_TCC2	891
C.42.4	PRECONDITION_TCC3	892
C.42.5	PRECONDITION_TCC4	892
C.42.6	POSTCONDITION_TCC1	893
C.42.7	spec02_TCC1	894
C.42.8	spec02	894
C.42.9	spec03	895
C.42.10	spec04_TCC1	897
C.42.11	spec04_TCC2	897
C.42.12	spec04	898
C.42.13	spec05	900
C.42.14	spec06_TCC1	901
C.42.15	spec06_TCC2	902
C.42.16	spec06_TCC3	902
C.42.17	spec06_TCC4	902
C.42.18	spec06	903
C.42.19	spec07	904
C.42.20	N_TCC1	906
C.42.21	range_int_values_TCC1	906
C.42.22	range_int_values	907
C.42.23	spec08_TCC1	908
C.42.24	spec08	908
C.42.25	spec09	910
C.42.26	spec10_TCC1	912
C.42.27	spec10	912
C.43	Proofs for Cpp_Integral_Types (types.pvs)	913
C.43.1	range_integral_TCC1	913
C.43.2	range_integral_TCC2	914
C.43.3	range_integral_TCC3	914
C.43.4	range_integral_TCC4	914
C.43.5	range_integral_TCC5	915

C.43.6	range_integral_TCC6	915
C.43.7	range_floating_point_TCC1	916
C.43.8	dt_integral_TCC1	916
C.43.9	dt_integral_TCC2	918
C.43.10	dt_integral_TCC3	920
C.43.11	dt_integral_TCC4	922
C.43.12	dt_integral_TCC5	924
C.43.13	dt_integral_TCC6	925
C.43.14	dt_integral_TCC7	927
C.43.15	dt_integral_TCC8	929
C.43.16	dt_integral_TCC9	930
C.43.17	dt_integral_TCC10	932
C.43.18	dt_integral_TCC11	934
C.43.19	dt_integral_TCC12	936
C.43.20	dt_floating_point_TCC1	938
C.43.21	dt_floating_point_TCC2	940
C.43.22	dt_floating_point_TCC3	942
C.43.23	range_integral_unsigned_TCC1	943
C.43.24	range_integral_unsigned	944
C.43.25	max_value_TCC1	945
C.43.26	max_value_TCC2	946
C.43.27	min_value_TCC1	946
C.43.28	max_value_unsigned	947
C.43.29	min_value_unsigned	948
C.43.30	max_value_bits_TCC1	949
C.43.31	max_value_bits_TCC2	949
C.43.32	max_value_bits_TCC3	949
C.43.33	max_value_bits_TCC4	950
C.43.34	max_value_bits_TCC5	950
C.43.35	max_value_bits_TCC6	950
C.44	Proofs for Cpp_Types (types.pvs)	951
C.44.1	alignment_TCC1	951
C.44.2	alignment_TCC2	951
C.44.3	alignment_TCC3	952
C.44.4	alignment_TCC4	952
C.44.5	alignment_TCC5	952
C.44.6	alignment_TCC6	952
C.44.7	alignment_TCC7	953
C.44.8	alignment_TCC8	953
C.44.9	alignment_TCC9	953
C.44.10	alignment_TCC10	953
C.44.11	alignment_TCC11	954
C.44.12	alignment_TCC12	954
C.44.13	alignment_TCC13	954
C.44.14	pointer_to_aligned_object_TCC1	955
C.44.15	pointer_to_aligned_object_TCC2	955
C.44.16	pointer_to_aligned_object_TCC3	958
C.44.17	align_array_of_type_TCC1	958
C.44.18	align_array_of_type_TCC2	958
C.44.19	range_TCC1	959
C.44.20	range_TCC2	959
C.44.21	range_TCC3	959

C.44.22	range_TCC4	960
C.44.23	range_TCC5	961
C.44.24	range_TCC6	961
C.44.25	range_TCC7	962
C.44.26	range_cv_base_TCC1	962
C.44.27	range_cv_base	965
C.44.28	dt_non_bool_integral_enum_TCC1	968
C.44.29	dt_non_bool_integral_enum_TCC2	968
C.44.30	dt_non_bool_integral_enum_TCC3	971
C.44.31	dt_non_bool_integral_enum_TCC4	971
C.44.32	dt_non_bool_integral_enum_pod	973
C.44.33	dt_TCC1	978
C.44.34	dt_TCC2	990
C.44.35	dt_TCC3	992
C.44.36	dt_TCC4	997
C.44.37	dt_TCC5	1000
C.44.38	dt_pod	1000
C.44.39	dt_cv_pointer_TCC1	1013
C.44.40	dt_cv_pointer_TCC2	1014
C.44.41	dt_cv_pointer_TCC3	1014
C.44.42	dt_cv_pointer_TCC4	1015
C.44.43	dt_cv_pointer_TCC5	1015
C.44.44	dt_cv_pointer_TCC6	1016
C.44.45	dt_cv_pointer_TCC7	1016
C.44.46	dt_cv_pointer_TCC8	1020
C.44.47	dt_cv_pointer_TCC9	1020
C.44.48	dt_cv_pointer_TCC10	1021
C.44.49	dt_cv_pointer_TCC11	1021
C.44.50	dt_cv_pointer_TCC12	1021
C.44.51	dt_cv_pointer_TCC13	1022
C.44.52	dt_cv_pointer_TCC14	1025
C.44.53	dt_cv_pointer_TCC15	1026
C.44.54	dt_cv_bitfield_TCC1	1027
C.44.55	dt_cv_bitfield_TCC2	1027
C.44.56	dt_cv_bitfield_TCC3	1028
C.44.57	dt_cv_bitfield_TCC4	1028
C.44.58	dt_cv_bitfield_TCC5	1028
C.44.59	dt_cv_bitfield_TCC6	1029
C.44.60	dt_cv_bitfield_TCC7	1030
C.44.61	dt_cv_bitfield_TCC8	1033
C.44.62	dt_cv_bitfield_TCC9	1033
C.44.63	dt_cv_bitfield_TCC10	1034
C.44.64	dt_cv_bitfield_TCC11	1034
C.44.65	dt_cv_bitfield_TCC12	1035
C.44.66	dt_cv_bitfield_TCC13	1035
C.44.67	dt_cv_bitfield_TCC14	1039
C.44.68	dt_cv_bitfield_TCC15	1039
C.44.69	dt_cv_ptm_TCC1	1040
C.44.70	dt_cv_ptm_TCC2	1040
C.44.71	dt_cv_ptm_TCC3	1041
C.44.72	dt_cv_ptm_TCC4	1042
C.44.73	dt_cv_ptm_TCC5	1042

C.44.74	dt_cv_ptm_TCC6	1042
C.44.75	dt_cv_ptm_TCC7	1043
C.44.76	dt_cv_ptm_TCC8	1047
C.44.77	dt_cv_ptm_TCC9	1047
C.44.78	dt_cv_ptm_TCC10	1048
C.44.79	dt_cv_ptm_TCC11	1048
C.44.80	dt_cv_ptm_TCC12	1049
C.44.81	dt_cv_ptm_TCC13	1049
C.44.82	dt_cv_ptm_TCC14	1052
C.44.83	dt_cv_ptm_TCC15	1053
C.44.84	dt_cv_float_TCC1	1054
C.44.85	dt_cv_float_TCC2	1054
C.44.86	dt_cv_float_TCC3	1055
C.44.87	dt_cv_float_TCC4	1055
C.44.88	dt_cv_float_TCC5	1056
C.44.89	dt_cv_float_TCC6	1056
C.44.90	dt_cv_float_TCC7	1057
C.44.91	dt_cv_float_TCC8	1061
C.44.92	dt_cv_float_TCC9	1061
C.44.93	dt_cv_float_TCC10	1062
C.44.94	dt_cv_float_TCC11	1062
C.44.95	dt_cv_float_TCC12	1063
C.44.96	dt_cv_float_TCC13	1064
C.44.97	dt_cv_float_TCC14	1067
C.44.98	dt_cv_float_TCC15	1068
C.44.99	uidt_TCC1	1069
C.44.100	uidt_TCC2	1069
C.44.101	uidt_TCC3	1069
C.44.102	uidt_TCC4	1070
C.44.103	uidt_TCC5	1070
C.44.104	uidt_TCC6	1070
C.44.105	uidt_TCC7	1071
C.44.106	uidt_TCC8	1071
C.44.107	uidt_TCC9	1071
C.44.108	uidt_TCC10	1071
C.44.109	uidt_TCC11	1072
C.44.110	uidt_TCC12	1072
C.44.111	uidt_TCC13	1072
C.44.112	uidt_TCC14	1073
C.44.113	uidt_TCC15	1073
C.44.114	uidt_TCC16	1073
C.44.115	uidt_TCC17	1074
C.44.116	uidt_TCC18	1074
C.44.117	uidt_TCC19	1075
C.44.118	uidt_TCC20	1075
C.44.119	uidt_TCC21	1075
C.44.120	uidt_TCC22	1076
C.44.121	uidt_TCC23	1076
C.44.122	uidt_TCC24	1076
C.44.123	uidt_TCC25	1077
C.44.124	uidt_TCC26	1077
C.44.125	uidt_TCC27	1077

C.44.126	uidt_TCC28	1078
C.44.127	uidt_TCC29	1078
C.44.128	uidt_TCC30	1078
C.44.129	uidt_TCC31	1079
C.44.130	uidt_TCC32	1079
C.44.131	uidt_TCC33	1079
C.44.132	uidt_TCC34	1080
C.44.133	uidt_TCC35	1080
C.44.134	uidt_TCC36	1080
C.44.135	uidt_TCC37	1081
C.44.136	uidt_TCC38	1081
C.44.137	uidt_TCC39	1082
C.44.138	uidt_TCC40	1082
C.44.139	uidt_TCC41	1083
C.44.140	mod_float_TCC1	1083
C.44.141	value_bitmask_TCC1	1084
C.45	Proofs for Cpp_Verification (cpp-verification.pvs)	1084
C.46	Proofs for Cpp_array (types.pvs)	1084
C.46.1	uidt_array_TCC1	1084
C.47	Proofs for Cpp_bitfield (types.pvs)	1085
C.47.1	range_bitfield_TCC1	1085
C.47.2	bitfield_range_underlying_TCC1	1086
C.47.3	bitfield_range_underlying_TCC2	1086
C.47.4	bitfield_range_underlying_TCC3	1086
C.47.5	dt_bitfield_TCC1	1087
C.48	Proofs for Cpp_bool (types.pvs)	1087
C.48.1	range_bool_TCC1	1087
C.48.2	dt_bool_TCC1	1088
C.49	Proofs for Cpp_char (types.pvs)	1088
C.49.1	dt_char_TCC1	1088
C.49.2	value_bitmask_char_TCC1	1088
C.49.3	min_char	1088
C.49.4	max_char	1089
C.49.5	char_binary	1091
C.50	Proofs for Cpp_class (types.pvs)	1094
C.50.1	uidt_class_TCC1	1094
C.51	Proofs for Cpp_const_volatile (types.pvs)	1095
C.51.1	dt_volatile_TCC1	1095
C.52	Proofs for Cpp_double (types.pvs)	1095
C.52.1	dt_double_TCC1	1095
C.53	Proofs for Cpp_enum (types.pvs)	1096
C.53.1	valid_range?_TCC1	1096
C.53.2	valid_range?_TCC2	1096
C.53.3	range_enum_TCC1	1096
C.53.4	dt_enum_TCC1	1097
C.53.5	value_bitmask_enum_TCC1	1098
C.53.6	enum_size_TCC1	1099
C.53.7	enum_size_TCC2	1099
C.54	Proofs for Cpp_float (types.pvs)	1099
C.54.1	range_float_TCC1	1099
C.54.2	dt_float_TCC1	1100
C.55	Proofs for Cpp_function (types.pvs)	1100

C.55.1	uidt_function_TCC1	1100
C.55.2	dt_vtable_TCC1	1100
C.56	Proofs for Cpp_int (types.pvs)	1101
C.56.1	range_int_TCC1	1101
C.56.2	dt_int_TCC1	1102
C.56.3	value_bitmask_int_TCC1	1102
C.57	Proofs for Cpp_long (types.pvs)	1102
C.57.1	range_long_TCC1	1102
C.57.2	dt_long_TCC1	1103
C.57.3	value_bitmask_long_TCC1	1103
C.58	Proofs for Cpp_longdouble (types.pvs)	1103
C.58.1	dt_longdouble_TCC1	1103
C.59	Proofs for Cpp_longlong (types.pvs)	1103
C.59.1	range_longlong_TCC1	1103
C.59.2	dt_longlong_TCC1	1104
C.59.3	value_bitmask_longlong_TCC1	1104
C.60	Proofs for Cpp_pointer (types.pvs)	1104
C.60.1	dt_pointer_TCC1	1104
C.60.2	pointers_to_layout_compatible_range_TCC1	1105
C.60.3	pointers_to_layout_compatible_range_TCC2	1105
C.60.4	pointers_to_layout_compatible_range_TCC3	1106
C.60.5	pointers_to_layout_compatible_range_TCC4	1106
C.60.6	pointers_to_layout_compatible_dt_TCC1	1106
C.60.7	void_pointer_char_pointer_dt_TCC1	1107
C.60.8	add_TCC1	1107
C.60.9	add_spec2_TCC1	1107
C.60.10	add_spec2_TCC2	1107
C.60.11	add_simplification	1108
C.60.12	void_pointer_other_pointer_value_TCC1	1110
C.60.13	void_pointer_other_pointer_value_TCC2	1110
C.61	Proofs for Cpp_pointer_to_member (types.pvs)	1111
C.61.1	dt_ptm_TCC1	1111
C.62	Proofs for Cpp_reference (types.pvs)	1112
C.62.1	dt_reference_TCC1	1112
C.63	Proofs for Cpp_schar (types.pvs)	1112
C.63.1	range_schar_TCC1	1112
C.63.2	dt_schar_TCC1	1113
C.63.3	schar_value_eq_uchar_value_TCC1	1113
C.63.4	value_bitmask_schar_TCC1	1113
C.64	Proofs for Cpp_short (types.pvs)	1114
C.64.1	range_short_TCC1	1114
C.64.2	dt_short_TCC1	1114
C.64.3	value_bitmask_short_TCC1	1114
C.65	Proofs for Cpp_uchar (types.pvs)	1115
C.65.1	range_uchar_TCC1	1115
C.65.2	dt_uchar_TCC1	1115
C.65.3	value_bitmask_uchar_TCC1	1115
C.66	Proofs for Cpp_uint (types.pvs)	1116
C.66.1	range_uint_TCC1	1116
C.66.2	dt_uint_TCC1	1116
C.66.3	uint_representation_same_for_positive_int_TCC1	1116
C.66.4	value_bitmask_uint_TCC1	1117

C Proof scripts

C.67	Proofs for Cpp_ulong (types.pvs)	1117
C.67.1	range_ulong_TCC1	1117
C.67.2	dt_ulong_TCC1	1117
C.67.3	ulong_represenation_same_for_positive_long_TCC1	1118
C.67.4	value_bitmask_ulong_TCC1	1118
C.68	Proofs for Cpp_ulonglong (types.pvs)	1118
C.68.1	range_ulonglong_TCC1	1118
C.68.2	dt_ulonglong_TCC1	1119
C.68.3	ulonglong_represenation_same_for_positive_longlong_TCC1	1119
C.68.4	value_bitmask_ulonglong_TCC1	1119
C.69	Proofs for Cpp_union (types.pvs)	1120
C.69.1	uidt_union_TCC1	1120
C.70	Proofs for Cpp_ushort (types.pvs)	1120
C.70.1	range_ushort_TCC1	1120
C.70.2	dt_ushort_TCC1	1121
C.70.3	ushort_represenation_same_for_positive_short_TCC1	1121
C.70.4	value_bitmask_ushort_TCC1	1121
C.71	Proofs for Cpp_wchar_t (types.pvs)	1122
C.71.1	range_wchar_t_TCC1	1122
C.71.2	dt_wchar_t_TCC1	1122
C.71.3	value_bitmask_wchar_t_TCC1	1122
C.72	Proofs for Datatype_Model (datatype_model.pvs)	1123
C.72.1	uidt_1_TCC1	1123
C.72.2	pod_1_TCC1	1123
C.72.3	pod_1_TCC2	1123
C.72.4	pod_1_TCC3	1123
C.72.5	pod_1_TCC4	1124
C.72.6	uidt_2_TCC1	1124
C.72.7	pod_2_TCC1	1124
C.72.8	pod_2_TCC2	1125
C.72.9	pod_2_TCC3	1125
C.73	Proofs for Datatype_Model_2 (datatype_model.pvs)	1125
C.73.1	uidt_bool_TCC1	1125
C.73.2	pod_bool_TCC1	1126
C.73.3	pod_bool_TCC2	1127
C.73.4	pod_bool_TCC3	1127
C.73.5	pod_bool_TCC4	1127
C.73.6	pod_bool_TCC5	1127
C.73.7	pod_bool_TCC6	1128
C.73.8	uidt_Byte_TCC1	1129
C.73.9	uidt_Byte_TCC2	1129
C.73.10	pod_Byte_TCC1	1129
C.73.11	pod_Byte_TCC2	1130
C.73.12	pod_Byte_TCC3	1130
C.73.13	pod_Byte_TCC4	1130
C.73.14	pod_Byte_TCC5	1131
C.74	Proofs for DeclarationStatements1 (statements.pvs)	1131
C.75	Proofs for DeclarationStatements2 (statements.pvs)	1131
C.76	Proofs for Device_Memory (device_memory.pvs)	1131
C.77	Proofs for EqualityExpressions (expressions.pvs)	1131
C.78	Proofs for Error_Code_Types (result.pvs)	1131
C.79	Proofs for Even_More_List_Props (vfiasco-prelude.pvs)	1132

C.79.1	every_extend	1132
C.80	Proofs for Exception_type (result.pvs)	1132
C.81	Proofs for Exception_type_adt (Exception_type_adt.pvs)	1132
C.82	Proofs for Exception_type_adt_reduce (Exception_type_adt.pvs)	1133
C.83	Proofs for Expand_State (device_memory.pvs)	1133
C.83.1	em_transformers_ok	1133
C.83.2	em_transformer_invariant	1133
C.83.3	em_lift_singleton	1133
C.84	Proofs for Expand_State2 (device_memory.pvs)	1134
C.84.1	em_lift_expr_2_super	1134
C.85	Proofs for Expand_State3 (device_memory.pvs)	1134
C.85.1	em_memory_read_transformers	1134
C.85.2	em_memory_write_transformers	1135
C.85.3	em_unchanged_memory_invariant	1137
C.85.4	em_changed_memory_invariant	1137
C.85.5	em_const_unchanged_memory_invariant_TCC1	1139
C.85.6	em_const_unchanged_memory_invariant	1139
C.85.7	em_side_effect_content_unchanged	1140
C.86	Proofs for ExprResult (result.pvs)	1140
C.87	Proofs for ExprResult_adt (ExprResult_adt.pvs)	1140
C.88	Proofs for ExprResult_adt_map (ExprResult_adt.pvs)	1140
C.89	Proofs for ExprResult_adt_reduce (ExprResult_adt.pvs)	1141
C.90	Proofs for ExpressionStatements (statements.pvs)	1141
C.91	Proofs for Expression_Composition_Rewrites (state-transformer.pvs)	1141
C.91.1	comp_eval_if_ok_fexpr	1141
C.92	Proofs for Expressions (expressions.pvs)	1141
C.93	Proofs for Expt_Lemmas (vfiasco-prelude.pvs)	1141
C.93.1	both_sides_expt_gt1_le	1141
C.93.2	expt_plus	1142
C.94	Proofs for Extended_Real (vfiasco-prelude.pvs)	1142
C.94.1	number_field_extended	1142
C.94.2	er_plus_TCC1	1142
C.94.3	er_neg_TCC1	1142
C.94.4	er_plus_number_field	1143
C.94.5	er_minus_number_field	1143
C.94.6	er_times_number_field	1143
C.94.7	er_div_number_field	1143
C.94.8	er_neg_number_field	1144
C.94.9	er_plus_real	1144
C.94.10	er_minus_real	1144
C.94.11	er_times_real	1144
C.94.12	er_div_real	1145
C.94.13	er_neg_real	1145
C.94.14	er_plus_rational	1145
C.94.15	er_minus_rational	1145
C.94.16	er_times_rational	1146
C.94.17	er_div_rational	1146
C.94.18	er_neg_rational	1146
C.94.19	er_plus_integer	1146
C.94.20	er_minus_integer	1147
C.94.21	er_times_integer	1147
C.94.22	er_neg_integer	1147

C Proof scripts

C.95	Proofs for Finite_Set_Reduce (vfiasco-prelude.pvs)	1147
C.95.1	list_of_finite_set_TCC1	1147
C.95.2	list_of_finite_set_TCC2	1148
C.95.3	length_list_of_finite_set	1148
C.95.4	member_list_of_finite_set	1149
C.95.5	unique_member_list_of_finite_set	1151
C.96	Proofs for General (constants.pvs)	1156
C.96.1	Byte_TCC1	1156
C.96.2	less_than_max_byte	1156
C.97	Proofs for Graph (graph.pvs)	1156
C.97.1	path?_TCC1	1156
C.97.2	path?_TCC2	1157
C.97.3	path?_TCC3	1157
C.97.4	path?_TCC4	1157
C.97.5	nodes_path_start	1157
C.97.6	nodes_path_end	1158
C.97.7	path_head_TCC1	1158
C.97.8	path_head	1159
C.97.9	path_tail_TCC1	1159
C.97.10	path_tail	1160
C.97.11	path_cdr_TCC1	1160
C.97.12	path_cdr	1161
C.97.13	path_concat_path	1161
C.97.14	trans_closure_char_graph	1164
C.97.15	path_cycle_free	1167
C.97.16	different_nodes_on_edge	1171
C.97.17	tree_path	1172
C.97.18	root_path_TCC1	1177
C.97.19	path_length_bound_TCC1	1177
C.97.20	path_length_bound	1177
C.97.21	non_root_path	1178
C.97.22	non_root_path_extend_TCC1	1180
C.97.23	non_root_path_extend	1180
C.97.24	tree_path_to_root	1182
C.97.25	tree_unique_path_to_root	1187
C.97.26	root_path_singleton	1196
C.97.27	path_to_root_TCC1	1197
C.97.28	path_factor_TCC1	1197
C.97.29	path_factor_TCC2	1197
C.97.30	path_factor_TCC3	1198
C.97.31	path_factor_TCC4	1198
C.97.32	path_factor	1199
C.97.33	node_pair_distinction_TCC1	1206
C.97.34	node_pair_distinction	1207
C.97.35	symmetric_node_pair_distinction_TCC1	1226
C.97.36	symmetric_node_pair_distinction	1227
C.98	Proofs for Graph_Relations (graph.pvs)	1228
C.98.1	trans_closure_char_TCC1	1228
C.98.2	trans_closure_char_TCC2	1228
C.98.3	trans_closure_char_TCC3	1228
C.98.4	trans_closure_char	1229
C.99	Proofs for Graph_Util (graph.pvs)	1238

C.99.1	every_nth	1238
C.99.2	nth_append_left_TCC1	1240
C.99.3	nth_append_left	1241
C.99.4	nth_append_right_TCC1	1242
C.99.5	nth_append_right	1242
C.99.6	list_pred_card	1243
C.99.7	path_start_TCC1	1247
C.99.8	path_end_TCC1	1247
C.99.9	path_list_induction_TCC1	1247
C.99.10	path_list_induction_TCC2	1248
C.99.11	path_list_induction	1248
C.99.12	concat_path_TCC1	1249
C.99.13	concat_path_TCC2	1249
C.99.14	length_concat_path	1250
C.100	Proofs for Hoare (hoare.pvs)	1250
C.100.1	valid_TCC1	1250
C.100.2	valid_TCC2	1251
C.100.3	valid_and	1251
C.100.4	valid_implied	1251
C.100.5	Valid_and	1252
C.101	Proofs for IA32 (constants.pvs)	1252
C.101.1	max_linear_offset_val	1252
C.101.2	mem_addr_4g_TCC1	1252
C.102	Proofs for Import_All (everything.pvs)	1252
C.103	Proofs for IntegralConversions (conversions.pvs)	1252
C.104	Proofs for IntegralPromotions (conversions.pvs)	1253
C.104.1	int_conversion_TCC1	1253
C.104.2	int_conversion_spec_TCC1	1253
C.104.3	int_conversion_TCC2	1254
C.104.4	int_conversion_TCC3	1255
C.104.5	bool2int_TCC1	1255
C.104.6	bool2int_TCC2	1256
C.105	Proofs for Interpreted_Data (abstract_data.pvs)	1256
C.105.1	interpreted_data_type?_TCC1	1256
C.105.2	pod_is_interpreted_data	1257
C.105.3	pod_size	1257
C.105.4	data_type_length	1257
C.105.5	length_to_byte	1257
C.105.6	valid_from_byte	1258
C.105.7	valid_iff_from_byte	1258
C.105.8	valid_to_byte	1258
C.105.9	up_from_byte_to_byte	1258
C.105.10	from_byte_to_byte	1259
C.105.11	in_blessed_memory_subset	1259
C.106	Proofs for Interpreted_Data_Lift (abstract_data.pvs)	1259
C.107	Proofs for IterationStatements1 (statements.pvs)	1259
C.107.1	iterate_while_TCC1	1259
C.107.2	iterate_while_TCC2	1260
C.107.3	while_termination_point?_TCC1	1260
C.107.4	while_TCC1	1260
C.107.5	nonzero_min_while_termination_point	1260
C.107.6	while_as_if_while	1261

C.107.7	iterate_do_as_while	1282
C.107.8	do_termination_point?_TCC1	1283
C.107.9	termination_point_do_as_while_TCC1	1283
C.107.10	termination_point_do_as_while	1283
C.107.11	do_while_TCC1	1284
C.107.12	do_as_while	1284
C.107.13	do_while_unroll	1287
C.108	Proofs for IterationStatements2 (statements.pvs)	1287
C.108.1	iterate_for_TCC1	1287
C.108.2	iterate_for_TCC2	1287
C.108.3	iterate_for_as_while_TCC1	1288
C.108.4	iterate_for_as_while	1288
C.108.5	for_termination_point?_TCC1	1290
C.108.6	termination_point_for_as_while_TCC1	1291
C.108.7	termination_point_for_as_while	1292
C.108.8	min_termination_point_for_as_while_TCC1	1292
C.108.9	min_termination_point_for_as_while_TCC2	1292
C.108.10	min_termination_point_for_as_while_TCC3	1294
C.108.11	min_termination_point_for_as_while	1295
C.108.12	nonzero_min_for_termination_point_TCC1	1296
C.108.13	nonzero_min_for_termination_point	1297
C.108.14	for_as_while	1297
C.108.15	for_unroll	1324
C.109	Proofs for IterationStatements3 (statements.pvs)	1325
C.109.1	do_while_inv_unroll	1325
C.110	Proofs for IterationStatements4 (statements.pvs)	1325
C.110.1	for_inv_unroll	1325
C.111	Proofs for JumpStatements1 (statements.pvs)	1326
C.112	Proofs for JumpStatements2 (statements.pvs)	1326
C.113	Proofs for LabeledStatements (statements.pvs)	1326
C.114	Proofs for Linear_Memory (linear_memory.pvs)	1326
C.114.1	xlat_idx_TCC1	1326
C.114.2	xlat_idx_TCC2	1327
C.114.3	xlat_idx_pe_aligned_TCC1	1327
C.114.4	xlat_idx_pe_aligned_TCC2	1327
C.114.5	xlat_idx_pe_aligned	1327
C.114.6	xlat_idx_memory_address	1331
C.114.7	xlat_idx_memory_address2	1332
C.114.8	xlat_ofs_memory_address	1333
C.114.9	xlat_ofs_memory_address2	1333
C.114.10	raise_fault_TCC1	1334
C.114.11	translate_TCC1	1335
C.114.12	translate_memory_address	1335
C.114.13	translate_memory_address2	1340
C.114.14	translate_result_leaf_TCC1	1346
C.114.15	translate_result_leaf_TCC2	1346
C.114.16	translate_result_leaf	1346
C.114.17	translate_result_no_leaf_TCC1	1347
C.114.18	translate_result_no_leaf	1347
C.114.19	linear_resolve_TCC1	1348
C.114.20	linear_resolve_TCC2	1348
C.114.21	linear_resolve_TCC3	1349

C.114.22	linear_resolve_TCC4	1349
C.114.23	linear_resolve_TCC5	1350
C.114.24	linear_resolve_TCC6	1350
C.114.25	linear_resolve_TCC7	1351
C.114.26	linear_resolve_TCC8	1351
C.114.27	linear_resolve_memory_address	1352
C.114.28	linear_read_TCC1	1361
C.114.29	linear_read_TCC2	1361
C.114.30	linear_read_TCC3	1361
C.114.31	linear_read_side_effect_TCC1	1362
C.115	Proofs for Linear_Memory_Blessing (challenge-linear.pvs)	1364
C.115.1	PTab_Address_TCC1	1364
C.115.2	pe_in_pdir_range?_TCC1	1364
C.115.3	pe_in_ptab_range?_TCC1	1364
C.115.4	pe_in_ptab_range?_TCC2	1364
C.115.5	pe_in_ptab_range?_TCC3	1365
C.115.6	pe_in_ptab_range?_TCC4	1365
C.115.7	pe_in_pt_range?_TCC1	1366
C.115.8	pe_in_pt_range?_TCC2	1366
C.115.9	pe_in_pt_range_aligned	1366
C.115.10	pe_in_pt_address_in_pt	1367
C.115.11	is_linear_plain_memory?_TCC1	1369
C.115.12	is_linear_plain_memory?_TCC2	1370
C.115.13	is_linear_plain_memory?_TCC3	1370
C.115.14	is_linear_plain_memory?_TCC4	1371
C.115.15	is_linear_plain_memory?_TCC5	1371
C.115.16	is_linear_plain_memory?_TCC6	1373
C.116	Proofs for Linear_Memory_Blessing_Properties (challenge-linear.pvs)	1377
C.116.1	linear_plain_transformers_ok	1377
C.116.2	linear_plain_unchanged_memory_invariant	1409
C.116.3	linear_plain_unchanged_memory_invariant_write	1472
C.116.4	linear_plain_unchanged_memory_write_invariant	1488
C.116.5	linear_plain_transformer_invariant	1509
C.116.6	linear_plain_changed_memory_invariant	1511
C.116.7	linear_plain_read_side_effect_unchanged	1538
C.116.8	linear_plain_write_side_effect_unchanged	1551
C.116.9	linear_memory_plain_memory	1562
C.117	Proofs for Linear_Memory_Properties (challenge-linear.pvs)	1563
C.117.1	pm_read_linear	1563
C.117.2	pm_write_linear	1563
C.117.3	pm_read_side_effect_linear	1563
C.117.4	pm_write_side_effect_linear	1564
C.117.5	pm_states	1564
C.117.6	pm_plain_phy	1564
C.117.7	pm_memory_addr_TCC1	1565
C.117.8	pm_memory_addr_TCC2	1565
C.117.9	pm_memory_addr	1565
C.117.10	pm_linear_blessed	1566
C.117.11	pm_linear_resolve_read_ok_TCC1	1566
C.117.12	pm_linear_resolve_read_ok	1566
C.117.13	pm_linear_resolve_write_ok_TCC1	1567
C.117.14	pm_linear_resolve_write_ok	1567

C.117.15	pm_linear_resolve_reg_transformers_other	1567
C.117.16	pm_unchanged_singleton_linear_resolve_reg_transformers	1568
C.117.17	pm_address_in_pt_range_in_rw_addr	1572
C.117.18	address_block_in_memory_TCC1	1572
C.117.19	address_block_in_memory	1573
C.117.20	address_block_in_memory2	1574
C.117.21	raise_fault_transformer_invariant	1574
C.117.22	raise_fault_unchanged_memory_invariant	1577
C.117.23	apply_side_effects_pm_states	1583
C.117.24	apply_side_effects_ok	1586
C.117.25	apply_side_effects_same_result_TCC1	1589
C.117.26	apply_side_effects_same_result	1589
C.117.27	apply_side_effects_unchanged	1614
C.117.28	subset_ptab_address_rw_addr	1626
C.117.29	translate_transformer_invariant	1627
C.117.30	translate_unchanged_pm_phy_except_pe_TCC1	1637
C.117.31	translate_unchanged_pm_phy_except_pe_TCC2	1637
C.117.32	translate_unchanged_pm_phy_except_pe	1638
C.117.33	translate_result_ok_TCC1	1658
C.117.34	translate_result_ok	1659
C.117.35	translate_result_as_read_TCC1	1675
C.117.36	translate_result_as_read_TCC2	1675
C.117.37	translate_result_as_read	1676
C.117.38	translate_same_result	1676
C.117.39	xlat_pe_in_pdir_range_TCC1	1677
C.117.40	xlat_pe_in_pdir_range_TCC2	1677
C.117.41	xlat_pe_in_pdir_range_TCC3	1678
C.117.42	xlat_pe_in_pdir_range	1684
C.117.43	xlat_pe_in_pt_range_TCC1	1686
C.117.44	xlat_pe_in_pt_range_TCC2	1687
C.117.45	xlat_pe_in_pt_range_TCC3	1687
C.117.46	xlat_pe_in_pt_range_TCC4	1688
C.117.47	xlat_pe_in_pt_range_TCC5	1690
C.117.48	xlat_pe_in_pt_range	1704
C.117.49	linear_resolve_unchanged_pm_phy	1734
C.117.50	linear_resolve_read_transformers_ok	1914
C.117.51	linear_resolve_write_transformers_ok	1914
C.117.52	linear_resolve_transformer_invariant	1915
C.117.53	linear_resolve_states_TCC1	1915
C.117.54	linear_resolve_states	1916
C.117.55	linear_resolve_same_result	1916
C.117.56	delta_memory_address	2012
C.117.57	xlat_idx_same_page_address_TCC1	2014
C.117.58	xlat_idx_same_page_address_TCC2	2015
C.117.59	xlat_idx_same_page_address	2015
C.117.60	xlat_ofs_same_page_address_TCC1	2028
C.117.61	xlat_ofs_same_page_address_TCC2	2028
C.117.62	xlat_ofs_same_page_address	2029
C.117.63	translate_same_page_address_ok	2036
C.117.64	translate_same_page_address_result	2039
C.117.65	linear_resolve_same_page_address_ok_TCC1	2043
C.117.66	linear_resolve_same_page_address_ok	2043

C.117.67	linear_resolve_same_page_address_TCC1	2079
C.117.68	linear_resolve_same_page_address	2079
C.117.69	address_block_split_type	2079
C.117.70	same_page_address_block_read_TCC1	2083
C.117.71	same_page_address_block_read	2083
C.117.72	same_page_address_block_write_TCC1	2098
C.117.73	same_page_address_block_write	2099
C.117.74	split_linear_read_side_effects_states_TCC1	2113
C.117.75	split_linear_read_side_effects_states	2114
C.117.76	split_linear_write_side_effects_states_TCC1	2133
C.117.77	split_linear_write_side_effects_states	2134
C.117.78	split_linear_read_side_effects_ok	2155
C.117.79	split_linear_write_side_effects_ok	2178
C.117.80	split_linear_read_side_effects_data	2202
C.117.81	split_linear_write_side_effects_data	2224
C.117.82	split_linear_read_side_effects_unchanged_memory	2250
C.117.83	split_linear_write_side_effects_unchanged_memory	2294
C.117.84	linear_unchanged_invariant	2320
C.117.85	pm_phy_read_after_resolve_ok_TCC1	2355
C.117.86	pm_phy_read_after_resolve_ok	2355
C.117.87	pm_resolve_address	2355
C.117.88	pm_resolve_address_write	2358
C.118	Proofs for List_Split (linear_memory.pvs)	2361
C.118.1	split_TCC1	2361
C.118.2	split_TCC2	2361
C.118.3	split_pair_cross_size	2363
C.118.4	split_pair_concat	2369
C.118.5	split_range	2373
C.118.6	split_no_null	2380
C.118.7	split_null	2382
C.118.8	split_type_TCC1	2382
C.118.9	split_type_TCC2	2383
C.118.10	split_type	2383
C.119	Proofs for LogicalExpressions (expressions.pvs)	2385
C.120	Proofs for LvalueToRvalueConversion (conversions.pvs)	2385
C.121	Proofs for Memory (memory.pvs)	2386
C.121.1	memory_write_list_nse_TCC1	2386
C.121.2	memory_write_list_nse_TCC2	2386
C.121.3	memory_read_list_nse_TCC1	2386
C.121.4	memory_read_list_nse_TCC2	2386
C.121.5	memory_read_list_ok_length	2386
C.121.6	memory_read_transformers_memory_read	2387
C.121.7	memory_read_transformers_mono	2388
C.121.8	memory_read_transformers_union	2388
C.121.9	memory_write_transformers_memory_write	2388
C.121.10	memory_write_transformers_mono	2389
C.121.11	memory_read_side_effect_super_transformers_memory_read_side_effect	2389
C.121.12	memory_read_side_effect_super_transformers_mono	2389
C.121.13	memory_write_side_effect_super_transformers_memory_write_side_effect	2390
C.121.14	memory_write_side_effect_super_transformers_mono	2390
C.121.15	side_effect_content_unchanged_content	2390
C.121.16	side_effect_content_unchanged_mono	2391

C.121.17	side_effect_content_unchanged_composition	2391
C.122	Proofs for Memory_Change (memory.pvs)	2393
C.122.1	unchanged_memory_invariant?_TCC1	2393
C.122.2	unchanged_memory_invariant_unchanged_TCC1	2394
C.122.3	unchanged_memory_invariant_unchanged	2394
C.122.4	unchanged_memory_invariant_invariant	2394
C.122.5	unchanged_memory_invariant_mono	2395
C.122.6	unchanged_memory_invariant_next_ok_TCC1	2395
C.122.7	unchanged_memory_invariant_next_ok	2395
C.122.8	unchanged_memory_invariant_union_transformers	2396
C.122.9	unchanged_memory_invariant_union_addresses	2400
C.122.10	unchanged_memory_invariant_empty	2401
C.122.11	unchanged_memory_invariant_all_transformers	2401
C.123	Proofs for Memory_Change_2 (memory.pvs)	2402
C.123.1	unchanged_memory_write_invariant_mono_addresses	2402
C.123.2	unchanged_memory_write_invariant_transformer_invariant	2403
C.123.3	unchanged_memory_write_invariant_unchanged_memory_invariant	2404
C.123.4	unchanged_memory_invariant_union	2408
C.123.5	transformer_invariant_write_list_nse	2409
C.123.6	transformer_invariant_write_list	2414
C.123.7	changed_memory_invariant?_TCC1	2416
C.123.8	changed_memory_invariant_mono	2416
C.123.9	memory_read_list_nse_ok	2418
C.123.10	memory_read_list_nse_next_length_TCC1	2422
C.123.11	memory_read_list_nse_next_length_TCC2	2422
C.123.12	memory_read_list_nse_next_length	2422
C.123.13	memory_read_list_ok	2428
C.123.14	memory_read_list_next_ok_TCC1	2430
C.123.15	memory_read_list_next_ok	2430
C.123.16	unchanged_memory_read_list_nse_TCC1	2432
C.123.17	unchanged_memory_read_list_nse_TCC2	2433
C.123.18	unchanged_memory_read_list_nse_TCC3	2434
C.123.19	unchanged_memory_read_list_nse	2435
C.123.20	unchanged_memory_read_list_TCC1	2450
C.123.21	unchanged_memory_read_list_TCC2	2451
C.123.22	unchanged_memory_read_list_TCC3	2451
C.123.23	unchanged_memory_read_list	2452
C.123.24	unchanged_memory_invariant_unchanged_read_list_nse_TCC1	2457
C.123.25	unchanged_memory_invariant_unchanged_read_list_nse_TCC2	2458
C.123.26	unchanged_memory_invariant_unchanged_read_list_nse_TCC3	2459
C.123.27	unchanged_memory_invariant_unchanged_read_list_nse	2459
C.123.28	memory_write_list_ok_nse	2466
C.123.29	memory_write_list_ok	2470
C.123.30	unchanged_memory_invariant_unchanged_write_list_nse_TCC1	2472
C.123.31	unchanged_memory_invariant_unchanged_write_list_nse_TCC2	2473
C.123.32	unchanged_memory_invariant_unchanged_write_list_nse_TCC3	2474
C.123.33	unchanged_memory_invariant_unchanged_write_list_nse	2475
C.123.34	unchanged_memory_invariant_unchanged_write_list_TCC1	2484
C.123.35	unchanged_memory_invariant_unchanged_write_list_TCC2	2485
C.123.36	unchanged_memory_invariant_unchanged_write_list_TCC3	2488
C.123.37	unchanged_memory_invariant_unchanged_write_list	2489
C.123.38	unchanged_memory_invariant_read_list	2493

C.123.39	unchanged_memory_read_list_write_list_nse_TCC1	2499
C.123.40	unchanged_memory_read_list_write_list_nse_TCC2	2500
C.123.41	unchanged_memory_read_list_write_list_nse	2502
C.123.42	unchanged_memory_read_list_write_list_TCC1	2538
C.123.43	unchanged_memory_read_list_write_list_TCC2	2538
C.123.44	unchanged_memory_read_list_write_list	2545
C.124	Proofs for Memory_Change_3 (memory.pvs)	2559
C.124.1	expr_unchanged_memory_invariant_unchanged_TCC1	2559
C.124.2	expr_unchanged_memory_invariant_unchanged	2559
C.124.3	expr_unchanged_memory_invariant_next_ok_TCC1	2560
C.124.4	expr_unchanged_memory_invariant_next_ok	2560
C.125	Proofs for Memory_Change_4 (memory.pvs)	2560
C.125.1	expr_unchanged_memory_invariant_composition	2560
C.125.2	fexpr_unchanged_memory_invariant_composition	2571
C.126	Proofs for Memory_Physical_Memory (physical_memory.pvs)	2579
C.127	Proofs for Memory_access (result.pvs)	2579
C.128	Proofs for Memory_access_adt (Memory_access_adt.pvs)	2579
C.129	Proofs for Memory_access_adt_reduce (Memory_access_adt.pvs)	2579
C.130	Proofs for Memory_access_util (result.pvs)	2579
C.131	Proofs for Memory_privilege (result.pvs)	2580
C.132	Proofs for Memory_privilege_adt (Memory_privilege_adt.pvs)	2580
C.133	Proofs for Memory_privilege_adt_reduce (Memory_privilege_adt.pvs)	2580
C.134	Proofs for Memory_privilege_util (result.pvs)	2580
C.134.1	bool_to_memory_privilege_iso	2580
C.135	Proofs for Min_Max_Extension (vfiasco-prelude.pvs)	2580
C.135.1	max_stable_under_extension_TCC1	2580
C.135.2	max_stable_under_extension	2580
C.135.3	min_stable_under_extension	2581
C.136	Proofs for More_Divides (vfiasco-prelude.pvs)	2582
C.136.1	expt_divides	2582
C.136.2	divides_expt_gt	2582
C.137	Proofs for More_Function (vfiasco-prelude.pvs)	2583
C.137.1	restrict_to_image_TCC1	2583
C.137.2	surjective_restrict_to_image	2583
C.137.3	bijective_restrict_to_image	2583
C.138	Proofs for More_List_Props (vfiasco-prelude.pvs)	2583
C.138.1	length_is_cons	2583
C.138.2	cons_length	2584
C.138.3	length_cdr_TCC1	2584
C.138.4	length_cdr	2584
C.138.5	conlist_induction	2584
C.138.6	head_TCC1	2586
C.138.7	head_TCC2	2586
C.138.8	head_TCC3	2586
C.138.9	head_tail	2586
C.138.10	length_head	2587
C.138.11	length_tail	2588
C.138.12	nth_head_TCC1	2589
C.138.13	nth_head_TCC2	2590
C.138.14	nth_head	2590
C.138.15	nth_tail_TCC1	2591
C.138.16	nth_tail_TCC2	2591

C.138.17	nth_tail	2591
C.138.18	car_head_TCC1	2593
C.138.19	car_head_TCC2	2593
C.138.20	car_head	2593
C.138.21	cons_tail	2593
C.138.22	car_tail_TCC1	2594
C.138.23	car_tail	2594
C.138.24	cdr_tail_TCC1	2595
C.138.25	cdr_tail	2595
C.138.26	every_implied	2597
C.138.27	every_conjunct_left	2598
C.138.28	every_iff_forall	2598
C.138.29	some_iff_exists	2599
C.138.30	list_extensionality_TCC1	2601
C.138.31	list_extensionality	2601
C.138.32	list_remove_TCC1	2604
C.138.33	list_remove_TCC2	2604
C.138.34	list_remove_commutative	2604
C.138.35	member_list_remove	2604
C.138.36	nth_member	2605
C.138.37	flatten_TCC1	2607
C.138.38	member_append	2607
C.138.39	member_flatten	2607
C.138.40	map2_TCC1	2609
C.138.41	map2_property	2609
C.139	Proofs for More_List_Props_2 (vfiasco-prelude.pvs)	2610
C.139.1	member_map	2610
C.140	Proofs for More_More_List_Props (vfiasco-prelude.pvs)	2611
C.140.1	list_all_length	2611
C.141	Proofs for More_Relations (vfiasco-prelude.pvs)	2612
C.141.1	zorn_lr	2612
C.142	Proofs for More_Sets_Lemmas (vfiasco-prelude.pvs)	2613
C.142.1	subset_equal	2613
C.142.2	union_subset3	2613
C.142.3	subset_bigger_union_left	2613
C.142.4	subset_bigger_union_right	2613
C.142.5	union_empty_left	2614
C.142.6	union_left	2614
C.142.7	union_right	2614
C.142.8	subset_of_differences	2614
C.142.9	difference_disjoint_3	2615
C.142.10	subset_singleton	2615
C.142.11	subset_member	2615
C.142.12	disjoint_symmetric	2615
C.142.13	disjoint_mono	2616
C.142.14	disjoint_subset_difference	2616
C.143	Proofs for Number_Props (vfiasco-prelude.pvs)	2616
C.143.1	number_split	2616
C.143.2	number_split_less	2616
C.143.3	bit_split_less	2617
C.143.4	rem_def_pos	2617
C.143.5	rem_rem	2618

C.143.6	divides_times	2619
C.143.7	ndiv_floor	2621
C.143.8	ndiv_self	2621
C.143.9	ndiv_plus_1	2622
C.143.10	ndiv_plus_2	2623
C.143.11	ndiv_plus_mod	2623
C.143.12	ndiv_plus_mod_2	2624
C.143.13	ndiv_minus_2	2625
C.143.14	ndiv_times_divident_1	2627
C.143.15	ndiv_times_divident_2	2629
C.143.16	rem_prod_3	2629
C.143.17	ndiv_times_2	2632
C.143.18	both_sides_ndiv_lt1	2632
C.143.19	rem_ndiv	2633
C.143.20	mod_rem	2634
C.143.21	ndiv_rem_divisible_TCC1	2635
C.143.22	ndiv_rem_divisible	2635
C.143.23	expt_2_8	2637
C.143.24	ndiv_0	2637
C.143.25	ndiv_1	2637
C.143.26	ndiv_bigger	2638
C.143.27	ndiv_reduce	2638
C.143.28	ndiv_expt_expt_TCC1	2639
C.143.29	ndiv_expt_expt	2639
C.143.30	min_plus1_TCC1	2641
C.143.31	min_plus1	2642
C.143.32	min_member	2643
C.144	Proofs for Ok_Result_Rewrite (state-transformer.pvs)	2644
C.144.1	ok_result_q_ok	2644
C.144.2	ok_result_q_state_TCC1	2644
C.144.3	ok_result_q_state_TCC2	2644
C.144.4	ok_result_q_state	2644
C.144.5	ok_result_q_data_TCC1	2645
C.144.6	ok_result_q_data	2645
C.144.7	ok_result_elimination_1	2645
C.145	Proofs for PDBR_Data_Model (paging-data-models.pvs)	2646
C.145.1	pdbr_valid?_TCC1	2646
C.145.2	pdbr_valid?_TCC2	2646
C.145.3	pdbr_to_byte_TCC1	2646
C.145.4	pdbr_to_byte_TCC2	2646
C.145.5	pdbr_to_byte_TCC3	2647
C.145.6	pdbr_from_byte_TCC1	2647
C.145.7	pdbr_from_byte_TCC2	2648
C.145.8	pdbr_from_byte_TCC3	2648
C.145.9	pdbr_from_byte_TCC4	2648
C.145.10	pdbr_from_byte_TCC5	2650
C.145.11	pdbr_model_TCC1	2650
C.145.12	pdbr_valid_to_byte	2650
C.145.13	pdbr_is_pod	2651
C.146	Proofs for PDBR_Datatype (paging-data.pvs)	2653
C.146.1	pdbr_data_type_TCC1	2653
C.147	Proofs for PDBR_Type (paging-data.pvs)	2654

C Proof scripts

C.147.1	Pdbr_type_TCC1	2654
C.148	Proofs for PDE_Datatype (paging-data.pvs)	2654
C.148.1	pde_data_type_TCC1	2654
C.149	Proofs for PDE_Model (paging-data-models.pvs)	2654
C.149.1	pde_valid?_TCC1	2654
C.149.2	pde_valid?_TCC2	2654
C.149.3	pde_valid?_TCC3	2655
C.149.4	pde_pte_to_byte_TCC1	2655
C.149.5	pde_pte_to_byte_TCC2	2655
C.149.6	pde_pte_to_byte_TCC3	2656
C.149.7	pde_super_to_byte_TCC1	2656
C.149.8	pde_super_to_byte_TCC2	2657
C.149.9	pde_super_to_byte_TCC3	2657
C.149.10	pde_to_byte_TCC1	2658
C.149.11	pde_to_byte_TCC2	2659
C.149.12	pde_4m_from_byte_TCC1	2659
C.149.13	pde_4m_from_byte_TCC2	2659
C.149.14	pde_4m_from_byte_TCC3	2659
C.149.15	pde_4m_from_byte_TCC4	2660
C.149.16	pde_4m_from_byte_TCC5	2660
C.149.17	pde_4m_from_byte_TCC6	2660
C.149.18	pde_4k_from_byte_TCC1	2661
C.149.19	pde_4k_from_byte_TCC2	2662
C.149.20	pde_4k_from_byte_TCC3	2662
C.149.21	pde_4k_from_byte_TCC4	2662
C.149.22	pde_4k_from_byte_TCC5	2663
C.149.23	pde_from_byte_TCC1	2665
C.149.24	pde_from_byte_TCC2	2665
C.149.25	pde_model_TCC1	2665
C.149.26	pde_valid_to_byte	2666
C.149.27	pde_is_pod	2666
C.150	Proofs for PF_EC_Model (paging-data-models.pvs)	2671
C.150.1	pf_ec_valid?_TCC1	2671
C.150.2	pf_ec_valid?_TCC2	2672
C.150.3	pf_ec_valid?_TCC3	2672
C.150.4	pf_ec_valid?_TCC4	2672
C.150.5	pf_ec_to_byte_TCC1	2672
C.150.6	pf_ec_to_byte_TCC2	2673
C.150.7	pf_ec_from_byte_TCC1	2673
C.150.8	pf_ec_from_byte_TCC2	2674
C.150.9	pf_ec_model_TCC1	2674
C.150.10	pf_ec_valid_to_byte	2674
C.150.11	pf_ec_is_pod	2674
C.151	Proofs for PF_Error_Code_Type (paging-data.pvs)	2676
C.151.1	pf_ec_data_type_TCC1	2676
C.152	Proofs for PTE_Datatype (paging-data.pvs)	2676
C.152.1	pte_data_type_TCC1	2676
C.153	Proofs for PTE_Model (paging-data-models.pvs)	2677
C.153.1	pte_page_to_byte_TCC1	2677
C.153.2	pte_page_to_byte_TCC2	2677
C.153.3	pte_page_to_byte_TCC3	2678
C.153.4	pte_to_byte_TCC1	2678

C.153.5	pte_to_byte_TCC2	2678
C.153.6	pte_page_from_byte_TCC1	2679
C.153.7	pte_page_from_byte_TCC2	2679
C.153.8	pte_page_from_byte_TCC3	2679
C.153.9	pte_page_from_byte_TCC4	2679
C.153.10	pte_page_from_byte_TCC5	2680
C.153.11	pte_page_from_byte_TCC6	2680
C.153.12	pte_from_byte_TCC1	2682
C.153.13	pte_model_TCC1	2682
C.153.14	pte_valid_to_byte	2682
C.153.15	pte_is_pod	2682
C.154	Proofs for Page_Directory_Types (paging-data.pvs)	2686
C.154.1	Pde_pt_type_TCC1	2686
C.155	Proofs for Paging_Datatype (paging-data.pvs)	2686
C.155.1	paging_data_type_TCC1	2686
C.155.2	paging_data_type_TCC2	2686
C.155.3	paging_data_type_TCC3	2687
C.155.4	paging_data_type_TCC4	2691
C.155.5	paging_data_type_TCC5	2691
C.155.6	paging_data_type_TCC6	2691
C.155.7	paging_data_type_TCC7	2693
C.156	Proofs for Paging_type (paging-data.pvs)	2693
C.157	Proofs for Paging_type_adt (Paging_type_adt.pvs)	2693
C.158	Proofs for Paging_type_adt_reduce (Paging_type_adt.pvs)	2693
C.159	Proofs for Paging_type_helpers (paging-data.pvs)	2694
C.159.1	range_pt_TCC1	2694
C.159.2	base_TCC1	2694
C.159.3	base_TCC2	2694
C.159.4	set_reference_range	2694
C.159.5	set_reference_pdir_entry	2694
C.159.6	set_reference_ptab_entry	2695
C.159.7	set_reference_paging_type	2695
C.159.8	set_reference_present	2696
C.159.9	set_reference_base_TCC1	2698
C.159.10	set_reference_base	2699
C.159.11	set_reference_accessible	2703
C.159.12	set_reference_privileged	2713
C.159.13	set_reference_leaf	2724
C.160	Proofs for Pde_type (paging-data.pvs)	2726
C.161	Proofs for Pde_type_adt (Pde_type_adt.pvs)	2726
C.161.1	Pde_4m_TCC1	2726
C.162	Proofs for Pde_type_adt_reduce (Pde_type_adt.pvs)	2726
C.163	Proofs for Phy_Mem_Blessing (challenge-phymem.pvs)	2727
C.163.1	phy_mem_plain_memory	2727
C.164	Proofs for Phy_Mem_Blessing_Data (challenge-phymem.pvs)	2731
C.164.1	in_memory_blessed_memory	2731
C.165	Proofs for Phy_Mem_Challenge_Read_Other (challenge-phymem.pvs)	2731
C.165.1	read_write_other_ok	2731
C.165.2	read_write_other_res_TCC1	2732
C.165.3	read_write_other_res_TCC2	2732
C.165.4	read_write_other_res	2733
C.165.5	read_read_ok	2733

C.165.6	read_read_TCC1	2734
C.165.7	read_read_TCC2	2734
C.165.8	read_read	2735
C.166	Proofs for Phy_Mem_Challenge_Read_Same (challenge-phymem.pvs)	2735
C.166.1	read_write_ok	2735
C.166.2	read_write_res_TCC1	2736
C.166.3	read_write_res	2736
C.167	Proofs for Phy_Mem_Properties (challenge-phymem.pvs)	2737
C.167.1	min_TCC1	2737
C.167.2	phy_mem_write_transformers_ok	2737
C.167.3	phy_mem_read_transformers_ok	2737
C.167.4	phy_mem_read_side_effects_transformers_ok	2737
C.167.5	phy_mem_write_side_effects_transformers_ok	2738
C.167.6	phy_mem_unchanged_memory_invariant_read_transformers	2738
C.167.7	phy_mem_unchanged_memory_invariant_read_side_effect_transformers	2739
C.167.8	phy_mem_unchanged_memory_invariant_write_side_effect_transformers	2740
C.167.9	phy_mem_unchanged_memory_write_invariant	2741
C.167.10	phy_mem_changed_memory_invariant	2742
C.168	Proofs for Physical_Memory (physical_memory.pvs)	2742
C.169	Proofs for Physical_Memory_Corollaries (physical_memory.pvs)	2743
C.169.1	physical_read_ok_next_state_TCC1	2743
C.169.2	physical_read_ok_next_state	2743
C.169.3	physical_write_ok_next_state_TCC1	2743
C.169.4	physical_write_ok_next_state	2743
C.169.5	physical_read_ok_get_data	2744
C.170	Proofs for Physical_Memory_Properties (physical_memory.pvs)	2744
C.170.1	physical_read_ok	2744
C.170.2	physical_write_ok	2744
C.171	Proofs for Plain_Mem_Properties (plain_memory.pvs)	2745
C.171.1	plain_mem_write_list_read_list_TCC1	2745
C.171.2	plain_mem_write_list_read_list	2745
C.171.3	plain_mem_q_read_list_TCC1	2746
C.171.4	plain_mem_q_read_list	2746
C.172	Proofs for Plain_Mem_Properties_2 (plain_memory.pvs)	2746
C.172.1	plain_memory_inv_pred_write_data_TCC1	2746
C.172.2	plain_memory_inv_pred_write_data	2747
C.172.3	plain_memory_read_data_ok	2747
C.172.4	plain_memory_write_data_ok	2749
C.172.5	plain_memory_write_data_valid_TCC1	2750
C.172.6	plain_memory_write_data_valid	2751
C.172.7	plain_memory_transformers_ok_write_data	2752
C.172.8	plain_memory_transformer_invariant_write_data	2753
C.172.9	plain_memory_transformer_invariant_read_data	2754
C.172.10	plain_memory_unchanged_memory_invariant_write_data	2755
C.172.11	plain_memory_unchanged_memory_invariant_read_data	2761
C.172.12	plain_memory_read_data_q_ok_TCC1	2764
C.172.13	plain_memory_read_data_q_ok	2764
C.172.14	plain_memory_read_data_q_same_TCC1	2768
C.172.15	plain_memory_read_data_q_same_TCC2	2768
C.172.16	plain_memory_read_data_q_same	2768
C.172.17	plain_memory_read_write_ok	2769
C.172.18	plain_memory_read_write_res_TCC1	2770

C.172.19	plain_memory_read_write_res	2771
C.172.20	plain_memory_unchanged_memory_ok_result	2773
C.173	Proofs for Plain_Mem_Properties_3 (plain_memory.pvs)	2775
C.173.1	plain_memory_read_write_other_ok	2775
C.173.2	plain_memory_read_write_other_res_TCC1	2776
C.173.3	plain_memory_read_write_other_res_TCC2	2777
C.173.4	plain_memory_read_write_other_res	2777
C.173.5	plain_memory_read_read_ok	2778
C.173.6	plain_memory_read_read_TCC1	2781
C.173.7	plain_memory_read_read_TCC2	2781
C.173.8	plain_memory_read_read	2782
C.173.9	plain_memory_unchanged_composition	2783
C.174	Proofs for Plain_Mem_Rewrites (plain_memory_rewrites.pvs)	2785
C.175	Proofs for Plain_Mem_Rewrites1 (plain_memory_rewrites.pvs)	2785
C.175.1	pm_q_prop_TCC1	2785
C.175.2	pm_q_prop_ok_stmt	2785
C.176	Proofs for Plain_Mem_Rewrites2 (plain_memory_rewrites.pvs)	2786
C.176.1	in_blessed_memory_rw_ro	2786
C.176.2	pm_q_prop_TCC1	2786
C.176.3	pm_q_prop_ok_expr	2786
C.176.4	pm_q_prop_single_read	2787
C.176.5	pm_q_prop_ok_result_single	2787
C.176.6	plain_memory_write_ok_single	2787
C.176.7	pm_q_prop_read_TCC1	2788
C.177	Proofs for Plain_Mem_Rewrites3 (plain_memory_rewrites.pvs)	2788
C.177.1	pm_q_prop_read_TCC1	2788
C.178	Proofs for Plain_Mem_Rewrites4 (plain_memory_rewrites.pvs)	2788
C.178.1	plain_memory_write_ok_q_expr	2788
C.178.2	pm_q_prop_read_ok_expr	2789
C.178.3	pm_q_prop_read_pm_q_prop_expr	2789
C.178.4	pm_q_prop_read_write_q_expr	2790
C.178.5	pm_q_prop_ok_result_q_expr	2790
C.178.6	pm_q_prop_read_ok_result_single	2790
C.178.7	plain_memory_read_write_q_data_expr_TCC1	2791
C.178.8	plain_memory_read_write_q_data_expr	2791
C.178.9	pm_q_prop_read_write_other_single	2792
C.178.10	pm_q_prop_read_read_single	2792
C.179	Proofs for Plain_Mem_Rewrites5 (plain_memory_rewrites.pvs)	2793
C.179.1	plain_memory_write_ok_q_stmt	2793
C.179.2	pm_q_prop_read_ok_stmt	2793
C.179.3	pm_q_prop_read_pm_q_prop_stmt	2794
C.179.4	pm_q_prop_read_write_q_stmt	2795
C.179.5	pm_q_prop_ok_result_q_stmt	2795
C.179.6	plain_memory_read_write_q_data_stmt_TCC1	2796
C.179.7	plain_memory_read_write_q_data_stmt	2796
C.179.8	pm_q_prop_single_write	2797
C.179.9	pm_q_prop_read_write_single	2797
C.179.10	pm_q_prop_e2s	2797
C.179.11	pm_q_prop_stmt_e2s	2798
C.180	Proofs for Plain_Mem_Rewrites6 (plain_memory_rewrites.pvs)	2798
C.180.1	pm_q_prop_read_write_other_q_expr	2798
C.180.2	pm_q_prop_read_read_q_expr	2799

C.180.3	pm_q_prop_read_ok_result_q_expr	2801
C.180.4	plain_memory_read_read_q_data_expr_TCC1	2801
C.180.5	plain_memory_read_read_q_data_expr_TCC2	2801
C.180.6	plain_memory_read_read_q_data_expr	2802
C.181	Proofs for Plain_Mem_Rewrites7 (plain_memory_rewrites.pvs)	2802
C.181.1	pm_q_prop_read_write_other_q_stmt	2802
C.181.2	plain_memory_read_write_other_single_data_TCC1	2804
C.181.3	plain_memory_read_write_other_single_data_TCC2	2805
C.181.4	plain_memory_read_write_other_single_data	2806
C.181.5	pm_q_prop_read_read_q_stmt	2807
C.181.6	pm_q_prop_read_ok_result_q_stmt	2807
C.181.7	plain_memory_read_write_other_q_data_stmt_TCC1	2808
C.181.8	plain_memory_read_write_other_q_data_stmt_TCC2	2808
C.181.9	plain_memory_read_write_other_q_data_stmt	2809
C.181.10	plain_memory_read_read_q_data_stmt_TCC1	2809
C.181.11	plain_memory_read_read_q_data_stmt_TCC2	2810
C.181.12	plain_memory_read_read_q_data_stmt	2810
C.181.13	pm_q_prop_read_e2s	2811
C.181.14	pm_q_prop_read_stmt_e2s	2811
C.182	Proofs for Plain_Mem_Rewrites8 (plain_memory_rewrites.pvs)	2812
C.182.1	plain_memory_read_write_other_q_data_expr_TCC1	2812
C.182.2	plain_memory_read_write_other_q_data_expr_TCC2	2812
C.182.3	plain_memory_read_write_other_q_data_expr	2813
C.183	Proofs for Plain_Memory (plain_memory.pvs)	2814
C.183.1	plain_memory_invariant_read_transformers_ro_addr	2814
C.183.2	plain_memory_invariant_read_transformers_rw_addr	2814
C.183.3	plain_memory_invariant_read_transformers_ro_rw_addr	2814
C.183.4	plain_memory_invariant_write_transformers_rw_addr	2815
C.183.5	plain_memory_invariant	2815
C.183.6	plain_memory_unchanged_invariant	2816
C.183.7	plain_memory_unchanged_memory_invariant_write	2817
C.183.8	plain_memory_unchanged_memory_write_invariant	2817
C.183.9	plain_memory_changed_memory_invariant	2817
C.183.10	plain_memory_invariant_read_block	2818
C.183.11	plain_memory_invariant_write_block	2818
C.183.12	plain_memory_transformer_invariant_read_side_effects	2818
C.183.13	plain_memory_transformer_invariant_write_side_effects	2820
C.183.14	plain_memory_transformer_invariant_read_list_block	2821
C.183.15	plain_memory_transformer_invariant_write_list_block	2821
C.183.16	plain_memory_transformers_ok_read_ro_rw	2821
C.183.17	plain_memory_transformers_ok_read_block	2822
C.183.18	plain_memory_transformers_ok_write_rw	2823
C.183.19	plain_memory_transformers_ok_write_block	2823
C.183.20	plain_memory_transformers_ok_read_side_effects_ro_rw	2824
C.183.21	plain_memory_transformers_ok_read_side_effects_block	2824
C.183.22	plain_memory_transformers_ok_write_side_effects_ro_rw	2825
C.183.23	plain_memory_transformers_ok_write_side_effects_block	2825
C.183.24	plain_memory_transformers_ok_read_list_block	2826
C.183.25	plain_memory_transformers_ok_write_list_block	2826
C.183.26	plain_memory_unchanged_read_block_rw_addr	2826
C.183.27	plain_memory_unchanged_read_ro_rw_addr	2827
C.183.28	plain_memory_unchanged_read_block_ro_rw_addr	2827

C.183.29	plain_memory_unchanged_write_block_ro_addr	2828
C.183.30	plain_memory_unchanged_read_block	2828
C.183.31	plain_memory_unchanged_read_side_effect_block_ro_rw	2829
C.183.32	plain_memory_unchanged_read_side_effect_block	2829
C.183.33	plain_memory_unchanged_write_side_effect_block_ro_rw	2830
C.183.34	plain_memory_unchanged_write_side_effect_block	2830
C.183.35	plain_memory_unchanged_block	2831
C.183.36	plain_memory_changed_block	2831
C.183.37	plain_memory_side_effect_content_unchanged_read_block	2832
C.183.38	plain_memory_side_effect_content_unchanged_write_block	2832
C.183.39	plain_memory_memory_read_ok	2832
C.183.40	plain_memory_states_memory_read_TCC1	2833
C.183.41	plain_memory_states_memory_read	2833
C.183.42	plain_memory_memory_read_memory_read_TCC1	2833
C.183.43	plain_memory_memory_read_memory_read_TCC2	2834
C.183.44	plain_memory_memory_read_memory_read_TCC3	2834
C.183.45	plain_memory_memory_read_memory_read	2835
C.183.46	plain_memory_memory_write_ok	2835
C.183.47	plain_memory_states_memory_write_TCC1	2835
C.183.48	plain_memory_states_memory_write	2836
C.183.49	plain_memory_memory_read_memory_write_other_TCC1	2836
C.183.50	plain_memory_memory_read_memory_write_other_TCC2	2837
C.183.51	plain_memory_memory_read_memory_write_other_TCC3	2837
C.183.52	plain_memory_memory_read_memory_write_other	2837
C.183.53	stays_unchanged_TCC1	2839
C.183.54	stays_unchanged_TCC2	2839
C.183.55	stays_unchanged_unchanged_TCC1	2839
C.183.56	stays_unchanged_unchanged_TCC2	2840
C.183.57	stays_unchanged_unchanged	2840
C.183.58	stays_unchanged_symmetric	2840
C.183.59	stays_unchanged_mono	2841
C.183.60	stays_unchanged_only_changes	2841
C.183.61	stays_unchanged_only_changes_disjoint	2842
C.183.62	only_changes_symmetric	2843
C.183.63	only_changes_mono	2843
C.183.64	only_changes_unchanged_TCC1	2844
C.183.65	only_changes_unchanged_TCC2	2844
C.183.66	only_changes_unchanged	2844
C.183.67	plain_memory_stays_unchanged_memory_read_TCC1	2845
C.183.68	plain_memory_stays_unchanged_memory_read	2845
C.183.69	plain_memory_stays_unchanged_memory_write_TCC1	2846
C.183.70	plain_memory_stays_unchanged_memory_write	2846
C.184	Proofs for PointerToMemberExpressions (expressions.pvs)	2846
C.184.1	ptm_member_TCC1	2846
C.184.2	ptm_arrow_TCC1	2847
C.185	Proofs for Posnat_induction (vfiasco-prelude.pvs)	2855
C.185.1	posnat_induction	2855
C.186	Proofs for PostfixExpressions (expressions.pvs)	2856
C.186.1	postinc_TCC1	2856
C.186.2	postinc_float_TCC1	2857
C.186.3	postinc_float_TCC2	2857
C.186.4	postinc_float_TCC3	2858

C Proof scripts

C.187 Proofs for PrimaryExpressions (expressions.pvs)	2858
C.188 Proofs for Pte_type (paging-data.pvs)	2858
C.189 Proofs for Pte_type_adt (Pte_type_adt.pvs)	2858
C.189.1 Pte_TCC1	2858
C.190 Proofs for Pte_type_adt_reduce (Pte_type_adt.pvs)	2858
C.191 Proofs for Random_Device (random_device.pvs)	2858
C.191.1 base_TCC1	2858
C.191.2 rnd_seed_TCC1	2859
C.191.3 mem_cnt_TCC1	2859
C.191.4 rnd_val_TCC1	2860
C.191.5 increase_access_count_TCC1	2860
C.191.6 clear_counter_TCC1	2860
C.191.7 random_TCC1	2861
C.191.8 read_mem_cnt_TCC1	2861
C.191.9 read_mem_cnt_TCC2	2861
C.191.10 read_mem_cnt_TCC3	2862
C.191.11 read_seed_TCC1	2864
C.191.12 read_seed_TCC2	2864
C.191.13 read_seed_TCC3	2865
C.192 Proofs for Read_After_Write_Fails (datatype_model.pvs)	2869
C.192.1 plain_mem_write_list_states_TCC1	2869
C.192.2 plain_mem_write_list_states	2869
C.192.3 plain_mem_write_list_read_list_ok	2871
C.192.4 Fatal_Result	2872
C.193 Proofs for Read_After_Write_Fails_2 (datatype_model.pvs)	2873
C.193.1 plain_mem_write_list_states_TCC1	2873
C.193.2 plain_mem_write_list_states	2873
C.193.3 plain_mem_write_list_read_list_ok	2875
C.193.4 Fatal_Result	2876
C.194 Proofs for Register_Id (constants.pvs)	2877
C.195 Proofs for Register_Id_adt (Register_Id_adt.pvs)	2877
C.196 Proofs for Register_Id_adt_reduce (Register_Id_adt.pvs)	2878
C.197 Proofs for Rewrite_Test (plain_memory_rewrites.pvs)	2878
C.197.1 rewrite_test	2878
C.197.2 rewrite_test_data_TCC1	2878
C.197.3 rewrite_test_data	2879
C.197.4 rewrite_test_data_long_TCC1	2880
C.197.5 rewrite_test_data_long	2882
C.197.6 rewrite_test_data_eval_if_ok_ok	2884
C.197.7 rewrite_test_data_eval_if_ok_TCC1	2885
C.197.8 rewrite_test_data_eval_if_ok	2885
C.197.9 rewrite_test_data_eval_if_ok_ok_result_ok	2886
C.197.10 rewrite_test_data_eval_if_ok_ok_result_TCC1	2887
C.197.11 rewrite_test_data_eval_if_ok_ok_result	2887
C.198 Proofs for Search_Example (search-example.pvs)	2888
C.198.1 value_TCC1	2888
C.198.2 Search_Pre_TCC1	2889
C.198.3 Search_Pre_TCC2	2889
C.198.4 Search_Pre_TCC3	2892
C.198.5 Search_Pre_TCC4	2893
C.198.6 Search_Pre_TCC5	2893
C.198.7 Search_Pre_TCC6	2894

C.198.8	Search_Pre_TCC7	2895
C.198.9	Search_Pre_TCC8	2895
C.198.10	Search_Pre_TCC9	2896
C.198.11	search_TCC1	2898
C.198.12	search_TCC2	2900
C.198.13	search_TCC3	2901
C.198.14	search_TCC4	2901
C.198.15	deref_current_TCC1	2903
C.198.16	deref_current_TCC2	2904
C.198.17	read_array_TCC1	2904
C.198.18	read_array_TCC2	2904
C.198.19	read_array_TCC3	2905
C.198.20	not_found_TCC1	2905
C.198.21	not_found_TCC2	2905
C.198.22	not_found_TCC3	2906
C.198.23	not_found_TCC4	2906
C.198.24	found_TCC1	2907
C.198.25	found_TCC2	2907
C.198.26	found_TCC3	2908
C.198.27	uchar_range_values_TCC1	2909
C.198.28	uchar_range_values	2910
C.198.29	search_terminates	2910
C.198.30	search_read_first	2912
C.198.31	search_read_current	2914
C.198.32	search_deref_current	2915
C.198.33	search_read_last	2917
C.198.34	search_read_array	2919
C.198.35	search_not_found	2921
C.198.36	search_found	2922
C.199	Proofs for Segment_Reg_Datatype (paging-data.pvs)	2924
C.199.1	segment_selector_data_type_TCC1	2924
C.199.2	segment_reg_data_type_TCC1	2924
C.200	Proofs for Segment_Types (paging-data.pvs)	2925
C.201	Proofs for SelectionStatements (statements.pvs)	2925
C.202	Proofs for Set_Lift (vfiasco-prelude.pvs)	2925
C.203	Proofs for Single_Statement_Rewrites0 (statement-rewrites.pvs)	2925
C.203.1	break_catch_break	2925
C.203.2	break_break_stmt	2925
C.203.3	break_break_catch_continue	2926
C.203.4	break_break_catch_default	2926
C.203.5	break_break_lift_stmt	2926
C.203.6	break_break_case	2926
C.203.7	break_break_default	2927
C.203.8	break_break	2927
C.203.9	stmt_break_break	2927
C.203.10	break_stmt_break	2927
C.203.11	break_stmt_catch_continue	2927
C.203.12	break_stmt_catch_default	2928
C.203.13	break_stmt_lift	2928
C.203.14	break_stmt_case	2928
C.203.15	break_stmt_default	2928
C.203.16	continue_catch_continue	2929

C.203.17	continue_break_stmt	2929
C.203.18	continue_break_catch_continue	2929
C.203.19	continue_break_catch_default	2929
C.203.20	continue_break_lift_stmt	2930
C.203.21	continue_break_case	2930
C.203.22	continue_break_default	2930
C.203.23	continue_continue	2930
C.203.24	stmt_continue_continue	2930
C.203.25	continue_stmt_continue	2931
C.203.26	continue_stmt_catch_break	2931
C.203.27	continue_stmt_catch_default	2931
C.203.28	continue_stmt_lift	2931
C.203.29	continue_stmt_case	2932
C.203.30	continue_stmt_default	2932
C.203.31	switch_stmt	2932
C.203.32	stmt_switch_stmt	2932
C.203.33	switch_case_taken	2933
C.203.34	stmt_switch_case_taken	2933
C.203.35	switch_case_not_taken	2933
C.203.36	stmt_switch_case_not_taken	2933
C.203.37	switch_default	2934
C.203.38	stmt_switch_default	2934
C.203.39	switch_catch_break	2934
C.203.40	stmt_switch_catch_break	2935
C.203.41	switch_catch_continue	2935
C.203.42	stmt_switch_catch_continue	2935
C.203.43	switch_catch_default	2935
C.203.44	stmt_switch_catch_default	2936
C.203.45	default_stmt	2936
C.203.46	stmt_default_stmt	2936
C.203.47	default_case	2936
C.203.48	stmt_default_case	2937
C.203.49	default_default	2937
C.203.50	stmt_default_default	2937
C.203.51	default_catch_break	2937
C.203.52	stmt_default_catch_break	2938
C.203.53	default_catch_continue	2938
C.203.54	stmt_default_catch_continue	2938
C.203.55	default_catch_default	2938
C.203.56	stmt_default_catch_default	2939
C.203.57	return_void_result	2939
C.203.58	skip_ok	2939
C.203.59	skip_state_TCC1	2939
C.203.60	skip_state	2940
C.203.61	skip_elimination	2940
C.203.62	stmt_skip_elimination	2940
C.203.63	stmt_remove_catch_default	2940
C.203.64	stmt_remove_case	2941
C.203.65	stmt_remove_default	2941
C.203.66	stmt_remove_catch_continue	2941
C.203.67	stmt_remove_catch_break	2941
C.203.68	stmt_ok_lift	2941

C.203.69	while_unroll_TCC1	2942
C.203.70	while_unroll	2942
C.203.71	stmt_while_unroll	2942
C.203.72	iterate_while_ok	2943
C.203.73	while_hang	2945
C.203.74	composition_assoc_rewrite_stmt_ok	2946
C.203.75	composition_assoc_rewrite_stmt_ok_cstmt	2947
C.204	Proofs for Single_Statement_Rewrites1 (statement-rewrites.pvs)	2947
C.204.1	forward_s_c_s_of_s_expr	2947
C.204.2	stmt_ok_lift_expr	2947
C.204.3	break_break_stmt_expr	2947
C.204.4	break_break_catch_continue_expr	2948
C.204.5	break_break_catch_default_expr	2948
C.204.6	break_break_lift_stmt_expr	2948
C.204.7	break_break_case_expr	2949
C.204.8	break_break_default_expr	2949
C.204.9	continue_break_stmt_expr	2949
C.204.10	continue_break_catch_continue_expr	2949
C.204.11	continue_break_catch_default_expr	2950
C.204.12	continue_break_lift_stmt_expr	2950
C.204.13	continue_break_case_expr	2950
C.204.14	continue_break_default_expr	2951
C.204.15	while_unroll_expr_TCC1	2951
C.204.16	while_unroll_expr	2951
C.204.17	while_unroll_lexpr	2952
C.204.18	stmt_while_unroll_expr	2952
C.204.19	stmt_while_unroll_lexpr	2953
C.204.20	e2s_ok	2953
C.204.21	stmt_e2s_ok	2954
C.204.22	e2s_state_TCC1	2954
C.204.23	e2s_state_TCC2	2954
C.204.24	e2s_state	2955
C.204.25	stmt_e2s_state_TCC1	2955
C.204.26	stmt_e2s_state_TCC2	2955
C.204.27	stmt_e2s_state	2956
C.204.28	comp_eval_if_ok_fstmt	2956
C.204.29	stmt_eval_if_ok_fstmt	2956
C.204.30	comp_eval_if_ok_fstmt_cstmt	2956
C.204.31	stmt_eval_if_ok_fstmt_cstmt	2957
C.204.32	composition_assoc_rewrite_expr_1	2957
C.204.33	composition_assoc_rewrite_expr_2	2957
C.204.34	composition_assoc_rewrite_expr_3	2958
C.204.35	composition_assoc_rewrite_expr_4	2958
C.204.36	composition_assoc_rewrite_expr_5	2958
C.204.37	composition_assoc_rewrite_expr_6	2959
C.204.38	composition_assoc_rewrite_stmt_ok	2959
C.204.39	composition_assoc_rewrite_stmt_ok_expr	2959
C.204.40	composition_assoc_rewrite_stmt_ok_expr_lexpr	2960
C.204.41	ok_result_fstmt	2960
C.204.42	pm_q_prop_read_ok_s_read_ok	2960
C.205	Proofs for Single_Statement_Rewrites2 (statement-rewrites.pvs)	2961
C.205.1	ok_result_fexpr	2961

C.205.2	e2s_merge	2961
C.205.3	stmt_e2s_merge	2961
C.205.4	e2s_expr	2962
C.205.5	stmt_e2s_expr	2962
C.205.6	composition_assoc_expression_rewrite_stmt	2962
C.205.7	comp_eval_if_ok_fstmt_expr	2962
C.205.8	stmt_eval_if_ok_fstmt_expr	2963
C.205.9	stmt_eval_if_ok_fexpr	2963
C.205.10	comp_eval_if_ok_fstmt_cstmt_expr	2963
C.205.11	stmt_eval_if_ok_fstmt_cstmt_expr	2964
C.206	Proofs for Single_Statement_Rewrites3 (statement-rewrites.pvs)	2964
C.206.1	ok_result_elimination_2	2964
C.206.2	comp_eval_if_ok_fexpr_expr	2965
C.206.3	expr_eval_if_ok_fexpr	2965
C.206.4	composition_assoc_expression_rewrite	2965
C.206.5	stmt_eval_if_ok_fexpr_expr	2966
C.207	Proofs for Single_Statement_Rewrites4 (statement-rewrites.pvs)	2966
C.207.1	expr_eval_if_ok_fexpr_expr	2966
C.208	Proofs for SkipStatements (statements.pvs)	2966
C.209	Proofs for State_Transformer (state-transformer.pvs)	2966
C.210	Proofs for State_Transformer_Lift (state-transformer.pvs)	2967
C.211	Proofs for State_Transformer_Lift_Expr (state-transformer.pvs)	2967
C.211.1	ok_result_ok	2967
C.211.2	ok_result_data_TCC1	2967
C.211.3	ok_result_data	2967
C.211.4	ok_result_state	2967
C.211.5	expr_2_super_ok_result	2968
C.211.6	ok_lift_ok	2968
C.211.7	has_next_state_ok_result	2968
C.211.8	has_next_state_fatal_result	2968
C.211.9	expr_2_super_fatal_result	2968
C.212	Proofs for State_Transformer_Lift_Stmt (state-transformer.pvs)	2969
C.213	Proofs for Statement_Composition_Rewrites (state-transformer.pvs)	2969
C.213.1	composition_assoc_rewrite_1	2969
C.213.2	composition_assoc_rewrite_2	2969
C.213.3	composition_assoc_rewrite_3	2969
C.213.4	composition_assoc_rewrite_4	2970
C.213.5	composition_assoc_rewrite_5	2970
C.213.6	composition_assoc_rewrite_6	2970
C.213.7	composition_assoc_rewrite_7	2971
C.213.8	composition_assoc_rewrite_8	2971
C.214	Proofs for Statement_Rewrites (statement-rewrites.pvs)	2971
C.215	Proofs for Statements (statements.pvs)	2971
C.216	Proofs for StmtResult (result.pvs)	2972
C.217	Proofs for StmtResult_adt (StmtResult_adt.pvs)	2972
C.218	Proofs for StmtResult_adt_map (StmtResult_adt.pvs)	2972
C.219	Proofs for StmtResult_adt_reduce (StmtResult_adt.pvs)	2972
C.220	Proofs for Subtype_stable (graph.pvs)	2972
C.220.1	length_stable	2972
C.220.2	nth_stable_TCC1	2972
C.220.3	nth_stable	2973
C.221	Proofs for SuperResult (result.pvs)	2974

C.222	Proofs for SuperResult_adt (SuperResult_adt.pvs)	2974
C.223	Proofs for SuperResult_adt_map (SuperResult_adt.pvs)	2974
C.224	Proofs for SuperResult_adt_reduce (SuperResult_adt.pvs)	2974
C.225	Proofs for Super_Embedding (result.pvs)	2974
C.226	Proofs for Super_Embedding_Expr (result.pvs)	2974
C.226.1	expr_2_super_res_TCC1	2974
C.226.2	expr_2_super_res_TCC2	2974
C.227	Proofs for Super_Embedding_Stmt (result.pvs)	2975
C.227.1	stmt_2_super_res_TCC1	2975
C.227.2	stmt_2_super_res_TCC2	2975
C.228	Proofs for Super_Result_Util (result.pvs)	2975
C.229	Proofs for Transformer_Invariant (state-transformer.pvs)	2976
C.229.1	result_pred_TCC1	2976
C.229.2	super_transformer_invariant_next_ok_TCC1	2976
C.229.3	super_transformer_invariant_next_ok	2976
C.229.4	transformer_invariant_mono_transformers	2977
C.229.5	transformer_invariant_union_transformers	2977
C.229.6	transformer_invariant_truth	2978
C.229.7	transformer_invariant_all_transformers	2978
C.229.8	super_transformers_ok_ok	2978
C.229.9	transformers_ok_mono_transformers	2979
C.229.10	transformers_ok_mono_states	2979
C.229.11	transformers_ok_union_transformers	2979
C.229.12	transformers_ok_all_transformers	2980
C.230	Proofs for Transformer_Invariant_2 (state-transformer.pvs)	2981
C.230.1	expr_transformer_invariant_next_ok_TCC1	2981
C.230.2	expr_transformer_invariant_next_ok	2981
C.230.3	expr_transformers_ok_ok	2981
C.230.4	expr_result_pred_next_state_TCC1	2982
C.230.5	expr_result_pred_next_state	2982
C.231	Proofs for Transformer_Invariant_3 (state-transformer.pvs)	2982
C.231.1	fexpr_composition_transformers_ok	2982
C.231.2	expr_composition_transformers_ok	2983
C.231.3	fexpr_composition_transformer_invariant	2984
C.231.4	expr_composition_transformer_invariant	2985
C.232	Proofs for Transformer_Super_Embedding (state-transformer.pvs)	2985
C.233	Proofs for Transformer_Super_Embedding_Expr (state-transformer.pvs)	2985
C.233.1	ok_expr_2_super	2985
C.233.2	has_next_state_expr_2_super	2986
C.233.3	state_expr_2_super_TCC1	2986
C.233.4	state_expr_2_super_TCC2	2986
C.233.5	state_expr_2_super	2987
C.234	Proofs for Transformer_Super_Embedding_Stmt (state-transformer.pvs)	2987
C.234.1	ok_stmt_2_super	2987
C.234.2	has_next_state_stmt_2_super	2987
C.234.3	state_stmt_2_super_TCC1	2987
C.234.4	state_stmt_2_super_TCC2	2988
C.234.5	state_stmt_2_super	2988
C.235	Proofs for UnaryExpressions (expressions.pvs)	2988
C.235.1	deref_TCC1	2988
C.235.2	arrow_TCC1	2989
C.235.3	arrow_TCC2	2989

C Proof scripts

C.235.4	unary_minus_TCC1	2989
C.235.5	unary_minus_TCC2	2990
C.235.6	unary_minus_unsigned_TCC1	2990
C.235.7	unary_minus_unsigned_TCC2	2990
C.235.8	preinc_TCC1	2990
C.235.9	sizeof_TCC1	2991
C.236	Proofs for Uninterpreted_Data (abstract_data.pvs)	2991
C.236.1	u_data_type_length	2991
C.237	Proofs for Unit (viasco-prelude.pvs)	2991
C.238	Proofs for Unit_adt (Unit_adt.pvs)	2992
C.239	Proofs for Unit_adt_reduce (Unit_adt.pvs)	2992
C.240	Proofs for While_Hoare (statement-rewrites.pvs)	2992
C.240.1	iterate_while_right	2992
C.240.2	while_variant?_TCC1	2993
C.240.3	while_no_cb_TCC1	2995
C.240.4	while_no_cb_cb_while	2995
C.240.5	iterate_while_invariant	2995
C.240.6	min_termination_point_no_less	2997
C.240.7	while_terminates	2997
C.240.8	hoare_while	2997
C.241	Proofs for While_Rewrites (statement-rewrites.pvs)	2998
C.241.1	while_to_while_no_cb	2998
C.241.2	while_inv_rewrite_termination_result	2998
C.241.3	while_inv_rewrite_data_ok	2998
C.241.4	while_inv_rewrite_data_break	2999
C.241.5	while_inv_rewrite_data_return	2999
C.241.6	stmt_while_inv_rewrite_termination_result_TCC1	3000
C.241.7	stmt_while_inv_rewrite_termination_result	3000
C.241.8	stmt_while_inv_rewrite_data_ok	3001
C.241.9	stmt_while_inv_rewrite_data_break	3002
C.241.10	stmt_while_inv_rewrite_data_return	3002
C.241.11	pm_q_prop_while	3003
C.241.12	pm_q_prop_stmt_while_TCC1	3004
C.241.13	pm_q_prop_stmt_while	3004

C.1 Proofs for Abstract_Read_Write (abstract_data.pvs)

C.1.1 Abstract_Read_Write.read_data_valid_in_mem

Terse proof for read_data_valid_in_mem.

read_data_valid_in_mem:

$$\frac{}{\{1\} \quad \forall (\text{addr}: \text{Address}, \text{dt}: (\text{interpreted_data_type?}[\text{Data}]), \text{pm}: \text{Memory_struct}[\text{State}], \\ s: \text{State}): \\ \text{OK?}(\text{read_data}(\text{pm}, \text{dt})(\text{addr})(s)) \supset \text{valid_in_mem}(\text{pm}, \text{dt})(\text{addr})(s)}$$

Repeatedly Skolemizing and flattening,
 Expanding the definition of valid_in_mem,
 Expanding the definition of read_data,
 Expanding the definition of ok_lift,
 Expanding the definition of ##,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 Rewriting using valid_iff_from_byte, matching in * where idt gets dt!1,
 This completes the proof of read_data_valid_in_mem.
 Q.E.D.

C.2 Proofs for Abstract_Read_Write_Plain (plain_memory.pvs)

This theory contains no provable formal statements.

C.3 Proofs for Address_Datatype (paging-data.pvs)

C.3.1 Address_Datatype.address_data_type_TCC1

Terse proof for address_data_type_TCC1.

address_data_type_TCC1:

$$\frac{}{\{1\} \quad \exists (x_1: (\text{pod_data_type?}[\text{Memory_Address_4G}])): \text{TRUE}}$$

Using lemma address_data_type_exists,
 This completes the proof of address_data_type_TCC1.
 Q.E.D.

C.4 Proofs for Address_Helpers (constants.pvs)

C.4.1 Address_Helpers.type_add_rewrite

Terse proof for type_add_rewrite.

type_add_rewrite:

$$\frac{}{\{1\} \quad \forall (\text{m_addr}: \text{Memory_Address}, \text{size}: \text{int}): (\text{m_addr} + \text{size})\text{'type_of} = \text{m_addr}\text{'type_of}}$$

Trying repeated skolemization, instantiation, and if-lifting,

C Proof scripts

This completes the proof of `type_add_rewrite`.
Q.E.D.

C.4.2 Address_Helpers.offset_add_rewrite

Terse proof for `offset_add_rewrite`.

`offset_add_rewrite`:

$$\{1\} \quad \forall (m_addr: \text{Memory_Address}, \text{size}: \text{int}):$$
$$(m_addr + \text{size})'offset = m_addr'offset + \text{size}$$

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `offset_add_rewrite`.
Q.E.D.

C.4.3 Address_Helpers.type_add_memory

Terse proof for `type_add_memory`.

`type_add_memory`:

$$\{1\} \quad \forall (m_addr, \text{size}): \text{Mem}((m_addr + \text{size})'type_of)$$

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `type_add_memory`.
Q.E.D.

C.4.4 Address_Helpers.address_add_zero

Terse proof for `address_add_zero`.

`address_add_zero`:

$$\{1\} \quad \forall (a: \text{Address}): a + 0 = a$$

Expanding the definition of `+`,
which is trivially true.
This completes the proof of `address_add_zero`.
Q.E.D.

C.4.5 Address_Helpers.address_add_sum

Terse proof for `address_add_sum`.

`address_add_sum`:

$$\{1\} \quad \forall (a: \text{Address}, i, j: \text{int}): (a + i) + j = a + (i + j)$$

Expanding the definition of `+`,
which is trivially true.
This completes the proof of `address_add_sum`.
Q.E.D.

C.4.6 Address_Helpers.reg_size_TCC1

Terse proof for reg_size_TCC1.

reg_size_TCC1:

```
{1}  ∃ (max: Memory_Address, id: Register_Id):
      ¬ SYSENTER_EIP?(id) ∧
      ¬ SYSENTER_ESP?(id) ∧
      ¬ SYSENTER_CS?(id) ∧
      ¬ TR?(id) ∧
      ¬ LDTR?(id) ∧
      ¬ GDTR?(id) ∧ ¬ IDTR?(id) ∧ ¬ EFLAGS?(id) ∧ ¬ IP?(id) ∧ ¬ Mem?(id)
      ⊃ ¬ (GP?(id) ∧ Segment_Reg?(id)) ∧ ¬ (Segment_Reg?(id) ∧ Control_Reg?(id))
```

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of reg_size_TCC1.

Q.E.D.

C.4.7 Address_Helpers.reg_size_TCC2

Terse proof for reg_size_TCC2.

reg_size_TCC2:

```
{1}  ∃ (max: Memory_Address, id: Register_Id):
      ¬ SYSENTER_EIP?(id) ∧
      ¬ SYSENTER_ESP?(id) ∧
      ¬ SYSENTER_CS?(id) ∧
      ¬ TR?(id) ∧
      ¬ LDTR?(id) ∧
      ¬ GDTR?(id) ∧ ¬ IDTR?(id) ∧ ¬ EFLAGS?(id) ∧ ¬ IP?(id) ∧ ¬ Mem?(id)
      ⊃ GP?(id) ∨ Segment_Reg?(id) ∨ Control_Reg?(id)
```

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of reg_size_TCC2.

Q.E.D.

C.5 Proofs for Address_Model (paging-data-models.pvs)

C.5.1 Address_Model.address_to_byte_TCC1

Terse proof for address_to_byte_TCC1.

address_to_byte_TCC1:

```
{1}  ∃ (addr: Memory_Address_4G): offset(addr) ≥ 0
```

Repeatedly Skolemizing and flattening,

Installing automatic rewrites from: every less_than_max_byte overwrite_bool_bit_less overwrite_shift_bits_less max_linear_offset bus_width

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of address_to_byte_TCC1.

Q.E.D.

C.5.2 Address_Model.address_to_byte_TCC2

Terse proof for address_to_byte_TCC2.

address_to_byte_TCC2:

$$\{1\} \quad \forall (\text{addr}: \text{Memory_Address_4G}): \text{cut_bits}(\text{offset}(\text{addr}), 0, 8) < \text{max_byte}$$

Repeatedly Skolemizing and flattening,
 Using lemma cut_bits_below,
 Rewriting using less_than_max_byte, matching in *,
 This completes the proof of address_to_byte_TCC2.
 Q.E.D.

C.5.3 Address_Model.address_to_byte_TCC3

Terse proof for address_to_byte_TCC3.

address_to_byte_TCC3:

$$\{1\} \quad \forall (\text{addr}: \text{Memory_Address_4G}):$$

$$\text{every}[\text{nat}]$$

$$(\{s: \text{nat} \mid s < \text{max_byte}\})$$

$$((:\text{cut_bits}(\text{offset}(\text{addr}), 8, 8), \text{cut_bits}(\text{offset}(\text{addr}), 16, 8),$$

$$\text{cut_bits}(\text{offset}(\text{addr}), 24, 8):))$$

Repeatedly Skolemizing and flattening,
 Installing automatic rewrites from: every less_than_max_byte overwrite_bool_bit_less over-
 write_shift_bits_less max_linear_offset bus_width
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 This completes the proof of address_to_byte_TCC3.
 Q.E.D.

C.5.4 Address_Model.address_from_byte_TCC1

Terse proof for address_from_byte_TCC1.

address_from_byte_TCC1:

$$\{1\} \quad \forall (\text{data}: \text{list}[\text{Byte}], a: \text{Address}): \text{address_valid?}(\text{data}, a) \supset 0 < \text{length}[\text{Byte}](\text{data})$$

Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of address_from_byte_TCC1.
 Q.E.D.

C.5.5 Address_Model.address_from_byte_TCC2

Terse proof for address_from_byte_TCC2.

address_from_byte_TCC2:

$$\{1\} \quad \forall (\text{data}: \text{list}[\text{Byte}], a: \text{Address}): \text{address_valid?}(\text{data}, a) \supset 1 < \text{length}[\text{Byte}](\text{data})$$

Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of address_from_byte_TCC2.
 Q.E.D.

C.5.6 Address_Model.address_from_byte_TCC3

Terse proof for address_from_byte_TCC3.

address_from_byte_TCC3:

$$\frac{}{\{1\} \quad \forall (\text{data}: \text{list}[\text{Byte}], a: \text{Address}): \text{address_valid?}(\text{data}, a) \supset 2 < \text{length}[\text{Byte}] (\text{data})}$$

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of address_from_byte_TCC3.

Q.E.D.

C.5.7 Address_Model.address_from_byte_TCC4

Terse proof for address_from_byte_TCC4.

address_from_byte_TCC4:

$$\frac{}{\{1\} \quad \forall (\text{data}: \text{list}[\text{Byte}], a: \text{Address}): \text{address_valid?}(\text{data}, a) \supset 3 < \text{length}[\text{Byte}] (\text{data})}$$

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of address_from_byte_TCC4.

Q.E.D.

C.5.8 Address_Model.address_from_byte_TCC5

Terse proof for address_from_byte_TCC5.

address_from_byte_TCC5:

```

{1}  ∀ (data: list[Byte], a: Address):
      address_valid?(data, a) ⊃
      Mem?(Mem(overwrite_shift_bits(overwrite_shift_bits(overwrite_shift_bits(overwrite_shift_bits
                                                                    (0,
                                                                    nth[Byte](data, 2), 16, 8),
                                                                    nth[Byte](data, 3), 24, 8))'type_of)
                                                                    nth[Byte](data, 2), 16, 8),
                                                                    nth[Byte](data, 3), 24, 8))'offset
      ^
      0 ≤
      Mem(overwrite_shift_bits(overwrite_shift_bits(overwrite_shift_bits(overwrite_shift_bits
                                                                    (0,
                                                                    nth[Byte](data, 2), 16, 8),
                                                                    nth[Byte](data, 3), 24, 8))'offset)
                                                                    nth[Byte](data, 2), 16, 8),
                                                                    nth[Byte](data, 3), 24, 8))'offset
      ^
      Mem(overwrite_shift_bits(overwrite_shift_bits(overwrite_shift_bits(overwrite_shift_bits
                                                                    (0,
                                                                    nth[Byte](data, 2), 16, 8),
                                                                    nth[Byte](data, 3), 24, 8))'offset)
                                                                    nth[Byte](data, 2), 16, 8),
                                                                    nth[Byte](data, 3), 24, 8))'offset
      < max_linear_offset

```

Repeatedly Skolemizing and flattening,

Installing automatic rewrites from: every aligned_address? aligned_overwrite_shift_bits_more aligned_overwrite_shift_bits_less aligned_zero max_linear_expt_val overwrite_shift_bits_less address_valid? max_linear_offset bus_width

Expanding the definition of Mem,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of address_from_byte_TCC5.

Q.E.D.

C.5.9 Address_Model.address_model_TCC1

Terse proof for address_model_TCC1.

address_model_TCC1:

$$\frac{\{1\} \quad \forall (f: [[\text{Memory_Address_4G}, \text{Address}] \rightarrow [\text{list}[\text{Byte}] \rightarrow \text{bool}]]) : (f = (\lambda (d: \text{Memory_Address_4G}, a: \text{Address}): (\text{zero_mask?}(4)))) \supset (\forall (x_1: [\text{Memory_Address_4G}, \text{Address}]): \text{singleton?}[\text{list}[\text{Byte}]](f(x_1))))}{}$$

Repeatedly Skolemizing and flattening,

Replacing using formula -1,

Using lemma singleton_zero_mask,

Expanding the definition of singleton?,

which is trivially true.

This completes the proof of address_model_TCC1.

Q.E.D.

C.5.10 Address_Model.address_valid_to_byte

Terse proof for address_valid_to_byte.

address_valid_to_byte:

$$\frac{\{1\} \quad \forall (d: \text{Memory_Address_4G}, a: \text{Address}): \text{valid?}(\text{uidt}(\text{address_model}))(\text{to_byte}(\text{address_model})(d, a), a)}{}$$

Installing automatic rewrites from: address_model address_to_byte length address_valid?

Repeatedly simplifying with decision procedures, rewriting, propositional reasoning, quantifier instantiation, skolemization, if-lifting and equality replacement,

This completes the proof of address_valid_to_byte.

Q.E.D.

C.5.11 Address_Model.address_is_pod

Terse proof for address_is_pod.

address_is_pod:

$$\frac{\{1\} \quad \text{pod_data_type?}(\text{address_model})}{}$$

Expanding the definition of pod_data_type?,

Expanding the definition of interpreted_data_type?,

Expanding the definition of uninterpreted_data_type?,

Applying propositional simplification,

we get 6 subgoals:

address_is_pod.1:

$$\frac{\{1\} \quad \forall (l: \text{list}[\text{Byte}], a: \text{Address}): \neg \text{length}(l) = \text{size}(\text{address_model}'\text{uidt}) \supset \neg \text{valid?}(\text{address_model}'\text{uidt})(l, a)}{}$$

Repeatedly Skolemizing and flattening,

Expanding the definition of address_model,

C Proof scripts

Expanding the definition of `address_valid?`,
which is trivially true.

This completes the proof of `address_is_pod.1`.
`address_is_pod.2`:

$$\frac{}{\{1\} \quad \forall (d: \text{Memory_Address_4G}, a: \text{Address}): \text{valid?}(\text{uidt}(\text{address_model}))(\text{to_byte}(\text{address_model})(d, a), a)}$$

Repeatedly Skolemizing and flattening,
Rewriting using `address_valid_to_byte`, matching in `*`,
This completes the proof of `address_is_pod.2`.
`address_is_pod.3`:

$$\frac{}{\{1\} \quad \forall (l: \text{list}[\text{Byte}], a: \text{Address}): \text{valid?}(\text{uidt}(\text{address_model}))(l, a) \equiv \text{up?}(\text{from_byte}(\text{address_model})(l, a))}$$

Repeatedly Skolemizing and flattening,
Expanding the definition of `address_model`,
Expanding the definition of `address_from_byte`,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `address_is_pod.3`.
`address_is_pod.4`:

$$\frac{}{\{1\} \quad \forall (d: \text{Memory_Address_4G}, a: \text{Address}): \text{down}(\text{from_byte}(\text{address_model})(\text{to_byte}(\text{address_model})(d, a), a)) = d}$$

Repeatedly Skolemizing and flattening,
Installing automatic rewrites from: `address_model address_from_byte address_to_byte address_valid?`
`length nth overwrite_shift_bits_zero cut_bits_cut_bits overwrite_shift_bits_merge max_linear_offset`
`bus_width zero_aligned`

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Rewriting using `shift_bits_left_cut_bits`, matching in `*`,
Applying extensionality,
we get 2 subgoals:
`address_is_pod.4.1`:

$$\frac{\begin{array}{l} \{-1\} \quad \text{Mem?}(d' \text{'type_of}) \\ \{-2\} \quad 0 \leq d' \text{'offset} \\ \{-3\} \quad d' \text{'offset} < \text{expt}(2, 32) \end{array}}{\{1\} \quad \text{Mem}(\text{offset}(d')) \text{'offset} = d' \text{'offset}}$$

Expanding the definition of `Mem`,
which is trivially true.
This completes the proof of `address_is_pod.4.1`.
`address_is_pod.4.2`:

$$\frac{\begin{array}{l} \{-1\} \quad \text{Mem?}(d' \text{'type_of}) \\ \{-2\} \quad 0 \leq d' \text{'offset} \\ \{-3\} \quad d' \text{'offset} < \text{expt}(2, 32) \end{array}}{\{1\} \quad \text{Mem}(\text{offset}(d')) \text{'type_of} = d' \text{'type_of}}$$

Expanding the definition of `Mem`,
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `address_is_pod.4.2`.

address_is_pod.5:

$$\{1\} \quad \forall (l: \text{list}[\text{Byte}], a_1, a_2: \text{Address}): \\ \text{valid?}(\text{uidt}(\text{address_model}))(l, a_1) \equiv \text{valid?}(\text{uidt}(\text{address_model}))(l, a_2)$$

Repeatedly Skolemizing and flattening,

Hiding formulas: -1,

Expanding the definition of address_model,

Expanding the definition of address_valid?,
which is trivially true.

This completes the proof of address_is_pod.5.

address_is_pod.6:

$$\{1\} \quad \text{size}(\text{uidt}(\text{address_model})) > 0$$

Expanding the definition of address_model,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of address_is_pod.6.

Q.E.D.

C.6 Proofs for Address_Util (memory.pvs)

C.6.1 Address_Util.address_block_subset_1

Terse proof for address_block_subset_1.

address_block_subset_1:

$$\{1\} \quad \forall (\text{addr}: \text{Address}, \text{size}: \text{nat}): \\ (\text{address_block}(\text{addr} + 1, \text{size}) \subseteq \text{address_block}(\text{addr}, 1 + \text{size}))$$

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of address_block_subset_1.

Q.E.D.

C.6.2 Address_Util.address_block_subset_2

Terse proof for address_block_subset_2.

address_block_subset_2:

$$\{1\} \quad \forall (\text{addresses}: \text{PRED}[\text{Address}], \text{addr}: \text{Address}, \text{size}: \text{nat}): \\ (\text{address_block}(\text{addr}, \text{size} + 1) \subseteq \text{addresses}) \supset \\ (\text{address_block}(\text{addr} + 1, \text{size}) \subseteq \text{addresses})$$

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of address_block_subset_2.

Q.E.D.

C.6.3 Address_Util.blocks_disjoint_disjoint

Terse proof for blocks_disjoint_disjoint.

C Proof scripts

blocks_disjoint_disjoint:

<pre>{1} ∃ (addr1, addr2: Address, size1, size2: nat): size1 > 0 ∧ size2 > 0 ⊃ blocks_disjoint?(addr1, size1, addr2, size2) = disjoint?(address_block(addr1, size1), address_block(addr2, size2))</pre>
--

Repeatedly Skolemizing and flattening,

Converting equality to IFF,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 2 subgoals:

blocks_disjoint_disjoint.1:

<pre>{-1} size1' ≥ 0 {-2} size2' ≥ 0 {-3} size1' > 0 {-4} size2' > 0 {-5} blocks_disjoint?(addr1', size1', addr2', size2')</pre>
<pre>{1} disjoint?(address_block(addr1', size1'), address_block(addr2', size2'))</pre>

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of **blocks_disjoint_disjoint.1**.

blocks_disjoint_disjoint.2:

<pre>{-1} size1' ≥ 0 {-2} size2' ≥ 0 {-3} size1' > 0 {-4} size2' > 0 {-5} disjoint?(address_block(addr1', size1'), address_block(addr2', size2'))</pre>
<pre>{1} blocks_disjoint?(addr1', size1', addr2', size2')</pre>

Expanding the definition of **blocks_disjoint?**,

Applying disjunctive simplification to flatten sequent,

Expanding the definition(s) of (**disjoint?** **empty?** **intersection member**),

Case splitting on **addr1!1 < addr2!1**,

we get 2 subgoals:

blocks_disjoint_disjoint.2.1:

<pre>{-1} addr1' < addr2' {-2} size1' ≥ 0 {-3} size2' ≥ 0 {-4} size1' > 0 {-5} size2' > 0 {-6} type_of(addr1') = type_of(addr2') {-7} ∃ (x: Address): ¬ (address_block(addr1', size1')(x) ∧ address_block(addr2', size2')(x))</pre>
<pre>{1} addr1' + size1' ≤ addr2'</pre>
<pre>{2} addr2' + size2' ≤ addr1'</pre>

Instantiating the top quantifier in -7 with the terms: **addr2'**,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of **blocks_disjoint_disjoint.2.1**.

blocks_disjoint_disjoint.2.2:

{-1}	$size1' \geq 0$
{-2}	$size2' \geq 0$
{-3}	$size1' > 0$
{-4}	$size2' > 0$
{-5}	$type_of(addr1') = type_of(addr2')$
{-6}	$\forall (x: Address):$
$\neg (address_block(addr1', size1')(x) \wedge address_block(addr2', size2')(x))$	
{1}	$addr1' < addr2'$
{2}	$addr1' + size1' \leq addr2'$
{3}	$addr2' + size2' \leq addr1'$

Instantiating the top quantifier in -6 with the terms: $addr1'$,
 Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of `blocks_disjoint_disjoint.2.2`.
 Q.E.D.

C.6.4 Address_Util.blocks_disjoint_symmetric

Terse proof for `blocks_disjoint_symmetric`.

blocks_disjoint_symmetric:

{1}	$\forall (addr1, addr2: Address, size1, size2: nat):$
$blocks_disjoint?(addr1, size1, addr2, size2) = blocks_disjoint?(addr2, size2, addr1, size1)$	

Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of `blocks_disjoint_symmetric`.
 Q.E.D.

C.6.5 Address_Util.blocks_in_larger_set

Terse proof for `blocks_in_larger_set`.

blocks_in_larger_set:

{1}	$\forall (addr1, addr2: Address, size1, size2: nat, addresses: PRED[Address]):$
$(address_block(addr1, size1) \subseteq addresses) \wedge$	
$(address_block(addr2, size2) \subseteq addresses) \wedge$	
$blocks_disjoint?(addr1, size1, addr2, size2)$	
$\supset (address_block(addr2, size2) \subseteq (addresses \setminus address_block(addr1, size1)))$	

Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of `blocks_in_larger_set`.
 Q.E.D.

C.6.6 Address_Util.blocks_pairwise_disjoint_add

Terse proof for `blocks_pairwise_disjoint_add`.

blocks_pairwise_disjoint_add:

<pre>{1} ∀ (addr: Address, size: posnat, blocks: list[[Address, posnat]]): blocks_pairwise_disjoint(blocks) ∧ (∀ (oaddr: Address, osize: posnat): member((oaddr, osize), blocks) ⊃ blocks_disjoint?(addr, size, oaddr, osize)) ⊃ blocks_pairwise_disjoint(cons((addr, size), blocks))</pre>

Expanding the definition of blocks_pairwise_disjoint,
 Repeatedly Skolemizing and flattening,
 Expanding the definition of member,
 Case splitting on addr!2 = addr!1 AND size!2 = size!1,
 we get 2 subgoals:

blocks_pairwise_disjoint_add.1:

<pre>{-1} addr'' = addr' ∧ size'' = size' {-2} size'' > 0 {-3} size''' > 0 {-4} size' > 0 {-5} ∀ (addr_1, addr_2: Address, size_1, size_2: posnat): member((addr_1, size_1), blocks') ∧ member((addr_2, size_2), list_remove((addr_1, size_1), blocks')) ⊃ blocks_disjoint?(addr_1, size_1, addr_2, size_2) {-6} ∀ (oaddr: Address, osize: posnat): member((oaddr, osize), blocks') ⊃ blocks_disjoint?(addr', size', oaddr, osize) {-7} addr'' = addr' ∧ size'' = size' ∨ member((addr'', size''), blocks') {-8} member((addr''', size'''), list_remove((addr'', size''), cons((addr', size'), blocks')))</pre>
<pre>{1} blocks_disjoint?(addr'', size'', addr''', size''')</pre>

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 Expanding the definition of list_remove,
 Rewriting using member_list_remove, matching in *,
 Instantiating the top quantifier in -7 with the terms: addr''', size''',
 Simplifying, rewriting, and recording with decision procedures,
 This completes the proof of blocks_pairwise_disjoint_add.1.

blocks_pairwise_disjoint_add.2:

<pre>{-1} size'' > 0 {-2} size''' > 0 {-3} size' > 0 {-4} ∀ (addr_1, addr_2: Address, size_1, size_2: posnat): member((addr_1, size_1), blocks') ∧ member((addr_2, size_2), list_remove((addr_1, size_1), blocks')) ⊃ blocks_disjoint?(addr_1, size_1, addr_2, size_2) {-5} ∀ (oaddr: Address, osize: posnat): member((oaddr, osize), blocks') ⊃ blocks_disjoint?(addr', size', oaddr, osize) {-6} addr'' = addr' ∧ size'' = size' ∨ member((addr'', size''), blocks') {-7} member((addr''', size'''), list_remove((addr'', size''), cons((addr', size'), blocks')))</pre>
<pre>{1} addr'' = addr' ∧ size'' = size' {2} blocks_disjoint?(addr'', size'', addr''', size''')</pre>

Splitting conjunctions,
 we get 2 subgoals:

blocks_pairwise_disjoint_add.2.1:

{-1}	$\text{addr}'' = \text{addr}' \wedge \text{size}'' = \text{size}'$
{-2}	$\text{size}'' > 0$
{-3}	$\text{size}''' > 0$
{-4}	$\text{size}' > 0$
{-5}	$\forall (\text{addr}_1, \text{addr}_2: \text{Address}, \text{size}_1, \text{size}_2: \text{posnat}):$ $\text{member}((\text{addr}_1, \text{size}_1), \text{blocks}') \wedge$ $\text{member}((\text{addr}_2, \text{size}_2), \text{list_remove}((\text{addr}_1, \text{size}_1), \text{blocks}'))$ $\supset \text{blocks_disjoint?}(\text{addr}_1, \text{size}_1, \text{addr}_2, \text{size}_2)$
{-6}	$\forall (\text{oaddr}: \text{Address}, \text{osize}: \text{posnat}):$ $\text{member}((\text{oaddr}, \text{osize}), \text{blocks}') \supset \text{blocks_disjoint?}(\text{addr}', \text{size}', \text{oaddr}, \text{osize})$
{-7}	$\text{member}((\text{addr}''', \text{size}'''),$ $\text{list_remove}((\text{addr}'', \text{size}''), \text{cons}((\text{addr}', \text{size}'), \text{blocks}'))$
{1}	$\text{addr}'' = \text{addr}' \wedge \text{size}'' = \text{size}'$
{2}	$\text{blocks_disjoint?}(\text{addr}'', \text{size}'', \text{addr}''', \text{size}''')$

which is trivially true.

This completes the proof of blocks_pairwise_disjoint_add.2.1.

blocks_pairwise_disjoint_add.2.2:

{-1}	$\text{member}((\text{addr}'', \text{size}''), \text{blocks}')$
{-2}	$\text{size}'' > 0$
{-3}	$\text{size}''' > 0$
{-4}	$\text{size}' > 0$
{-5}	$\forall (\text{addr}_1, \text{addr}_2: \text{Address}, \text{size}_1, \text{size}_2: \text{posnat}):$ $\text{member}((\text{addr}_1, \text{size}_1), \text{blocks}') \wedge$ $\text{member}((\text{addr}_2, \text{size}_2), \text{list_remove}((\text{addr}_1, \text{size}_1), \text{blocks}'))$ $\supset \text{blocks_disjoint?}(\text{addr}_1, \text{size}_1, \text{addr}_2, \text{size}_2)$
{-6}	$\forall (\text{oaddr}: \text{Address}, \text{osize}: \text{posnat}):$ $\text{member}((\text{oaddr}, \text{osize}), \text{blocks}') \supset \text{blocks_disjoint?}(\text{addr}', \text{size}', \text{oaddr}, \text{osize})$
{-7}	$\text{member}((\text{addr}''', \text{size}'''),$ $\text{list_remove}((\text{addr}'', \text{size}''), \text{cons}((\text{addr}', \text{size}'), \text{blocks}'))$
{1}	$\text{addr}'' = \text{addr}' \wedge \text{size}'' = \text{size}'$
{2}	$\text{blocks_disjoint?}(\text{addr}'', \text{size}'', \text{addr}''', \text{size}''')$

Expanding the definition of list_remove,

Lifting IF-conditions to the top level,

Splitting conjunctions,

we get 2 subgoals:

blocks_pairwise_disjoint_add.2.2.1:

{-1}	$(\text{addr}'' = \text{addr}' \wedge \text{size}'' = \text{size}') \wedge$ $\text{member}((\text{addr}''', \text{size}'''), \text{list_remove}((\text{addr}'', \text{size}''), \text{blocks}'))$
{-2}	$\text{member}((\text{addr}'', \text{size}''), \text{blocks}')$
{-3}	$\text{size}'' > 0$
{-4}	$\text{size}''' > 0$
{-5}	$\text{size}' > 0$
{-6}	$\forall (\text{addr}_1, \text{addr}_2: \text{Address}, \text{size}_1, \text{size}_2: \text{posnat}):$ $\text{member}((\text{addr}_1, \text{size}_1), \text{blocks}') \wedge$ $\text{member}((\text{addr}_2, \text{size}_2), \text{list_remove}((\text{addr}_1, \text{size}_1), \text{blocks}'))$ $\supset \text{blocks_disjoint?}(\text{addr}_1, \text{size}_1, \text{addr}_2, \text{size}_2)$
{-7}	$\forall (\text{oaddr}: \text{Address}, \text{osize}: \text{posnat}):$ $\text{member}((\text{oaddr}, \text{osize}), \text{blocks}') \supset \text{blocks_disjoint?}(\text{addr}', \text{size}', \text{oaddr}, \text{osize})$
{1}	$\text{addr}'' = \text{addr}' \wedge \text{size}'' = \text{size}'$
{2}	$\text{blocks_disjoint?}(\text{addr}'', \text{size}'', \text{addr}''', \text{size}''')$

C Proof scripts

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `blocks_pairwise_disjoint_add.2.2.1`.

`blocks_pairwise_disjoint_add.2.2.2`:

{-1}	$\neg (\text{addr}'' = \text{addr}' \wedge \text{size}'' = \text{size}') \wedge$ $\text{member}((\text{addr}''', \text{size}'''),$ $\quad \text{cons}((\text{addr}', \text{size}'), \text{list_remove}((\text{addr}'', \text{size}''), \text{blocks}'))$
{-2}	$\text{member}((\text{addr}'', \text{size}''), \text{blocks}')$
{-3}	$\text{size}'' > 0$
{-4}	$\text{size}''' > 0$
{-5}	$\text{size}' > 0$
{-6}	$\forall (\text{addr}_1, \text{addr}_2: \text{Address}, \text{size}_1, \text{size}_2: \text{posnat}):$ $\text{member}((\text{addr}_1, \text{size}_1), \text{blocks}') \wedge$ $\text{member}((\text{addr}_2, \text{size}_2), \text{list_remove}((\text{addr}_1, \text{size}_1), \text{blocks}'))$ $\supset \text{blocks_disjoint?}(\text{addr}_1, \text{size}_1, \text{addr}_2, \text{size}_2)$
{-7}	$\forall (\text{oaddr}: \text{Address}, \text{osize}: \text{posnat}):$ $\text{member}((\text{oaddr}, \text{osize}), \text{blocks}') \supset \text{blocks_disjoint?}(\text{addr}', \text{size}', \text{oaddr}, \text{osize})$
{1} $\text{addr}'' = \text{addr}' \wedge \text{size}'' = \text{size}'$	
{2} $\text{blocks_disjoint?}(\text{addr}'', \text{size}'', \text{addr}''', \text{size}''')$	

Applying disjunctive simplification to flatten sequent,

Hiding formulas: 1,

Expanding the definition of member,

Splitting conjunctions,

we get 2 subgoals:

`blocks_pairwise_disjoint_add.2.2.2.1`:

{-1}	$\text{addr}''' = \text{addr}' \wedge \text{size}''' = \text{size}'$
{-2}	$\text{member}((\text{addr}'', \text{size}''), \text{blocks}')$
{-3}	$\text{size}'' > 0$
{-4}	$\text{size}''' > 0$
{-5}	$\text{size}' > 0$
{-6}	$\forall (\text{addr}_1, \text{addr}_2: \text{Address}, \text{size}_1, \text{size}_2: \text{posnat}):$ $\text{member}((\text{addr}_1, \text{size}_1), \text{blocks}') \wedge$ $\text{member}((\text{addr}_2, \text{size}_2), \text{list_remove}((\text{addr}_1, \text{size}_1), \text{blocks}'))$ $\supset \text{blocks_disjoint?}(\text{addr}_1, \text{size}_1, \text{addr}_2, \text{size}_2)$
{-7}	$\forall (\text{oaddr}: \text{Address}, \text{osize}: \text{posnat}):$ $\text{member}((\text{oaddr}, \text{osize}), \text{blocks}') \supset \text{blocks_disjoint?}(\text{addr}', \text{size}', \text{oaddr}, \text{osize})$
{1} $\text{addr}'' = \text{addr}' \wedge \text{size}'' = \text{size}'$	
{2} $\text{blocks_disjoint?}(\text{addr}'', \text{size}'', \text{addr}''', \text{size}''')$	

Applying disjunctive simplification to flatten sequent,

Instantiating the top quantifier in -8 with the terms: `addr''`, `size''`,

Simplifying, rewriting, and recording with decision procedures,

Rewriting using `blocks_disjoint_symmetric`, matching in *,

This completes the proof of `blocks_pairwise_disjoint_add.2.2.2.1`.

blocks_pairwise_disjoint_add.2.2.2.2:

{-1}	member((addr''', size'''), list_remove((addr'', size''), blocks'))
{-2}	member((addr'', size''), blocks')
{-3}	size'' > 0
{-4}	size''' > 0
{-5}	size' > 0
{-6}	\forall (addr_1, addr_2: Address, size_1, size_2: posnat): member((addr_1, size_1), blocks') \wedge member((addr_2, size_2), list_remove((addr_1, size_1), blocks')) \supset blocks_disjoint?(addr_1, size_1, addr_2, size_2)
{-7}	\forall (oaddr: Address, osize: posnat): member((oaddr, osize), blocks') \supset blocks_disjoint?(addr', size', oaddr, osize)
{1}	addr''' = addr' \wedge size''' = size'
{2}	blocks_disjoint?(addr'', size'', addr''', size''')

Instantiating the top quantifier in -6 with the terms: addr'', addr''', size'', size''',
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of blocks_pairwise_disjoint_add.2.2.2.2.
Q.E.D.

C.6.7 Address_Util.blocks_pairwise_disjoint_remove

Terse proof for blocks_pairwise_disjoint_remove.

blocks_pairwise_disjoint_remove:

{1}	\forall (addr: Address, size: posnat, blocks: list[[Address, posnat]]): blocks_pairwise_disjoint(blocks) \supset blocks_pairwise_disjoint(list_remove((addr, size), blocks))
-----	--

Expanding the definition of blocks_pairwise_disjoint,
Repeatedly Skolemizing and flattening,
Rewriting using list_remove_commutative, matching in *,
Rewriting using member_list_remove, matching in *,
Rewriting using member_list_remove, matching in *,
Instantiating the top quantifier in -4 with the terms: addr'', addr''', size'', size''',
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of blocks_pairwise_disjoint_remove.
Q.E.D.

C.6.8 Address_Util.subset_block_is_free

Terse proof for subset_block_is_free.

subset_block_is_free:

{1}	\forall (b ₁ , b ₂ : PRED[Address], alloc_points: list[[Address, posnat]]): (b ₁ \subseteq b ₂) \wedge block_is_free(b ₂ , alloc_points) \supset block_is_free(b ₁ , alloc_points)
-----	--

Repeatedly Skolemizing and flattening,
Expanding the definition of block_is_free,
Repeatedly Skolemizing and flattening,
Instantiating quantified variables,
Simplifying, rewriting, and recording with decision procedures,

C Proof scripts

Using lemma `disjoint_mono`,
Simplifying, rewriting, and recording with decision procedures,
Rewriting using `subset_equal`, matching in `*`,
This completes the proof of `subset_block_is_free`.
Q.E.D.

C.6.9 Address_Util.block_is_free_add

Terse proof for `block_is_free_add`.

`block_is_free_add`:

$$\{1\} \quad \forall (\text{block: PRED}[\text{Address}], \text{alloc_points: list}[[\text{Address}, \text{posnat}]], \text{addr: Address}, \\ \text{size: posnat}): \\ \text{block_is_free}(\text{block}, \text{alloc_points}) \wedge \text{disjoint?}(\text{block}, \text{address_block}(\text{addr}, \text{size})) \supset \\ \text{block_is_free}(\text{block}, \text{cons}((\text{addr}, \text{size}), \text{alloc_points}))$$

Repeatedly Skolemizing and flattening,
Expanding the definition of `block_is_free`,
Repeatedly Skolemizing and flattening,
Installing automatic rewrites from: `member`
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Instantiating the top quantifier in -3 with the terms: `addr''`, `size''`,
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `block_is_free_add`.
Q.E.D.

C.7 Proofs for Alignment (`vfiasco-prelude.pvs`)

C.7.1 Alignment.aligned_zero

Terse proof for `aligned_zero`.

`aligned_zero`:

$$\{1\} \quad \forall (n: \text{nat}): \text{aligned?}(n)(0)$$

Repeatedly Skolemizing and flattening,
Expanding the definition of `aligned?`,
Rewriting using `divides_zero`, matching in `*`,
This completes the proof of `aligned_zero`.
Q.E.D.

C.7.2 Alignment.zero_aligned

Terse proof for `zero_aligned`.

`zero_aligned`:

$$\{1\} \quad \forall (n: \text{nat}): \text{aligned?}(0)(n)$$

Repeatedly Skolemizing and flattening,
Expanding the definition of `aligned?`,
Expanding the definition of `expt`,

Rewriting using `one_divides`, matching in `*`,
 This completes the proof of `zero_aligned`.
 Q.E.D.

C.7.3 Alignment.aligned_bigger

Terse proof for `aligned_bigger`.

`aligned_bigger`:

$$\frac{}{\{1\} \quad \forall (m, n_1, n_2: \text{nat}): n_2 \leq n_1 \wedge \text{aligned?}(n_1)(m) \supset \text{aligned?}(n_2)(m)}$$

Repeatedly Skolemizing and flattening,
 Expanding the definition of `aligned?`,
 Forward chaining on `divides_expt_gt`,
 This completes the proof of `aligned_bigger`.
 Q.E.D.

C.7.4 Alignment.aligned_plus

Terse proof for `aligned_plus`.

`aligned_plus`:

$$\frac{}{\{1\} \quad \forall (m, n: \text{nat}, a: \text{aligned}(n), b: \text{aligned}(m)):\text{aligned?}(\min(n, m))(a + b)}$$

Repeatedly Skolemizing and flattening,
 Expanding the definition of `min`,
 Expanding the definition of `aligned?`,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 we get 2 subgoals:

`aligned_plus.1`:

$$\frac{\begin{array}{l} \{-1\} \quad m' \geq 0 \\ \{-2\} \quad n' \geq 0 \\ \{-3\} \quad \text{divides}(\text{expt}(2, n'), a') \\ \{-4\} \quad \text{divides}(\text{expt}(2, m'), b') \\ \{-5\} \quad n' > m' \end{array}}{\{1\} \quad \text{divides}(\text{expt}(2, m'), a' + b')}$$

Rewriting using `divides_sum`, matching in `*`,
 Hiding formulas: -4, 2,
 Using lemma `expt_divides`,
 Simplifying, rewriting, and recording with decision procedures,
 Using lemma `divides_transitive`,
 Simplifying, rewriting, and recording with decision procedures,
 This completes the proof of `aligned_plus.1`.

`aligned_plus.2:`

{-1}	$m' \geq 0$
{-2}	$n' \geq 0$
{-3}	$\text{divides}(\text{expt}(2, n'), a')$
{-4}	$\text{divides}(\text{expt}(2, m'), b')$
{1}	$n' > m'$
{2}	$\text{divides}(\text{expt}(2, n'), a' + b')$

Rewriting using `divides_sum`, matching in `*`,
Hiding formulas: -3, 3,
Using lemma `expt_divides`,
Simplifying, rewriting, and recording with decision procedures,
Using lemma `divides_transitive`,
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `aligned_plus.2`.
Q.E.D.

C.7.5 Alignment.`aligned_plus_n`

Terse proof for `aligned_plus_n`.

`aligned_plus_n:`

{1}	$\forall (n: \text{nat}, a, b: \text{aligned}(n)): \text{aligned?}(n)(a + b)$
-----	---

Repeatedly Skolemizing and flattening,
Using lemma `aligned_plus`,
Expanding the definition of `min`,
which is trivially true.
This completes the proof of `aligned_plus_n`.
Q.E.D.

C.7.6 Alignment.`aligned_minus_TCC1`

Terse proof for `aligned_minus_TCC1`.

`aligned_minus_TCC1:`

{1}	$\forall (m, n: \text{nat}, a: \text{aligned}(n), b: \text{aligned}(m)): a \geq b \supset a - b \geq 0$
-----	---

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `aligned_minus_TCC1`.
Q.E.D.

C.7.7 Alignment.`aligned_minus`

Terse proof for `aligned_minus`.

`aligned_minus:`

{1}	$\forall (m, n: \text{nat}, a: \text{aligned}(n), b: \text{aligned}(m)):$ $a \geq b \supset \text{aligned?}(\text{min}(n, m))(a - b)$
-----	--

Repeatedly Skolemizing and flattening,

Expanding the definition of `min`,
 Expanding the definition of `aligned?`,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 we get 2 subgoals:

`aligned_minus.1`:

{-1}	$m' \geq 0$
{-2}	$n' \geq 0$
{-3}	$\text{divides}(\text{expt}(2, n'), a')$
{-4}	$\text{divides}(\text{expt}(2, m'), b')$
{-5}	$a' \geq b'$
{-6}	$n' > m'$
{1}	
	$\text{divides}(\text{expt}(2, m'), a' - b')$

Rewriting using `divides_diff`, matching in `*`,
 Hiding formulas: -4, 2,
 Using lemma `expt_divides`,
 Simplifying, rewriting, and recording with decision procedures,
 Using lemma `divides_transitive`,
 Simplifying, rewriting, and recording with decision procedures,
 This completes the proof of `aligned_minus.1`.

`aligned_minus.2`:

{-1}	$m' \geq 0$
{-2}	$n' \geq 0$
{-3}	$\text{divides}(\text{expt}(2, n'), a')$
{-4}	$\text{divides}(\text{expt}(2, m'), b')$
{-5}	$a' \geq b'$
{1}	
	$n' > m'$
{2}	
	$\text{divides}(\text{expt}(2, n'), a' - b')$

Rewriting using `divides_diff`, matching in `*`,
 Hiding formulas: -3, 3,
 Using lemma `expt_divides`,
 Simplifying, rewriting, and recording with decision procedures,
 Using lemma `divides_transitive`,
 Simplifying, rewriting, and recording with decision procedures,
 This completes the proof of `aligned_minus.2`.
 Q.E.D.

C.7.8 Alignment.`aligned_mult_expt_2`

Terse proof for `aligned_mult_expt_2`.

`aligned_mult_expt_2`:

{1}	$\forall (m, n: \text{nat}, a: \text{aligned}(n)): \text{aligned?}(n + m)(a \times \text{expt}(2, m))$
-----	--

Repeatedly Skolemizing and flattening,
 Expanding the definition of `aligned?`,
 Rewriting using `expt_plus_aux`, matching in `*`,
 Using lemma `divides_prod_elim1`,
 Simplifying, rewriting, and recording with decision procedures,
 This completes the proof of `aligned_mult_expt_2`.
 Q.E.D.

C.7.9 Alignment.aligned_rem

Terse proof for `aligned_rem`.

`aligned_rem`:

$$\frac{}{\{1\} \quad \forall (m, n, i: \text{nat}): \text{aligned?}(n)(m) \supset \text{aligned?}(n)(\text{rem}(\text{expt}(2, i))(m))}$$

Repeatedly Skolemizing and flattening,

Expanding the definition of `aligned?`,

Applying `rem_def2`

Instantiating the top quantifier in -1 with the terms: `expt(2, n')`, `rem(expt(2, i'))(m')`, 0,

we get 2 subgoals:

`aligned_rem.1`:

$$\frac{\begin{array}{l} \{-1\} \quad \text{rem}(\text{expt}(2, n'))(\text{rem}(\text{expt}(2, i'))(m')) = 0 \equiv \\ \quad \text{divides}(\text{expt}(2, n'), \text{rem}(\text{expt}(2, i'))(m') - 0) \\ \{-2\} \quad m' \geq 0 \\ \{-3\} \quad n' \geq 0 \\ \{-4\} \quad i' \geq 0 \\ \{-5\} \quad \text{divides}(\text{expt}(2, n'), m') \end{array}}{\{1\} \quad \text{divides}(\text{expt}(2, n'), \text{rem}(\text{expt}(2, i'))(m'))}$$

Simplifying, rewriting, and recording with decision procedures,

Hiding formulas: 2,

Case splitting on `i!1 <= n!1`,

we get 2 subgoals:

`aligned_rem.1.1`:

$$\frac{\begin{array}{l} \{-1\} \quad i' \leq n' \\ \{-2\} \quad m' \geq 0 \\ \{-3\} \quad n' \geq 0 \\ \{-4\} \quad i' \geq 0 \\ \{-5\} \quad \text{divides}(\text{expt}(2, n'), m') \end{array}}{\{1\} \quad \text{rem}(\text{expt}(2, n'))(\text{rem}(\text{expt}(2, i'))(m')) = 0}$$

Rewriting using `rem_mod`, matching in `*`,

we get 2 subgoals:

`aligned_rem.1.1.1`:

$$\frac{\begin{array}{l} \{-1\} \quad i' \leq n' \\ \{-2\} \quad m' \geq 0 \\ \{-3\} \quad n' \geq 0 \\ \{-4\} \quad i' \geq 0 \\ \{-5\} \quad \text{divides}(\text{expt}(2, n'), m') \end{array}}{\{1\} \quad \text{rem}(\text{expt}(2, i'))(m') = 0}$$

Using lemma `rem_def2`,

Simplifying, rewriting, and recording with decision procedures,

Forward chaining on `divides_expt_gt`,

This completes the proof of `aligned_rem.1.1.1`.

aligned_rem.1.1.2:

{-1}	$i' \leq n'$
{-2}	$m' \geq 0$
{-3}	$n' \geq 0$
{-4}	$i' \geq 0$
{-5}	$\text{divides}(\text{expt}(2, n'), m')$
{1}	$\text{rem}(\text{expt}(2, i'))(m') < \text{expt}(2, n')$
{2}	$\text{rem}(\text{expt}(2, n'))(\text{rem}(\text{expt}(2, i'))(m')) = 0$

Using lemma both_sides_expt_gt1_le,
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of aligned_rem.1.1.2.

aligned_rem.1.2:

{-1}	$m' \geq 0$
{-2}	$n' \geq 0$
{-3}	$i' \geq 0$
{-4}	$\text{divides}(\text{expt}(2, n'), m')$
{1}	$i' \leq n'$
{2}	$\text{rem}(\text{expt}(2, n'))(\text{rem}(\text{expt}(2, i'))(m')) = 0$

Rewriting using rem_rem, matching in *,
we get 2 subgoals:

aligned_rem.1.2.1:

{-1}	$m' \geq 0$
{-2}	$n' \geq 0$
{-3}	$i' \geq 0$
{-4}	$\text{divides}(\text{expt}(2, n'), m')$
{1}	$i' \leq n'$
{2}	$\text{rem}(\text{expt}(2, n'))(m') = 0$

Rewriting using rem_def2, matching in *,
This completes the proof of aligned_rem.1.2.1.

aligned_rem.1.2.2:

{-1}	$m' \geq 0$
{-2}	$n' \geq 0$
{-3}	$i' \geq 0$
{-4}	$\text{divides}(\text{expt}(2, n'), m')$
{1}	$\text{divides}(\text{expt}(2, n'), \text{expt}(2, i'))$
{2}	$i' \leq n'$
{3}	$\text{rem}(\text{expt}(2, n'))(\text{rem}(\text{expt}(2, i'))(m')) = 0$

Rewriting using expt_divides, matching in *,
This completes the proof of aligned_rem.1.2.2.

aligned_rem.2:

{-1}	$m' \geq 0$
{-2}	$n' \geq 0$
{-3}	$i' \geq 0$
{-4}	$\text{divides}(\text{expt}(2, n'), m')$
{1}	$0 < \text{expt}(2, n')$
{2}	$\text{divides}(\text{expt}(2, n'), \text{rem}(\text{expt}(2, i'))(m'))$

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `aligned_rem.2`.

Q.E.D.

C.7.10 Alignment.`aligned_add_below`

Terse proof for `aligned_add_below`.

`aligned_add_below`:

$$\frac{\{1\} \quad \forall (n, m, i, k: \text{nat}): \quad i \leq k \wedge n < \text{expt}(2, k) \wedge \text{aligned?}(i)(n) \wedge m < \text{expt}(2, i) \supset \quad n + m < \text{expt}(2, k)}{\quad}$$

Repeatedly Skolemizing and flattening,

Expanding the definition of `aligned?`,

Expanding the definition of `divides`,

Repeatedly Skolemizing and flattening,

Replacing using formula -8,

Applying `bit_split_less`

Instantiating the top quantifier in -1 with the terms: $x', m', k' - i', i'$,

we get 3 subgoals:

`aligned_add_below.1`:

$$\frac{\begin{array}{l} \{-1\} \quad m' < \text{expt}(2, i') \supset \\ \quad (x' \times \text{expt}(2, i') + m' < \text{expt}(2, i' + k' - i') \equiv \\ \quad \quad x' < \text{expt}(2, k' - i')) \\ \{-2\} \quad \text{integer_pred}(x') \\ \{-3\} \quad \text{expt}(2, i') \times x' \geq 0 \\ \{-4\} \quad m' \geq 0 \\ \{-5\} \quad i' \geq 0 \\ \{-6\} \quad k' \geq 0 \\ \{-7\} \quad i' \leq k' \\ \{-8\} \quad \text{expt}(2, i') \times x' < \text{expt}(2, k') \\ \{-9\} \quad n' = \text{expt}(2, i') \times x' \\ \{-10\} \quad m' < \text{expt}(2, i') \end{array}}{\{1\} \quad \text{expt}(2, i') \times x' + m' < \text{expt}(2, k')}$$

Simplifying, rewriting, and recording with decision procedures,

Applying `both_sides_times_pos_lt2`

Instantiating the top quantifier in -1 with the terms: $\text{expt}(2, i'), x', \text{expt}(2, k' - i')$,

Simplifying, rewriting, and recording with decision procedures,

Rewriting using `expt_plus`, matching in `*`,

This completes the proof of `aligned_add_below.1`.

aligned_add_below.2:

{-1}	integer_pred(x')
{-2}	$\text{expt}(2, i') \times x' \geq 0$
{-3}	$m' \geq 0$
{-4}	$i' \geq 0$
{-5}	$k' \geq 0$
{-6}	$i' \leq k'$
{-7}	$\text{expt}(2, i') \times x' < \text{expt}(2, k')$
{-8}	$n' = \text{expt}(2, i') \times x'$
{-9}	$m' < \text{expt}(2, i')$
{1}	$k' - i' \geq 0$
{2}	$\text{expt}(2, i') \times x' + m' < \text{expt}(2, k')$

Applying both_sides_expt_gt1_le

Instantiating the top quantifier in -1 with the terms: 2, i' , k' ,
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of aligned_add_below.2.

aligned_add_below.3:

{-1}	integer_pred(x')
{-2}	$\text{expt}(2, i') \times x' \geq 0$
{-3}	$m' \geq 0$
{-4}	$i' \geq 0$
{-5}	$k' \geq 0$
{-6}	$i' \leq k'$
{-7}	$\text{expt}(2, i') \times x' < \text{expt}(2, k')$
{-8}	$n' = \text{expt}(2, i') \times x'$
{-9}	$m' < \text{expt}(2, i')$
{1}	$x' \geq 0$
{2}	$\text{expt}(2, i') \times x' + m' < \text{expt}(2, k')$

Applying pos_times_le

Instantiating the top quantifier in -1 with the terms: $\text{expt}(2, i')$, x' ,
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of aligned_add_below.3.
Q.E.D.

C.8 Proofs for Allocation_Info (allocators.pvs)

C.8.1 Allocation_Info.consistent_allocator_state_change

Terse proof for consistent_allocator_state_change.

consistent_allocator_state_change:

{1}	$\forall (\text{ai}: \text{Allocator_Info}, \text{pm}: (\text{plain_memory?}), \text{ac}: (\text{allocator?}(\text{pm})), s_1, s_2: \text{State}):$ $\text{consistent_allocator?}(\text{pm})(\text{ac}, \text{ai})(s_1) \wedge$ $\text{ac}'\text{allocated}(\text{pm})(s_1) = \text{ac}'\text{allocated}(\text{pm})(s_2)$ $\supset \text{consistent_allocator?}(\text{pm})(\text{ac}, \text{ai})(s_2)$
-----	---

Repeatedly Skolemizing and flattening,

Expanding the definition of consistent_allocator?,

Applying disjunctive simplification to flatten sequent,

C Proof scripts

Simplifying, rewriting, and recording with decision procedures,
Repeatedly Skolemizing and flattening,
Instantiating quantified variables,
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `consistent_allocator_state_change`.
Q.E.D.

C.8.2 Allocation_Info.consistent_allocator_read_TCC1

Terse proof for `consistent_allocator_read_TCC1`.

`consistent_allocator_read_TCC1`:

$$\begin{array}{|l} \hline \{1\} \quad \forall (ai: \text{Allocator_Info}, pm: (\text{plain_memory?}), s: \text{State}, ac: (\text{allocator?}[\text{State}](pm)), \\ \quad \text{addr: Address}): \\ \quad pm' \text{states}(s) \wedge \\ \quad \text{consistent_allocator?}(pm)(ac, ai)(s) \wedge \text{union}(pm' \text{ro_addr}, pm' \text{rw_addr})(\text{addr}) \\ \quad \supset \\ \quad \text{OK?}[\text{State}, \text{Byte}](\text{memory_read}(pm' \text{mem})(\text{addr})(s)) \vee \\ \quad \text{Exception?}[\text{State}, \text{Byte}](\text{memory_read}(pm' \text{mem})(\text{addr})(s)) \\ \hline \end{array}$$

Repeatedly Skolemizing and flattening,
Rewriting using `plain_memory_memory_read_ok`, matching in *,
This completes the proof of `consistent_allocator_read_TCC1`.
Q.E.D.

C.8.3 Allocation_Info.consistent_allocator_read

Terse proof for `consistent_allocator_read`.

`consistent_allocator_read`:

$$\begin{array}{|l} \hline \{1\} \quad \forall (ai: \text{Allocator_Info}, pm: (\text{plain_memory?}), s: \text{State}, ac: (\text{allocator?}(pm)), \\ \quad \text{addr: Address}): \\ \quad pm' \text{states}(s) \wedge \\ \quad \text{consistent_allocator?}(pm)(ac, ai)(s) \wedge \text{union}(pm' \text{ro_addr}, pm' \text{rw_addr})(\text{addr}) \\ \quad \supset \text{consistent_allocator?}(pm)(ac, ai)(\text{state}(\text{memory_read}(pm' \text{mem})(\text{addr})(s))) \\ \hline \end{array}$$

Repeatedly Skolemizing and flattening,
Expanding the definition of `consistent_allocator?`,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Repeatedly Skolemizing and flattening,
Instantiating quantified variables,
Simplifying, rewriting, and recording with decision procedures,
Rewriting using `allocator_allocated_memory_read`, matching in *,
This completes the proof of `consistent_allocator_read`.
Q.E.D.

C.8.4 Allocation_Info.consistent_allocator_write_TCC1

Terse proof for `consistent_allocator_write_TCC1`.

consistent_allocator_write_TCC1:

$\{1\} \quad \forall (ai: \text{Allocator_Info}, pm: (\text{plain_memory?}), s: \text{State}, ac: (\text{allocator?}[\text{State}](pm)), \\ \text{addr}: \text{Address}, \text{byte}: \text{Byte}): \\ pm' \text{states}(s) \wedge \\ \text{consistent_allocator?}(pm)(ac, ai)(s) \wedge \\ pm' \text{rw_addr}(\text{addr}) \wedge \neg ac' \text{private_mem}(pm)(\text{addr}) \\ \supset \\ \text{OK?}[\text{State}, \text{Unit}](\text{memory_write}(pm' \text{mem})(\text{addr}, \text{byte})(s)) \vee \\ \text{Exception?}[\text{State}, \text{Unit}](\text{memory_write}(pm' \text{mem})(\text{addr}, \text{byte})(s))$
--

Repeatedly Skolemizing and flattening,
 Rewriting using plain_memory_memory_write_ok, matching in *,
 This completes the proof of consistent_allocator_write_TCC1.
 Q.E.D.

C.8.5 Allocation_Info.consistent_allocator_write

Terse proof for consistent_allocator_write.

consistent_allocator_write:

$\{1\} \quad \forall (ai: \text{Allocator_Info}, pm: (\text{plain_memory?}), s: \text{State}, ac: (\text{allocator?}(pm)), \text{addr}: \text{Ad-} \\ \text{dress}, \\ \text{byte}: \text{Byte}): \\ pm' \text{states}(s) \wedge \\ \text{consistent_allocator?}(pm)(ac, ai)(s) \wedge \\ pm' \text{rw_addr}(\text{addr}) \wedge \neg ac' \text{private_mem}(pm)(\text{addr}) \\ \supset \text{consistent_allocator?}(pm)(ac, ai)(\text{state}(\text{memory_write}(pm' \text{mem})(\text{addr}, \text{byte})(s)))$

Repeatedly Skolemizing and flattening,
 Expanding the definition of consistent_allocator?,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 Repeatedly Skolemizing and flattening,
 Instantiating quantified variables,
 Simplifying, rewriting, and recording with decision procedures,
 Rewriting using allocator_allocated_memory_write, matching in *,
 This completes the proof of consistent_allocator_write.
 Q.E.D.

C.8.6 Allocation_Info.consistent_allocator_alloc_TCC1

Terse proof for consistent_allocator_alloc_TCC1.

consistent_allocator_alloc_TCC1:

$\{1\} \quad \forall (ai: \text{Allocator_Info}, pm: (\text{plain_memory?}), s: \text{State}, ac: (\text{allocator?}[\text{State}](pm)), \\ \text{size}: \text{posnat}): \\ pm' \text{states}(s) \wedge \\ \text{consistent_allocator?}(pm)(ac, ai)(s) \wedge \\ \text{OK?}(ac' \text{alloc}(pm)(\text{size})(s)) \wedge \neg \text{data}(ac' \text{alloc}(pm)(\text{size})(s)) = \text{null_address} \\ \supset \\ \text{OK?}[\text{State}, \text{Address}](ac' \text{alloc}(pm)(\text{size})(s)) \vee \\ \text{Exception?}[\text{State}, \text{Address}](ac' \text{alloc}(pm)(\text{size})(s))$

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `consistent_allocator_alloc_TCC1`.
Q.E.D.

C.8.7 Allocation_Info.consistent_allocator_alloc

Terse proof for `consistent_allocator_alloc`.

`consistent_allocator_alloc`:

<pre>{1} ∃ (ai: Allocator_Info, pm: (plain_memory?), s: State, ac: (allocator?(pm)), size: posnat): pm' states(s) ∧ consistent_allocator?(pm)(ac, ai)(s) ∧ OK?(ac' alloc(pm)(size)(s)) ∧ ¬ data(ac' alloc(pm)(size)(s)) = null_address ⊃ consistent_allocator?(pm)(ac, cons((data(ac' alloc(pm)(size)(s)), size), ai)) (state(ac' alloc(pm)(size)(s)))</pre>

Repeatedly Skolemizing and flattening,
Expanding the definition of `consistent_allocator?`,
Applying propositional simplification,
we get 2 subgoals:

`consistent_allocator_alloc.1`:

<pre>{-1} plain_memory?(pm') {-2} allocator?(pm')(ac') {-3} size' > 0 {-4} pm' states(s') {-5} allocator_info?(ai') {-6} ∃ (addr: Address, size: posnat): member((addr, size), ai') ⊃ (address_block(addr, size) ⊆ ac' allocated(pm')(s')) {-7} OK?(ac' alloc(pm')(size')(s'))</pre>
<pre>{1} allocator_info?(cons((data(ac' alloc(pm')(size')(s')), size'), ai')) {2} data(ac' alloc(pm')(size')(s')) = null_address</pre>

Expanding the definition of `allocator_info?`,
Rewriting using `blocks_pairwise_disjoint_add`, matching in *,
Hiding formulas: 2,
Repeatedly Skolemizing and flattening,
Rewriting using `blocks_disjoint_disjoint`, matching in *,
Instantiating the top quantifier in -8 with the terms: `oaddr'`, `osize'`,
Simplifying, rewriting, and recording with decision procedures,
Using lemma `disjoint_mono`,
Installing automatic rewrites from: `subset_equal`
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Using lemma `allocator_alloc_disjoint`,
Simplifying, rewriting, and recording with decision procedures,
Rewriting using `disjoint_symmetric`, matching in *,
This completes the proof of `consistent_allocator_alloc.1`.

consistent_allocator_alloc.2:

{-1}	plain_memory?(pm')
{-2}	allocator?(pm')(ac')
{-3}	size' > 0
{-4}	pm' 'states(s')
{-5}	allocator_info?(ai')
{-6}	\forall (addr: Address, size: posnat): member((addr, size), ai') \supset (address_block(addr, size) \subseteq ac' 'allocated(pm')(s'))
{-7}	OK?(ac' 'alloc(pm')(size')(s'))
{1}	\forall (addr: Address, size: posnat): member((addr, size), cons((data(ac' 'alloc(pm')(size')(s')), size'), ai')) \supset (address_block(addr, size) \subseteq ac' 'allocated(pm')(state(ac' 'alloc(pm')(size')(s'))))
{2}	data(ac' 'alloc(pm')(size')(s')) = null_address

Hiding formulas: -5,

Repeatedly Skolemizing and flattening,

Expanding the definition of member,

Applying propositional simplification,

we get 2 subgoals:

consistent_allocator_alloc.2.1:

{-1}	addr' = data(ac' 'alloc(pm')(size')(s'))
{-2}	size'' = size'
{-3}	size'' > 0
{-4}	plain_memory?(pm')
{-5}	allocator?(pm')(ac')
{-6}	size' > 0
{-7}	pm' 'states(s')
{-8}	\forall (addr: Address, size: posnat): member((addr, size), ai') \supset (address_block(addr, size) \subseteq ac' 'allocated(pm')(s'))
{-9}	OK?(ac' 'alloc(pm')(size')(s'))
{1}	(address_block(addr', size'') \subseteq ac' 'allocated(pm')(state(ac' 'alloc(pm')(size')(s'))))
{2}	data(ac' 'alloc(pm')(size')(s')) = null_address

Rewriting using allocator_allocated_alloc, matching in *,

Replacing using formula -1,

Replacing using formula -2,

Rewriting using union_subset3, matching in *,

This completes the proof of consistent_allocator_alloc.2.1.

consistent_allocator_alloc.2.2:

{-1}	member((addr', size''), ai')
{-2}	size'' > 0
{-3}	plain_memory?(pm')
{-4}	allocator?(pm')(ac')
{-5}	size' > 0
{-6}	pm' 'states(s')
{-7}	\forall (addr: Address, size: posnat): member((addr, size), ai') \supset (address_block(addr, size) \subseteq ac' 'allocated(pm')(s'))
{-8}	OK?(ac' 'alloc(pm')(size')(s'))
{1}	(address_block(addr', size'') \subseteq ac' 'allocated(pm')(state(ac' 'alloc(pm')(size')(s'))))
{2}	data(ac' 'alloc(pm')(size')(s')) = null_address

Instantiating quantified variables,

Simplifying, rewriting, and recording with decision procedures,

Rewriting using allocator_allocated_alloc, matching in *,

Rewriting using subset_bigger_union_left, matching in *,

This completes the proof of consistent_allocator_alloc.2.2.

Q.E.D.

C.8.8 Allocation_Info.consistent_allocator_free_TCC1

Terse proof for consistent_allocator_free_TCC1.

consistent_allocator_free_TCC1:

{1}	\forall (ai: Allocator_Info, pm: (plain_memory?), s: State, ac: (allocator?[State](pm)), addr: Address): OK?(ac' 'free(pm)(addr)(s)) \wedge consistent_allocator?(pm)(ac, ai)(s) \wedge pm' 'states(s) \supset up?[posnat](ac' 'freed_size(pm)(addr)(s))
-----	--

Repeatedly Skolemizing and flattening,

Rewriting using allocator_free_freed_size, matching in *,

This completes the proof of consistent_allocator_free_TCC1.

Q.E.D.

C.8.9 Allocation_Info.consistent_allocator_free_TCC2

Terse proof for consistent_allocator_free_TCC2.

consistent_allocator_free_TCC2:

<pre> {1} ∃ (ai: Allocator_Info, pm: (plain_memory?), s: State, ac: (allocator?[State](pm)), addr: Address): pm' states(s) ∧ consistent_allocator?(pm)(ac, ai)(s) ∧ OK?(ac' free(pm)(addr)(s)) ∧ (member((addr, down(ac' freed_size(pm)(addr)(s))), ai) ∨ blocks_pairwise_disjoint(cons((addr, down(ac' freed_size(pm)(addr)(s))), ai))) ⊃ OK?[State, Unit](ac' free(pm)(addr)(s)) ∨ Exception?[State, Unit](ac' free(pm)(addr)(s)) </pre>

Repeatedly Skolemizing and flattening,

This completes the proof of consistent_allocator_free_TCC2.

Q.E.D.

C.8.10 Allocation_Info.consistent_allocator_free

Terse proof for consistent_allocator_free.

consistent_allocator_free:

<pre> {1} ∃ (ai: Allocator_Info, pm: (plain_memory?), s: State, ac: (allocator?(pm)), addr: Address): pm' states(s) ∧ consistent_allocator?(pm)(ac, ai)(s) ∧ OK?(ac' free(pm)(addr)(s)) ∧ (member((addr, down(ac' freed_size(pm)(addr)(s))), ai) ∨ blocks_pairwise_disjoint(cons((addr, down(ac' freed_size(pm)(addr)(s))), ai))) ⊃ consistent_allocator?(pm) (ac, list_remove((addr, down(ac' freed_size(pm)(addr)(s))), ai) (state(ac' free(pm)(addr)(s))) </pre>
--

Repeatedly Skolemizing and flattening,

Expanding the definition of consistent_allocator?,

Splitting conjunctions,

we get 2 subgoals:

consistent_allocator_free.1:

<div style="display: flex; flex-direction: column; gap: 5px;"> <div style="display: flex; align-items: flex-start;"> {-1} plain_memory?(pm') </div> <div style="display: flex; align-items: flex-start;"> {-2} allocator?(pm')(ac') </div> <div style="display: flex; align-items: flex-start;"> {-3} pm' 'states(s') </div> <div style="display: flex; align-items: flex-start;"> {-4} allocator_info?(ai') \wedge $(\forall (\text{addr}: \text{Address}, \text{size}: \text{posnat}):$ $\text{member}(\text{addr}, \text{size}), \text{ai}') \supset$ $(\text{address_block}(\text{addr}, \text{size}) \subseteq \text{ac}'\text{'allocated}(\text{pm}')(\text{s}'))$ </div> <div style="display: flex; align-items: flex-start;"> {-5} OK?(ac' 'free(pm')(addr')(s')) </div> <div style="display: flex; align-items: flex-start;"> {-6} $(\text{member}(\text{addr}', \text{down}(\text{ac}'\text{'freed_size}(\text{pm}')(\text{addr}')(\text{s}))), \text{ai}') \vee$ $\text{blocks_pairwise_disjoint}(\text{cons}(\text{addr}', \text{down}(\text{ac}'\text{'freed_size}(\text{pm}')(\text{addr}')(\text{s}))),$ $\text{ai}'))$ </div> </div>	<hr style="border: 0.5px solid black; margin-bottom: 5px;"/> <div style="display: flex; align-items: flex-start;"> {1} allocator_info?(list_remove((addr', down(ac' 'freed_size(pm')(addr')(s))), ai')) </div>
---	--

Expanding the definition of allocator_info?,

Rewriting using blocks_pairwise_disjoint_remove, matching in *,

This completes the proof of consistent_allocator_free.1.

consistent_allocator_free.2:

<div style="display: flex; flex-direction: column; gap: 5px;"> <div style="display: flex; align-items: flex-start;"> {-1} plain_memory?(pm') </div> <div style="display: flex; align-items: flex-start;"> {-2} allocator?(pm')(ac') </div> <div style="display: flex; align-items: flex-start;"> {-3} pm' 'states(s') </div> <div style="display: flex; align-items: flex-start;"> {-4} allocator_info?(ai') \wedge $(\forall (\text{addr}: \text{Address}, \text{size}: \text{posnat}):$ $\text{member}(\text{addr}, \text{size}), \text{ai}') \supset$ $(\text{address_block}(\text{addr}, \text{size}) \subseteq \text{ac}'\text{'allocated}(\text{pm}')(\text{s}'))$ </div> <div style="display: flex; align-items: flex-start;"> {-5} OK?(ac' 'free(pm')(addr')(s')) </div> <div style="display: flex; align-items: flex-start;"> {-6} $(\text{member}(\text{addr}', \text{down}(\text{ac}'\text{'freed_size}(\text{pm}')(\text{addr}')(\text{s}))), \text{ai}') \vee$ $\text{blocks_pairwise_disjoint}(\text{cons}(\text{addr}', \text{down}(\text{ac}'\text{'freed_size}(\text{pm}')(\text{addr}')(\text{s}))),$ $\text{ai}'))$ </div> </div>	<hr style="border: 0.5px solid black; margin-bottom: 5px;"/> <div style="display: flex; align-items: flex-start;"> {1} $\forall (\text{addr}: \text{Address}, \text{size}: \text{posnat}):$ $\text{member}(\text{addr}, \text{size}),$ $\text{list_remove}(\text{addr}', \text{down}(\text{ac}'\text{'freed_size}(\text{pm}')(\text{addr}')(\text{s}))), \text{ai}')$ \supset $(\text{address_block}(\text{addr}, \text{size}) \subseteq \text{ac}'\text{'allocated}(\text{pm}')(\text{state}(\text{ac}'\text{'free}(\text{pm}')(\text{addr}')(\text{s}))))$ </div>
---	---

Repeatedly Skolemizing and flattening,

Rewriting using member_list_remove, matching in *,

Splitting conjunctions,

we get 2 subgoals:

consistent_allocator_free.2.1:

{-1}	$size' > 0$
{-2}	$plain_memory?(pm')$
{-3}	$allocator?(pm')(ac')$
{-4}	$pm' \text{'states}(s')$
{-5}	$allocator_info?(ai')$
{-6}	$\forall (addr: Address, size: posnat):$ $member((addr, size), ai') \supset$ $(address_block(addr, size) \subseteq ac' \text{'allocated}(pm')(s'))$
{-7}	$OK?(ac' \text{'free}(pm')(addr')(s'))$
{-8}	$(member((addr', down(ac' \text{'freed_size}(pm')(addr')(s'))), ai') \vee$ $blocks_pairwise_disjoint(cons((addr', down(ac' \text{'freed_size}(pm')(addr')(s'))),$ $ai')))$
{1}	TRUE
{2}	$(address_block(addr'', size') \subseteq ac' \text{'allocated}(pm')(state(ac' \text{'free}(pm')(addr')(s'))))$

which is trivially true.

This completes the proof of consistent_allocator_free.2.1.

consistent_allocator_free.2.2:

{-1}	$\neg (addr'' = addr' \wedge size' = down(ac' \text{'freed_size}(pm')(addr')(s')))$ \wedge $member((addr'', size'), ai')$
{-2}	$size' > 0$
{-3}	$plain_memory?(pm')$
{-4}	$allocator?(pm')(ac')$
{-5}	$pm' \text{'states}(s')$
{-6}	$allocator_info?(ai')$
{-7}	$\forall (addr: Address, size: posnat):$ $member((addr, size), ai') \supset$ $(address_block(addr, size) \subseteq ac' \text{'allocated}(pm')(s'))$
{-8}	$OK?(ac' \text{'free}(pm')(addr')(s'))$
{-9}	$(member((addr', down(ac' \text{'freed_size}(pm')(addr')(s'))), ai') \vee$ $blocks_pairwise_disjoint(cons((addr', down(ac' \text{'freed_size}(pm')(addr')(s'))),$ $ai')))$
{1}	$(address_block(addr'', size') \subseteq ac' \text{'allocated}(pm')(state(ac' \text{'free}(pm')(addr')(s'))))$

Applying disjunctive simplification to flatten sequent,

Instantiating quantified variables,

Simplifying, rewriting, and recording with decision procedures,

Rewriting using allocator_allocated_free, matching in *,

Rewriting using disjoint_subset_difference, matching in *,

Hiding formulas: 3,

Splitting conjunctions,

we get 2 subgoals:

consistent_allocator_free.2.2.1:

{-1}	member((addr', down(ac' freed_size(pm')(addr')(s'))), ai')
{-2}	member((addr'', size'), ai')
{-3}	size' > 0
{-4}	plain_memory?(pm')
{-5}	allocator?(pm')(ac')
{-6}	pm' states(s')
{-7}	allocator_info?(ai')
{-8}	(address_block(addr'', size') \subseteq ac' allocated(pm')(s'))
{-9}	OK?(ac' free(pm')(addr')(s'))
{1}	disjoint?(address_block(addr'', size'), address_block(addr', down(ac' freed_size(pm')(addr')(s'))))
{2}	addr'' = addr' \wedge size' = down(ac' freed_size(pm')(addr')(s'))

Expanding the definition of allocator_info?,

Expanding the definition of blocks_pairwise_disjoint,

Instantiating the top quantifier in -7 with the terms: addr'', addr', size', down(ac' freed_size(pm')(addr')(s')),

Simplifying, rewriting, and recording with decision procedures,

Splitting conjunctions,

we get 2 subgoals:

consistent_allocator_free.2.2.1.1:

{-1}	blocks_disjoint?(addr'', size', addr', down(ac' freed_size(pm')(addr')(s')))
{-2}	member((addr', down(ac' freed_size(pm')(addr')(s'))), ai')
{-3}	member((addr'', size'), ai')
{-4}	size' > 0
{-5}	plain_memory?(pm')
{-6}	allocator?(pm')(ac')
{-7}	pm' states(s')
{-8}	(address_block(addr'', size') \subseteq ac' allocated(pm')(s'))
{-9}	OK?(ac' free(pm')(addr')(s'))
{1}	disjoint?(address_block(addr'', size'), address_block(addr', down(ac' freed_size(pm')(addr')(s'))))
{2}	addr'' = addr' \wedge size' = down(ac' freed_size(pm')(addr')(s'))

Rewriting using blocks_disjoint_disjoint, matching in *,

This completes the proof of consistent_allocator_free.2.2.1.1.

consistent_allocator_free.2.2.1.2:

{-1}	member((addr', down(ac' freed_size(pm')(addr')(s'))), ai')
{-2}	member((addr'', size'), ai')
{-3}	size' > 0
{-4}	plain_memory?(pm')
{-5}	allocator?(pm')(ac')
{-6}	pm' states(s')
{-7}	(address_block(addr'', size') \subseteq ac' allocated(pm')(s'))
{-8}	OK?(ac' free(pm')(addr')(s'))
{1}	member((addr', down(ac' freed_size(pm')(addr')(s'))), list_remove((addr'', size'), ai'))
{2}	disjoint?(address_block(addr'', size'), address_block(addr', down(ac' freed_size(pm')(addr')(s'))))
{3}	addr'' = addr' \wedge size' = down(ac' freed_size(pm')(addr')(s'))

Rewriting using member_list_remove, matching in *,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `consistent_allocator_free.2.2.1.2`.

`consistent_allocator_free.2.2.2`:

{-1}	<code>blocks_pairwise_disjoint(cons((addr', down(ac' freed_size(pm')(addr')(s'))), ai'))</code>
{-2}	<code>member((addr'', size'), ai')</code>
{-3}	<code>size' > 0</code>
{-4}	<code>plain_memory?(pm')</code>
{-5}	<code>allocator?(pm')(ac')</code>
{-6}	<code>pm' states(s')</code>
{-7}	<code>allocator_info?(ai')</code>
{-8}	<code>(address_block(addr'', size') ⊆ ac' allocated(pm')(s'))</code>
{-9}	<code>OK?(ac' free(pm')(addr')(s'))</code>
{1}	<code>disjoint?(address_block(addr'', size'), address_block(addr', down(ac' freed_size(pm')(addr')(s'))))</code>
{2}	<code>addr'' = addr' ∧ size' = down(ac' freed_size(pm')(addr')(s'))</code>

Expanding the definition of `blocks_pairwise_disjoint`,

Instantiating the top quantifier in -1 with the terms: `addr'`, `addr''`, `down(ac' freed_size(pm')(addr')(s'))`, `size'`,

Splitting conjunctions,

we get 3 subgoals:

`consistent_allocator_free.2.2.2.1`:

{-1}	<code>blocks_disjoint?(addr', down(ac' freed_size(pm')(addr')(s')), addr'', size')</code>
{-2}	<code>member((addr'', size'), ai')</code>
{-3}	<code>size' > 0</code>
{-4}	<code>plain_memory?(pm')</code>
{-5}	<code>allocator?(pm')(ac')</code>
{-6}	<code>pm' states(s')</code>
{-7}	<code>allocator_info?(ai')</code>
{-8}	<code>(address_block(addr'', size') ⊆ ac' allocated(pm')(s'))</code>
{-9}	<code>OK?(ac' free(pm')(addr')(s'))</code>
{1}	<code>disjoint?(address_block(addr'', size'), address_block(addr', down(ac' freed_size(pm')(addr')(s'))))</code>
{2}	<code>addr'' = addr' ∧ size' = down(ac' freed_size(pm')(addr')(s'))</code>

Rewriting using `blocks_disjoint_disjoint`, matching in *

Rewriting using `disjoint_symmetric`, matching in *

This completes the proof of `consistent_allocator_free.2.2.2.1`.

`consistent_allocator_free.2.2.2.2`:

{-1}	<code>member((addr'', size'), ai')</code>
{-2}	<code>size' > 0</code>
{-3}	<code>plain_memory?(pm')</code>
{-4}	<code>allocator?(pm')(ac')</code>
{-5}	<code>pm' states(s')</code>
{-6}	<code>allocator_info?(ai')</code>
{-7}	<code>(address_block(addr'', size') ⊆ ac' allocated(pm')(s'))</code>
{-8}	<code>OK?(ac' free(pm')(addr')(s'))</code>
{1}	<code>member((addr', down(ac' freed_size(pm')(addr')(s'))), cons((addr', down(ac' freed_size(pm')(addr')(s'))), ai'))</code>
{2}	<code>disjoint?(address_block(addr'', size'), address_block(addr', down(ac' freed_size(pm')(addr')(s'))))</code>
{3}	<code>addr'' = addr' ∧ size' = down(ac' freed_size(pm')(addr')(s'))</code>

Expanding the definition of member,
which is trivially true.

This completes the proof of `consistent_allocator_free.2.2.2.2`.
`consistent_allocator_free.2.2.2.3`:

{-1}	member((addr'', size'), ai')
{-2}	size' > 0
{-3}	plain_memory?(pm')
{-4}	allocator?(pm')(ac')
{-5}	pm' 'states(s')
{-6}	allocator_info?(ai')
{-7}	(address_block(addr'', size') \subseteq ac' 'allocated(pm')(s'))
{-8}	OK?(ac' 'free(pm')(addr')(s'))
{1}	
	member((addr'', size'), list_remove((addr', down(ac' 'freed_size(pm')(addr')(s'))), cons((addr', down(ac' 'freed_size(pm')(addr')(s'))), ai')))
{2}	disjoint?(address_block(addr'', size'), address_block(addr', down(ac' 'freed_size(pm')(addr')(s'))))
{3}	addr'' = addr' \wedge size' = down(ac' 'freed_size(pm')(addr')(s'))

Rewriting using member_list_remove, matching in *,

Expanding the definition of member,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `consistent_allocator_free.2.2.2.3`.

Q.E.D.

C.9 Proofs for Allocation_Table (allocators.pvs)

C.9.1 Allocation_Table.allocation_finite_tree

Terse proof for `allocation_finite_tree`.

`allocation_finite_tree`:

{1}	\forall (pm: (plain_memory?), at: Allocation_Table(pm)): finite_tree?(at 'hierarchy)
-----	--

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `allocation_finite_tree`.

Q.E.D.

C.9.2 Allocation_Table.allocation_subnodes_allocated_and_free

Terse proof for `allocation_subnodes_allocated_and_free`.

`allocation_subnodes_allocated_and_free`:

{1}	\forall (pm: (plain_memory?), s: State, at: Allocation_Table(pm), ac1, ac2: (at 'hierarchy' nodes)): allocation_ok?(pm)(at, s) \wedge at 'hierarchy' edges(ac1, ac2) \supset (ac2' memory_pool(pm)(s) \subseteq ac1' allocated(pm)(s)) \wedge block_is_free(ac2' memory_pool(pm)(s), at 'info(ac1))
-----	---

Repeatedly Skolemizing and flattening,

Expanding the definition of `allocation_ok?`,

Applying disjunctive simplification to flatten sequent,
 Instantiating quantified variables,
 Simplifying, rewriting, and recording with decision procedures,
 This completes the proof of `allocation_subnodes_allocated_and_free`.
 Q.E.D.

C.9.3 Allocation_Table.allocation_subnodes_disjoint

Terse proof for `allocation_subnodes_disjoint`.

`allocation_subnodes_disjoint`:

$$\{1\} \quad \forall (pm: (\text{plain_memory?}), s: \text{State}, at: \text{Allocation_Table}(pm), \\ ac1, ac2, ac3: (\text{at}'\text{hierarchy}'\text{nodes})): \\ \text{allocation_ok?}(pm)(at, s) \wedge \\ \text{at}'\text{hierarchy}'\text{edges}(ac1, ac2) \wedge \text{at}'\text{hierarchy}'\text{edges}(ac1, ac3) \\ \supset ac2 = ac3 \vee \text{disjoint?}(ac2'\text{memory_pool}(pm)(s), ac3'\text{memory_pool}(pm)(s))$$

Repeatedly Skolemizing and flattening,
 Expanding the definition of `allocation_ok?`,
 Applying disjunctive simplification to flatten sequent,
 Instantiating the top quantifier in -9 with the terms: `ac1'`, `ac2'`, `ac3'`,
 Simplifying, rewriting, and recording with decision procedures,
 This completes the proof of `allocation_subnodes_disjoint`.
 Q.E.D.

C.9.4 Allocation_Table.allocation_roots_disjoint

Terse proof for `allocation_roots_disjoint`.

`allocation_roots_disjoint`:

$$\{1\} \quad \forall (pm: (\text{plain_memory?}), s: \text{State}, at: \text{Allocation_Table}(pm), \\ ac1, ac2: (\text{roots}(at'\text{hierarchy}))) \\ \text{allocation_ok?}(pm)(at, s) \supset \\ ac1 = ac2 \vee \text{disjoint?}(ac1'\text{memory_pool}(pm)(s), ac2'\text{memory_pool}(pm)(s))$$

Repeatedly Skolemizing and flattening,
 Expanding the definition of `allocation_ok?`,
 Applying disjunctive simplification to flatten sequent,
 Instantiating quantified variables,
 Simplifying, rewriting, and recording with decision procedures,
 This completes the proof of `allocation_roots_disjoint`.
 Q.E.D.

C.9.5 Allocation_Table.allocation_private_memory_disjoint

Terse proof for `allocation_private_memory_disjoint`.

`allocation_private_memory_disjoint`:

$$\{1\} \quad \forall (pm: (\text{plain_memory?}), s: \text{State}, at: \text{Allocation_Table}(pm), \\ ac1, ac2: (\text{at}'\text{hierarchy}'\text{nodes})): \\ \text{allocation_ok?}(pm)(at, s) \supset \\ ac1 = ac2 \vee \text{disjoint?}(ac1'\text{private_mem}(pm), ac2'\text{private_mem}(pm))$$

Repeatedly Skolemizing and flattening,
 Expanding the definition of `allocation_ok?`,
 Applying disjunctive simplification to flatten sequent,
 Instantiating quantified variables,
 Simplifying, rewriting, and recording with decision procedures,
 This completes the proof of `allocation_private_memory_disjoint`.
 Q.E.D.

C.9.6 Allocation_Table.allocation_consistent_allocator

Terse proof for `allocation_consistent_allocator`.

`allocation_consistent_allocator`:

$\{1\} \quad \forall (pm: (plain_memory?), s: State, at: Allocation_Table(pm), ac: (at' hierarchy' nodes)):$ $allocation_ok?(pm)(at, s) \supset consistent_allocator?(pm)(ac, at' info(ac))(s)$

Repeatedly Skolemizing and flattening,
 Expanding the definition of `allocation_ok?`,
 Applying disjunctive simplification to flatten sequent,
 Instantiating quantified variables,
 This completes the proof of `allocation_consistent_allocator`.
 Q.E.D.

C.9.7 Allocation_Table.allocation_allocated_allocation_point

Terse proof for `allocation_allocated_allocation_point`.

`allocation_allocated_allocation_point`:

$\{1\} \quad \forall (pm: (plain_memory?), s: State, at: Allocation_Table(pm), ac: (at' hierarchy' nodes),$ $addr: Address, size: posnat):$ $allocation_ok?(pm)(at, s) \wedge member((addr, size), at' info(ac)) \supset$ $(address_block(addr, size) \subseteq ac' allocated(pm)(s))$
--

Repeatedly Skolemizing and flattening,
 Using lemma `allocation_consistent_allocator`,
 Simplifying, rewriting, and recording with decision procedures,
 Expanding the definition of `consistent_allocator?`,
 Applying disjunctive simplification to flatten sequent,
 Instantiating quantified variables,
 Simplifying, rewriting, and recording with decision procedures,
 This completes the proof of `allocation_allocated_allocation_point`.
 Q.E.D.

C.9.8 Allocation_Table.allocation_allocation_point_in_memory_pool

Terse proof for `allocation_allocation_point_in_memory_pool`.

`allocation_allocation_point_in_memory_pool`:

$\{1\} \quad \forall (pm: (plain_memory?), s: State, at: Allocation_Table(pm), ac: (at' hierarchy' nodes),$ $addr: Address, size: posnat):$ $pm' states(s) \wedge allocation_ok?(pm)(at, s) \wedge member((addr, size), at' info(ac)) \supset$ $(address_block(addr, size) \subseteq ac' memory_pool(pm)(s))$
--

Repeatedly Skolemizing and flattening,
 Using lemma allocation_allocated_allocation_point,
 Simplifying, rewriting, and recording with decision procedures,
 Using lemma allocator_subset_allocated_memory_pool,
 Simplifying, rewriting, and recording with decision procedures,
 Forward chaining on subset_transitive,
 This completes the proof of allocation_allocation_point_in_memory_pool.
 Q.E.D.

C.9.9 Allocation_Table.allocators_different_in_hierachy

Terse proof for allocators_different_in_hierachy.

allocators_different_in_hierachy:

$$\frac{}{\{1\} \quad \forall (\text{pm}: (\text{plain_memory?}), \text{at}: \text{Allocation_Table}(\text{pm}), \text{ac1}, \text{ac2}: (\text{at}'\text{hierarchy}'\text{nodes})): \text{at}'\text{hierarchy}'\text{edges}(\text{ac1}, \text{ac2}) \supset \neg \text{ac1} = \text{ac2}}$$

Repeatedly Skolemizing and flattening,
 Using lemma different_nodes_on_edge[(allocator?(pm'))],
 Simplifying, rewriting, and recording with decision procedures,
 Adding type constraints for at!1'hierarchy,
 Expanding the definition of finite_tree?,
 Expanding the definition of tree?,
 which is trivially true.
 This completes the proof of allocators_different_in_hierachy.
 Q.E.D.

C.9.10 Allocation_Table.uppath_memory_pool

Terse proof for uppath_memory_pool.

uppath_memory_pool:

$$\frac{}{\{1\} \quad \forall (\text{pm}: (\text{plain_memory?}), s: \text{State}, \text{at}: \text{Allocation_Table}(\text{pm}), p: (\text{path?}(\text{at}'\text{hierarchy}))) : \text{allocation_ok?}(\text{pm})(\text{at}, s) \wedge \text{pm}'\text{states}(s) \supset (\text{path_end}(p)'\text{memory_pool}(\text{pm})(s) \subseteq \text{path_start}(p)'\text{memory_pool}(\text{pm})(s))}$$

For the top quantifier in 1, we introduce Skolem constants: (pm' s' at' _),
 Inducting on p on formula 1 using induction scheme path_list_induction[(allocator?(pm'))],
 we get 3 subgoals:

uppath_memory_pool.1:

$$\frac{\begin{array}{l} \{-1\} \quad \text{allocation_ok?}(\text{pm}')(\text{at}', s') \\ \{-2\} \quad \text{pm}'\text{states}(s') \end{array}}{\begin{array}{l} \{1\} \quad \text{path?}(\text{at}'\text{hierarchy})(p') \\ \{2\} \quad (\text{path_end}(p')'\text{memory_pool}(\text{pm}')'(s') \subseteq \text{path_start}(p')'\text{memory_pool}(\text{pm}')'(s')) \end{array}}$$

Simplifying, rewriting, and recording with decision procedures,
 This completes the proof of uppath_memory_pool.1.

uppath_memory_pool.2:

$\{1\} \quad \forall (t: ((\text{allocator?}(\text{pm}')))):$ $\text{path?}(\text{at}'\text{hierarchy})(\text{cons}(t, \text{null})) \supset$ $\text{allocation_ok?}(\text{pm}')(\text{at}', s') \wedge \text{pm}'\text{states}(s') \supset$ $(\text{path_end}(\text{cons}(t, \text{null}))'\text{memory_pool}(\text{pm}')(\text{s}') \subseteq \text{path_start}(\text{cons}(t, \text{null}))'\text{memory_pool}(\text{pm}'))$
--

Repeatedly Skolemizing and flattening,

Installing automatic rewrites from: path_start path_end length nth subset_reflexive

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of uppath_memory_pool.2.

uppath_memory_pool.3:

$\{1\} \quad \forall (t: ((\text{allocator?}(\text{pm}'))), \text{tail}: (\text{path_list?})):$ $(\text{path?}(\text{at}'\text{hierarchy})(\text{tail}) \supset$ $\text{allocation_ok?}(\text{pm}')(\text{at}', s') \wedge \text{pm}'\text{states}(s') \supset$ $(\text{path_end}(\text{tail})'\text{memory_pool}(\text{pm}')(\text{s}') \subseteq \text{path_start}(\text{tail})'\text{memory_pool}(\text{pm}')(\text{s}'))$ \supset $\text{path?}(\text{at}'\text{hierarchy})(\text{cons}(t, \text{tail})) \supset$ $\text{allocation_ok?}(\text{pm}')(\text{at}', s') \wedge \text{pm}'\text{states}(s') \supset$ $(\text{path_end}(\text{cons}(t, \text{tail}))'\text{memory_pool}(\text{pm}')(\text{s}') \subseteq \text{path_start}(\text{cons}(t, \text{tail}))'\text{memory_pool}(\text{pm}'))$
--

Repeatedly Skolemizing and flattening,

Using lemma path_cdr,

Simplifying, rewriting, and recording with decision procedures,

Installing automatic rewrites from: path_start nth length path_list? path_end

Simplifying, rewriting, and recording with decision procedures,

Expanding the definition of path?,

Applying disjunctive simplification to flatten sequent,

Instantiating the top quantifier in -7 with the terms: 0,

Simplifying, rewriting, and recording with decision procedures,

Using lemma allocation_subnodes_allocated_and_free,

Simplifying, rewriting, and recording with decision procedures,

Applying disjunctive simplification to flatten sequent,

Hiding formulas: -2,

Forward chaining on subset_transitive,

Hiding formulas: -2, -7,

Using lemma allocator_subset_allocated_memory_pool,

Simplifying, rewriting, and recording with decision procedures,

Forward chaining on subset_transitive,

This completes the proof of uppath_memory_pool.3.

Q.E.D.

C.9.11

Allocation_Table.uppath_memory_pool_allocated_and_free_TCC1

Terse proof for uppath_memory_pool_allocated_and_free_TCC1.

uppath_memory_pool_allocated_and_free_TCC1:

$\{1\} \quad \forall (pm: (\text{plain_memory?}), s: \text{State}, at: \text{Allocation_Table}(pm), \\ p: (\text{path?}[\text{((allocator?(pm)))]}(\text{at 'hierarchy'}))): \\ (\text{path_end}(p)\text{'memory_pool}(pm)(s) \subseteq \text{path_start}(p)\text{'allocated}(pm)(s)) \wedge \\ pm\text{'states}(s) \wedge \text{allocation_ok?}(pm)(at, s) \wedge \text{length}(p) \geq 2 \\ \supset \text{at 'hierarchy' nodes}(\text{path_start}[\text{((allocator?(pm)))]}(p))$
--

Repeatedly Skolemizing and flattening,
 Rewriting using nodes_path_start, matching in *,
 This completes the proof of uppath_memory_pool_allocated_and_free_TCC1.
 Q.E.D.

C.9.12 Allocation_Table.uppath_memory_pool_allocated_and_free

Terse proof for uppath_memory_pool_allocated_and_free.

uppath_memory_pool_allocated_and_free:

$\{1\} \quad \forall (pm: (\text{plain_memory?}), s: \text{State}, at: \text{Allocation_Table}(pm), p: (\text{path?}(\text{at 'hierarchy'}))): \\ pm\text{'states}(s) \wedge \text{allocation_ok?}(pm)(at, s) \wedge \text{length}(p) \geq 2 \supset \\ (\text{path_end}(p)\text{'memory_pool}(pm)(s) \subseteq \text{path_start}(p)\text{'allocated}(pm)(s)) \wedge \\ \text{block_is_free}(\text{path_end}(p)\text{'memory_pool}(pm)(s), \text{at 'info}(\text{path_start}(p)))$

Repeatedly Skolemizing and flattening,
 Using lemma path_cdr,
 Installing automatic rewrites from: path_list? length_cdr path_end path_start nth subset_reflexive
 Simplifying, rewriting, and recording with decision procedures,
 Using lemma uppath_memory_pool,
 Simplifying, rewriting, and recording with decision procedures,
 Expanding the definition of path?,
 Applying disjunctive simplification to flatten sequent,
 Instantiating the top quantifier in -7 with the terms: 0,
 Simplifying, rewriting, and recording with decision procedures,
 Using lemma allocation_subnodes_allocated_and_free,
 Simplifying, rewriting, and recording with decision procedures,
 Applying disjunctive simplification to flatten sequent,
 Forward chaining on subset_transitive,
 Simplifying, rewriting, and recording with decision procedures,
 Hiding formulas: -1, -2,
 Expanding the definition of block_is_free,
 Repeatedly Skolemizing and flattening,
 Instantiating quantified variables,
 Simplifying, rewriting, and recording with decision procedures,
 Using lemma disjoint_mono,
 Simplifying, rewriting, and recording with decision procedures,
 This completes the proof of uppath_memory_pool_allocated_and_free.
 Q.E.D.

C.9.13 Allocation_Table.allocation_memory_pools

Terse proof for allocation_memory_pools.

C Proof scripts

allocation_memory_pools:

<pre> {1} ∀ (pm: (plain_memory?), s: State, at: Allocation_Table(pm), ac1, ac2: (at'hierarchy'nodes)): pm'states(s) ∧ allocation_ok?(pm)(at, s) ⊃ ac1 = ac2 ∨ disjoint?(ac1'memory_pool(pm)(s), ac2'memory_pool(pm)(s)) ∨ ((ac1'memory_pool(pm)(s) ⊆ ac2'allocated(pm)(s)) ∧ block_is_free(ac1'memory_pool(pm)(s), at'info(ac2))) ∨ ((ac2'memory_pool(pm)(s) ⊆ ac1'allocated(pm)(s)) ∧ block_is_free(ac2'memory_pool(pm)(s), at'info(ac1)))) </pre>
--

Repeatedly Skolemizing and flattening,

Using lemma symmetric_node_pair_distinction[(allocator?(pm'))],

Splitting conjunctions,

we get 7 subgoals:

allocation_memory_pools.1:

<pre> {-1} ∀ (n1_1, n2_1: (at'hierarchy'nodes)): n1_1 = n2_1 ∨ disjoint?(n1_1'memory_pool(pm')(s'), n2_1'memory_pool(pm')(s')) ∨ ((n1_1'memory_pool(pm')(s') ⊆ n2_1'allocated(pm')(s')) ∧ block_is_free(n1_1'memory_pool(pm')(s'), at'info(n2_1))) ∨ ((n2_1'memory_pool(pm')(s') ⊆ n1_1'allocated(pm')(s')) ∧ block_is_free(n2_1'memory_pool(pm')(s'), at'info(n1_1))) </pre>
<pre> {-2} plain_memory?(pm') {-3} allocator?(pm')(ac1') {-4} at'hierarchy'nodes(ac1') {-5} allocator?(pm')(ac2') {-6} at'hierarchy'nodes(ac2') {-7} pm'states(s') {-8} allocation_ok?(pm')(at', s') </pre>
<pre> {1} ac1' = ac2' {2} disjoint?(ac1'memory_pool(pm')(s'), ac2'memory_pool(pm')(s')) {3} ((ac1'memory_pool(pm')(s') ⊆ ac2'allocated(pm')(s')) ∧ block_is_free(ac1'memory_pool(pm')(s'), at'info(ac2'))) {4} ((ac2'memory_pool(pm')(s') ⊆ ac1'allocated(pm')(s')) ∧ block_is_free(ac2'memory_pool(pm')(s'), at'info(ac1'))) </pre>

Instantiating quantified variables,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of allocation_memory_pools.1.

allocation_memory_pools.2:

{-1}	plain_memory?(pm')
{-2}	allocator?(pm')(ac1')
{-3}	at' 'hierarchy' nodes(ac1')
{-4}	allocator?(pm')(ac2')
{-5}	at' 'hierarchy' nodes(ac2')
{-6}	pm' 'states(s')
{-7}	allocation_ok?(pm')(at', s')
{1}	TRUE
{2}	ac1' = ac2'
{3}	disjoint?(ac1' 'memory_pool(pm')(s'), ac2' 'memory_pool(pm')(s'))
{4}	((ac1' 'memory_pool(pm')(s') \subseteq ac2' 'allocated(pm')(s')) \wedge block_is_free(ac1' 'memory_pool(pm')(s'), at' 'info(ac2')))
{5}	((ac2' 'memory_pool(pm')(s') \subseteq ac1' 'allocated(pm')(s')) \wedge block_is_free(ac2' 'memory_pool(pm')(s'), at' 'info(ac1')))

which is trivially true.

This completes the proof of allocation_memory_pools.2.

allocation_memory_pools.3:

{-1}	plain_memory?(pm')
{-2}	allocator?(pm')(ac1')
{-3}	at' 'hierarchy' nodes(ac1')
{-4}	allocator?(pm')(ac2')
{-5}	at' 'hierarchy' nodes(ac2')
{-6}	pm' 'states(s')
{-7}	allocation_ok?(pm')(at', s')
{1}	symmetric?($\lambda (n_1, n_2: (at' 'hierarchy' nodes))$): $n_1 = n_2 \vee$ disjoint?(n_1 'memory_pool(pm')(s'), n_2 'memory_pool(pm')(s')) \vee ((n_1 'memory_pool(pm')(s') \subseteq n_2 'allocated(pm')(s')) \wedge block_is_free(n_1 'memory_pool(pm')(s'), at' 'info(n_2))) \vee ((n_2 'memory_pool(pm')(s') \subseteq n_1 'allocated(pm')(s')) \wedge block_is_free(n_2 'memory_pool(pm')(s'), at' 'info(n_1))))
{2}	ac1' = ac2'
{3}	disjoint?(ac1' 'memory_pool(pm')(s'), ac2' 'memory_pool(pm')(s'))
{4}	((ac1' 'memory_pool(pm')(s') \subseteq ac2' 'allocated(pm')(s')) \wedge block_is_free(ac1' 'memory_pool(pm')(s'), at' 'info(ac2')))
{5}	((ac2' 'memory_pool(pm')(s') \subseteq ac1' 'allocated(pm')(s')) \wedge block_is_free(ac2' 'memory_pool(pm')(s'), at' 'info(ac1')))

Keeping 1 and hiding *,

Expanding the definition of symmetric?,

Repeatedly Skolemizing and flattening,

Using lemma disjoint_symmetric,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of allocation_memory_pools.3.

allocation_memory_pools.4:

{-1}	plain_memory?(pm')
{-2}	allocator?(pm')(ac1')
{-3}	at' hierarchy' nodes(ac1')
{-4}	allocator?(pm')(ac2')
{-5}	at' hierarchy' nodes(ac2')
{-6}	pm' states(s')
{-7}	allocation_ok?(pm')(at', s')
{1}	$\forall (n: (\text{at}' \text{ hierarchy}' \text{ nodes})): \text{TRUE}$
{2}	ac1' = ac2'
{3}	disjoint?(ac1' memory_pool(pm')(s'), ac2' memory_pool(pm')(s'))
{4}	$((\text{ac1}' \text{ memory_pool}(\text{pm}')(\text{s}') \subseteq \text{ac2}' \text{ allocated}(\text{pm}')(\text{s}')) \wedge$ $\text{block_is_free}(\text{ac1}' \text{ memory_pool}(\text{pm}')(\text{s}'), \text{at}' \text{ info}(\text{ac2}'))$
{5}	$((\text{ac2}' \text{ memory_pool}(\text{pm}')(\text{s}') \subseteq \text{ac1}' \text{ allocated}(\text{pm}')(\text{s}')) \wedge$ $\text{block_is_free}(\text{ac2}' \text{ memory_pool}(\text{pm}')(\text{s}'), \text{at}' \text{ info}(\text{ac1}'))$

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of allocation_memory_pools.4.

allocation_memory_pools.5:

{-1}	plain_memory?(pm')
{-2}	allocator?(pm')(ac1')
{-3}	at' hierarchy' nodes(ac1')
{-4}	allocator?(pm')(ac2')
{-5}	at' hierarchy' nodes(ac2')
{-6}	pm' states(s')
{-7}	allocation_ok?(pm')(at', s')
{1}	$\forall (n1_1, n2_1: (\text{at}' \text{ hierarchy}' \text{ nodes}), p: (\text{path}'(\text{at}' \text{ hierarchy}'))):$ $\neg n1_1 = n2_1 \wedge \text{path_start}(p) = n2_1 \wedge \text{path_end}(p) = n1_1 \wedge \text{length}(p) \geq 2 \supset$ $n1_1 = n2_1 \vee$ $\text{disjoint?}(n1_1 \text{ memory_pool}(\text{pm}')(\text{s}'), n2_1 \text{ memory_pool}(\text{pm}')(\text{s}')) \vee$ $((n1_1 \text{ memory_pool}(\text{pm}')(\text{s}') \subseteq n2_1 \text{ allocated}(\text{pm}')(\text{s}')) \wedge$ $\text{block_is_free}(n1_1 \text{ memory_pool}(\text{pm}')(\text{s}'), \text{at}' \text{ info}(n2_1))$ \vee $((n2_1 \text{ memory_pool}(\text{pm}')(\text{s}') \subseteq n1_1 \text{ allocated}(\text{pm}')(\text{s}')) \wedge$ $\text{block_is_free}(n2_1 \text{ memory_pool}(\text{pm}')(\text{s}'), \text{at}' \text{ info}(n1_1))$
{2}	ac1' = ac2'
{3}	disjoint?(ac1' memory_pool(pm')(s'), ac2' memory_pool(pm')(s'))
{4}	$((\text{ac1}' \text{ memory_pool}(\text{pm}')(\text{s}') \subseteq \text{ac2}' \text{ allocated}(\text{pm}')(\text{s}')) \wedge$ $\text{block_is_free}(\text{ac1}' \text{ memory_pool}(\text{pm}')(\text{s}'), \text{at}' \text{ info}(\text{ac2}'))$
{5}	$((\text{ac2}' \text{ memory_pool}(\text{pm}')(\text{s}') \subseteq \text{ac1}' \text{ allocated}(\text{pm}')(\text{s}')) \wedge$ $\text{block_is_free}(\text{ac2}' \text{ memory_pool}(\text{pm}')(\text{s}'), \text{at}' \text{ info}(\text{ac1}'))$

Keeping (-6 -7 1) and hiding *,

Repeatedly Skolemizing and flattening,

Using lemma uppath_memory_pool_allocated_and_free,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of allocation_memory_pools.5.

allocation_memory_pools.6:

{-1}	plain_memory?(pm')
{-2}	allocator?(pm')(ac1')
{-3}	at' 'hierarchy' nodes(ac1')
{-4}	allocator?(pm')(ac2')
{-5}	at' 'hierarchy' nodes(ac2')
{-6}	pm' 'states'(s')
{-7}	allocation_ok?(pm')(at', s')
{1}	$\forall (n1_1, n2_1: (at' 'hierarchy' nodes), p_1, p_2: (path?(at' 'hierarchy'))):$ $\neg n1_1 = n2_1 \wedge$ $\text{root_path}(at' 'hierarchy, n1_1)(p_1) \wedge$ $\text{root_path}(at' 'hierarchy, n2_1)(p_2) \wedge \neg \text{path_start}(p_1) = \text{path_start}(p_2)$ \supset $n1_1 = n2_1 \vee$ $\text{disjoint?}(n1_1' \text{memory_pool}(pm')(s'), n2_1' \text{memory_pool}(pm')(s')) \vee$ $((n1_1' \text{memory_pool}(pm')(s') \subseteq n2_1' \text{allocated}(pm')(s')) \wedge$ $\text{block_is_free}(n1_1' \text{memory_pool}(pm')(s'), at' 'info'(n2_1)))$ \vee $((n2_1' \text{memory_pool}(pm')(s') \subseteq n1_1' \text{allocated}(pm')(s')) \wedge$ $\text{block_is_free}(n2_1' \text{memory_pool}(pm')(s'), at' 'info'(n1_1)))$
{2}	ac1' = ac2'
{3}	disjoint?(ac1' 'memory_pool'(pm')(s'), ac2' 'memory_pool'(pm')(s'))
{4}	$((ac1' \text{memory_pool}(pm')(s') \subseteq ac2' \text{allocated}(pm')(s')) \wedge$ $\text{block_is_free}(ac1' \text{memory_pool}(pm')(s'), at' 'info'(ac2')))$
{5}	$((ac2' \text{memory_pool}(pm')(s') \subseteq ac1' \text{allocated}(pm')(s')) \wedge$ $\text{block_is_free}(ac2' \text{memory_pool}(pm')(s'), at' 'info'(ac1')))$

Keeping (-6 -7 1) and hiding *,

Repeatedly Skolemizing and flattening,

Hiding formulas: 5, 6,

Using lemma uppath_memory_pool,

Using lemma uppath_memory_pool,

Simplifying, rewriting, and recording with decision procedures,

Expanding the definition of root_path,

Applying disjunctive simplification to flatten sequent,

Using lemma allocation_roots_disjoint,

Simplifying, rewriting, and recording with decision procedures,

Forward chaining on disjoint_mono,

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of allocation_memory_pools.6.

allocation_memory_pools.7:

<pre> {-1} plain_memory?(pm') {-2} allocator?(pm')(ac1') {-3} at' 'hierarchy' nodes(ac1') {-4} allocator?(pm')(ac2') {-5} at' 'hierarchy' nodes(ac2') {-6} pm' 'states'(s') {-7} allocation_ok?(pm')(at', s') </pre>	<pre> {1} $\forall (n1_1, n2_1, n3: (at' 'hierarchy' nodes), p1, p2: (path?(at' 'hierarchy'))):$ $\neg n1_1 = n2_1 \wedge$ $\neg n1_1 = n3 \wedge$ $\neg n2_1 = n3 \wedge$ $at' 'hierarchy' edges(n3, path_start(p1)) \wedge$ $at' 'hierarchy' edges(n3, path_start(p2)) \wedge$ $path_end(p1) = n1_1 \wedge path_end(p2) = n2_1 \wedge \neg path_start(p1) = path_start(p2)$ \supset $n1_1 = n2_1 \vee$ $disjoint?(n1_1' memory_pool(pm')(s'), n2_1' memory_pool(pm')(s')) \vee$ $((n1_1' memory_pool(pm')(s') \subseteq n2_1' allocated(pm')(s')) \wedge$ $block_is_free(n1_1' memory_pool(pm')(s'), at' 'info(n2_1)))$ \vee $((n2_1' memory_pool(pm')(s') \subseteq n1_1' allocated(pm')(s')) \wedge$ $block_is_free(n2_1' memory_pool(pm')(s'), at' 'info(n1_1)))$ {2} ac1' = ac2' {3} disjoint?(ac1' 'memory_pool(pm')(s'), ac2' 'memory_pool(pm')(s')) {4} $((ac1' 'memory_pool(pm')(s') \subseteq ac2' 'allocated(pm')(s')) \wedge$ $block_is_free(ac1' 'memory_pool(pm')(s'), at' 'info(ac2'))$ {5} $((ac2' 'memory_pool(pm')(s') \subseteq ac1' 'allocated(pm')(s')) \wedge$ $block_is_free(ac2' 'memory_pool(pm')(s'), at' 'info(ac1'))$ </pre>
--	---

Keeping (-6 -7 1) and hiding *,
 Repeatedly Skolemizing and flattening,
 Hiding formulas: 7, 8,
 Using lemma uppath_memory_pool,
 Using lemma uppath_memory_pool,
 Simplifying, rewriting, and recording with decision procedures,
 Using lemma allocation_subnodes_disjoint,
 Simplifying, rewriting, and recording with decision procedures,
 Forward chaining on disjoint_mono,
 Simplifying, rewriting, and recording with decision procedures,
 This completes the proof of allocation_memory_pools.7.
 Q.E.D.

C.9.14 Allocation_Table.allocation_info_TCC1

Terse proof for allocation_info_TCC1.

allocation_info_TCC1:

<pre> {1} $\forall (pm: (plain_memory?[State]), at: Allocation_Table(pm)):$ is_finite[at' 'hierarchy' nodes] (restrict[[(allocator?(pm))], (at' 'hierarchy' nodes), boolean] (at' 'hierarchy' nodes)) </pre>	
--	--

Repeatedly Skolemizing and flattening,
 Adding type constraints for at!1'hierarchy,
 Expanding the definition of finite_tree?,
 Applying disjunctive simplification to flatten sequent,
 Expanding the definition of finite?,
 Hiding formulas: -2, -3,
 Expanding the definition of restrict,
 Expanding the definition of is_finite,
 which is trivially true.
 This completes the proof of allocation_info_TCC1.
 Q.E.D.

C.9.15 Allocation_Table.member_allocation_info

Terse proof for member_allocation_info.
 member_allocation_info:

$$\{1\} \quad \forall (pm: (\text{plain_memory?}), at: \text{Allocation_Table}(pm), addr: \text{Address}, size: \text{posnat}):$$

$$\text{member}((addr, size), \text{allocation_info}(pm)(at)) \supset$$

$$(\exists (ac: (\text{at'hierarchy'nodes})): \text{member}((addr, size), \text{at'info}(ac)))$$

Repeatedly Skolemizing and flattening,
 Expanding the definition of allocation_info,
 Using lemma member_flatten[[Address, posnat]],
 Simplifying, rewriting, and recording with decision procedures,
 Hiding formulas: -4,
 Repeatedly Skolemizing and flattening,
 Using lemma member_map,
 Simplifying, rewriting, and recording with decision procedures,
 Hiding formulas: -2,
 Repeatedly Skolemizing and flattening,
 Instantiating quantified variables,
 Simplifying, rewriting, and recording with decision procedures,
 This completes the proof of member_allocation_info.
 Q.E.D.

C.9.16 Allocation_Table.allocation_consistent

Terse proof for allocation_consistent.
 allocation_consistent:

$$\{1\} \quad \forall (pm: (\text{plain_memory?}), s: \text{State}, at: \text{Allocation_Table}(pm)):$$

$$pm' \text{states}(s) \wedge \text{allocation_ok?}(pm)(at, s) \supset$$

$$\text{allocator_info?}(\text{allocation_info}(pm)(at))$$

Repeatedly Skolemizing and flattening,
 Expanding the definition of allocator_info?,
 Expanding the definition of blocks_pairwise_disjoint,
 Repeatedly Skolemizing and flattening,
 Rewriting using member_list_remove, matching in *,
 Splitting conjunctions,
 we get 2 subgoals:

`allocation_consistent.1:`

{-1}	$size' > 0$
{-2}	$size'' > 0$
{-3}	<code>plain_memory?(pm')</code>
{-4}	<code>pm' 'states(s')</code>
{-5}	<code>allocation_ok?(pm')(at', s')</code>
{-6}	<code>member((addr', size'), allocation_info(pm')(at'))</code>
{1}	<code>TRUE</code>
{2}	<code>blocks_disjoint?(addr', size', addr'', size'')</code>

which is trivially true.

This completes the proof of `allocation_consistent.1`.

`allocation_consistent.2:`

{-1}	$\neg (addr'' = addr' \wedge size'' = size') \wedge$ <code>member((addr'', size''), allocation_info(pm')(at'))</code>
{-2}	$size' > 0$
{-3}	$size'' > 0$
{-4}	<code>plain_memory?(pm')</code>
{-5}	<code>pm' 'states(s')</code>
{-6}	<code>allocation_ok?(pm')(at', s')</code>
{-7}	<code>member((addr', size'), allocation_info(pm')(at'))</code>
{1}	<code>blocks_disjoint?(addr', size', addr'', size'')</code>

Applying disjunctive simplification to flatten sequent,

Forward chaining on `member_allocation_info`,

Forward chaining on `member_allocation_info`,

Hiding formulas: -3, -9,

Repeatedly Skolemizing and flattening,

Using lemma `allocation_memory_pools`,

Simplifying, rewriting, and recording with decision procedures,

Splitting conjunctions,

we get 4 subgoals:

`allocation_consistent.2.1:`

{-1}	$ac' = ac''$
{-2}	<code>allocator?(pm')(ac'')</code>
{-3}	<code>at' 'hierarchy' nodes(ac'')</code>
{-4}	<code>allocator?(pm')(ac')</code>
{-5}	<code>at' 'hierarchy' nodes(ac')</code>
{-6}	<code>member((addr', size'), at' 'info(ac''))</code>
{-7}	<code>member((addr'', size''), at' 'info(ac''))</code>
{-8}	$size' > 0$
{-9}	$size'' > 0$
{-10}	<code>plain_memory?(pm')</code>
{-11}	<code>pm' 'states(s')</code>
{-12}	<code>allocation_ok?(pm')(at', s')</code>
{1}	$addr'' = addr' \wedge size'' = size'$
{2}	<code>blocks_disjoint?(addr', size', addr'', size'')</code>

Using lemma `allocation_consistent_allocator`,

Simplifying, rewriting, and recording with decision procedures,

Expanding the definition of `consistent_allocator?`,

Applying disjunctive simplification to flatten sequent,

Hiding formulas: -2,
 Expanding the definition of `allocator_info?`,
 Expanding the definition of `blocks_pairwise_disjoint`,
 Instantiating quantified variables,
 Rewriting using `member_list_remove`, matching in *,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 This completes the proof of `allocation_consistent.2.1`.

`allocation_consistent.2.2`:

{-1}	<code>disjoint?(ac' 'memory_pool(pm')(s'), ac'' 'memory_pool(pm')(s'))</code>
{-2}	<code>allocator?(pm')(ac'')</code>
{-3}	<code>at' 'hierarchy 'nodes(ac'')</code>
{-4}	<code>allocator?(pm')(ac')</code>
{-5}	<code>at' 'hierarchy 'nodes(ac')</code>
{-6}	<code>member((addr', size'), at' 'info(ac'))</code>
{-7}	<code>member((addr'', size''), at' 'info(ac''))</code>
{-8}	<code>size' > 0</code>
{-9}	<code>size'' > 0</code>
{-10}	<code>plain_memory?(pm')</code>
{-11}	<code>pm' 'states(s')</code>
{-12}	<code>allocation_ok?(pm')(at', s')</code>
{1}	<code>addr'' = addr' \wedge size'' = size'</code>
{2}	<code>blocks_disjoint?(addr', size', addr'', size'')</code>

Rewriting using `blocks_disjoint_disjoint`, matching in *,
 Using lemma `allocation_allocation_point_in_memory_pool`,
 Using lemma `allocation_allocation_point_in_memory_pool`,
 Simplifying, rewriting, and recording with decision procedures,
 Forward chaining on `disjoint_mono`,
 This completes the proof of `allocation_consistent.2.2`.

`allocation_consistent.2.3`:

{-1}	<code>((ac' 'memory_pool(pm')(s') \subseteq ac'' 'allocated(pm')(s')) \wedge <code>block_is_free(ac' 'memory_pool(pm')(s'), at' 'info(ac''))</code>)</code>
{-2}	<code>allocator?(pm')(ac'')</code>
{-3}	<code>at' 'hierarchy 'nodes(ac'')</code>
{-4}	<code>allocator?(pm')(ac')</code>
{-5}	<code>at' 'hierarchy 'nodes(ac')</code>
{-6}	<code>member((addr', size'), at' 'info(ac'))</code>
{-7}	<code>member((addr'', size''), at' 'info(ac''))</code>
{-8}	<code>size' > 0</code>
{-9}	<code>size'' > 0</code>
{-10}	<code>plain_memory?(pm')</code>
{-11}	<code>pm' 'states(s')</code>
{-12}	<code>allocation_ok?(pm')(at', s')</code>
{1}	<code>addr'' = addr' \wedge size'' = size'</code>
{2}	<code>blocks_disjoint?(addr', size', addr'', size'')</code>

Applying disjunctive simplification to flatten sequent,
 Rewriting using `blocks_disjoint_disjoint`, matching in *,
 Using lemma `allocation_allocation_point_in_memory_pool`,
 Simplifying, rewriting, and recording with decision procedures,
 Expanding the definition of `block_is_free`,
 Instantiating the top quantifier in -3 with the terms: `addr''`, `size''`,

C Proof scripts

Simplifying, rewriting, and recording with decision procedures,

Hiding formulas: -2,

Using lemma `disjoint_mono`,

Installing automatic rewrites from: `subset_reflexive`

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `allocation_consistent.2.3`.

`allocation_consistent.2.4`:

{-1}	$((ac''\text{'memory_pool}(pm')(s') \subseteq ac'\text{'allocated}(pm')(s')) \wedge$ $\text{block_is_free}(ac''\text{'memory_pool}(pm')(s'), at'\text{'info}(ac')))$
{-2}	<code>allocator?(pm')(ac'')</code>
{-3}	<code>at'\text{'hierarchy'nodes}(ac'')</code>
{-4}	<code>allocator?(pm')(ac')</code>
{-5}	<code>at'\text{'hierarchy'nodes}(ac')</code>
{-6}	<code>member((addr', size'), at'\text{'info}(ac'))</code>
{-7}	<code>member((addr'', size''), at'\text{'info}(ac''))</code>
{-8}	<code>size' > 0</code>
{-9}	<code>size'' > 0</code>
{-10}	<code>plain_memory?(pm')</code>
{-11}	<code>pm'\text{'states}(s')</code>
{-12}	<code>allocation_ok?(pm')(at', s')</code>
{1}	$addr'' = addr' \wedge size'' = size'$
{2}	<code>blocks_disjoint?(addr', size', addr'', size'')</code>

Applying disjunctive simplification to flatten sequent,

Hiding formulas: -1,

Rewriting using `blocks_disjoint_disjoint`, matching in `*`,

Using lemma `allocation_allocation_point_in_memory_pool`,

Simplifying, rewriting, and recording with decision procedures,

Expanding the definition of `block_is_free`,

Instantiating the top quantifier in -2 with the terms: `addr', size'`,

Simplifying, rewriting, and recording with decision procedures,

Using lemma `disjoint_mono`,

Installing automatic rewrites from: `subset_reflexive`

Simplifying, rewriting, and recording with decision procedures,

Rewriting using `disjoint_symmetric`, matching in `*`,

This completes the proof of `allocation_consistent.2.4`.

Q.E.D.

C.9.17

Allocation_Table.allocation_alloc_other_private_memory_TCC1

Terse proof for `allocation_alloc_other_private_memory_TCC1`.

allocation_alloc_other_private_memory_TCC1:

```
{1}  ∃ (pm: (plain_memory?), s: State, at: Allocation_Table(pm),
      ac1, ac2: (at'hierarchy'nodes), size: posnat):
      pm'states(s) ∧
      allocation_ok?(pm)(at, s) ∧
      OK?(ac1'alloc(pm)(size)(s)) ∧
      ¬ data(ac1'alloc(pm)(size)(s)) = null_address ∧ ¬ ac1 = ac2
      ⊃
      OK?[State, Address](ac1'alloc(pm)(size)(s)) ∨
      Exception?[State, Address](ac1'alloc(pm)(size)(s))
```

Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of allocation_alloc_other_private_memory_TCC1.
 Q.E.D.

C.9.18

Allocation_Table.allocation_alloc_other_private_memory_TCC2

Terse proof for allocation_alloc_other_private_memory_TCC2.

allocation_alloc_other_private_memory_TCC2:

```
{1}  ∃ (pm: (plain_memory?), s: State, at: Allocation_Table(pm),
      ac1, ac2: (at'hierarchy'nodes), size: posnat):
      pm'states(s) ∧
      allocation_ok?(pm)(at, s) ∧
      OK?(ac1'alloc(pm)(size)(s)) ∧
      ¬ data(ac1'alloc(pm)(size)(s)) = null_address ∧ ¬ ac1 = ac2
      ⊃ pm'states(state[State, Address](ac1'alloc(pm)(size)(s)))
```

Repeatedly Skolemizing and flattening,
 Rewriting using allocator_next_state_alloc, matching in *,
 Expanding the definition of has_next_state,
 which is trivially true.
 This completes the proof of allocation_alloc_other_private_memory_TCC2.
 Q.E.D.

C.9.19 Allocation_Table.allocation_alloc_other_private_memory

Terse proof for allocation_alloc_other_private_memory.

allocation_alloc_other_private_memory:

```
{1}  ∃ (pm: (plain_memory?), s: State, at: Allocation_Table(pm),
      ac1, ac2: (at'hierarchy'nodes), size: posnat):
      pm'states(s) ∧
      allocation_ok?(pm)(at, s) ∧
      OK?(ac1'alloc(pm)(size)(s)) ∧
      ¬ data(ac1'alloc(pm)(size)(s)) = null_address ∧ ¬ ac1 = ac2
      ⊃ stays_unchanged(pm)(s, state(ac1'alloc(pm)(size)(s)), ac2'private_mem(pm))
```

Repeatedly Skolemizing and flattening,

Using lemma `allocator_only_changes_alloc`,
 Simplifying, rewriting, and recording with decision procedures,
 Using lemma `stays_unchanged_only_changes_disjoint`,
 Installing automatic rewrites from: `allocator_next_state_alloc` `allocator_subset_private_memory_plain_memory`
`has_next_state`
 Simplifying, rewriting, and recording with decision procedures,
 Using lemma `allocation_private_memory_disjoint`,
 Simplifying, rewriting, and recording with decision procedures,
 This completes the proof of `allocation_alloc_other_private_memory`.
 Q.E.D.

C.9.20 Allocation_Table.allocation_alloc_TCC1

Terse proof for `allocation_alloc_TCC1`.

`allocation_alloc_TCC1`:

$ \begin{aligned} \{1\} \quad & \forall (\text{pm}: (\text{plain_memory?}), s: \text{State}, \text{at}: \text{Allocation_Table}(\text{pm}), \text{ac}: (\text{at}'\text{hierarchy}'\text{nodes}), \\ & \text{size}: \text{posnat}): \\ & \text{pm}'\text{states}(s) \wedge \\ & \text{allocation_ok?}(\text{pm})(\text{at}, s) \wedge \\ & \text{OK?}(\text{ac}'\text{alloc}(\text{pm})(\text{size})(s)) \wedge \neg \text{data}(\text{ac}'\text{alloc}(\text{pm})(\text{size})(s)) = \text{null_address} \\ & \supset \\ & \text{OK?}[\text{State}, \text{Address}](\text{ac}'\text{alloc}(\text{pm})(\text{size})(s)) \vee \\ & \text{Exception?}[\text{State}, \text{Address}](\text{ac}'\text{alloc}(\text{pm})(\text{size})(s)) \end{aligned} $

Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of `allocation_alloc_TCC1`.
 Q.E.D.

C.9.21 Allocation_Table.allocation_alloc

Terse proof for `allocation_alloc`.

`allocation_alloc`:

$ \begin{aligned} \{1\} \quad & \forall (\text{pm}: (\text{plain_memory?}), s: \text{State}, \text{at}: \text{Allocation_Table}(\text{pm}), \text{ac}: (\text{at}'\text{hierarchy}'\text{nodes}), \\ & \text{size}: \text{posnat}): \\ & \text{pm}'\text{states}(s) \wedge \\ & \text{allocation_ok?}(\text{pm})(\text{at}, s) \wedge \\ & \text{OK?}(\text{ac}'\text{alloc}(\text{pm})(\text{size})(s)) \wedge \neg \text{data}(\text{ac}'\text{alloc}(\text{pm})(\text{size})(s)) = \text{null_address} \\ & \supset \\ & \text{allocation_ok?}(\text{pm}) \\ & \quad (\text{at} \\ & \quad \quad \text{WITH } [\text{'info}(\text{ac}) \\ & \quad \quad \quad := \text{cons}((\text{data}(\text{ac}'\text{alloc}(\text{pm})(\text{size})(s)), \text{size}), \\ & \quad \quad \quad \text{at}'\text{info}(\text{ac}))], \\ & \quad \quad \text{state}(\text{ac}'\text{alloc}(\text{pm})(\text{size})(s)) \end{aligned} $
--

Repeatedly Skolemizing and flattening,
 Installing automatic rewrites from: `allocator_next_state_alloc` `has_next_state` `subset_bigger_union_left`
`allocator_memory_pool_malloc` `allocator_allocated_alloc` `subset_equal`
 Expanding the definition of `allocation_ok?`,
 Applying propositional simplification,

we get 5 subgoals:

allocation_alloc.1:

<ul style="list-style-type: none"> {-1} plain_memory?(pm') {-2} allocator?(pm')(ac') {-3} at' 'hierarchy' nodes(ac') {-4} size' > 0 {-5} pm' 'states'(s') {-6} allocation_ok?(pm')(at', s') {-7} OK?(ac' 'alloc(pm')(size')(s')) 	<hr/> <ul style="list-style-type: none"> {1} $\forall (ac1, ac2: (at' 'hierarchy' nodes)):$ $at' 'hierarchy' edges(ac1, ac2) \supset$ $(ac2' memory_pool(pm')(state(ac' 'alloc(pm')(size')(s')))) \subseteq ac1' allocated(pm')(state(ac' 'alloc(pm')(size')(s')))$ \wedge $block_is_free(ac2' memory_pool(pm')(state(ac' 'alloc(pm')(size')(s'))),$ $at' 'info$ $WITH [(ac')$ $:= cons((data(ac' 'alloc(pm')(size')(s')), size'),$ $at' 'info(ac'))]$ $(ac1)$ {2} $data(ac' 'alloc(pm')(size')(s')) = null_address$
--	---

Repeatedly Skolemizing and flattening,

Using lemma allocation_subnodes_allocated_and_free,

Simplifying, rewriting, and recording with decision procedures,

Applying disjunctive simplification to flatten sequent,

Case splitting on ac1 = ac2!1,

we get 2 subgoals:

allocation_alloc.1.1:

<ul style="list-style-type: none"> {-1} ac' = ac2' {-2} (ac2' memory_pool(pm')(s') \subseteq ac1' allocated(pm')(s')) {-3} block_is_free(ac2' memory_pool(pm')(s'), at' info(ac1')) {-4} allocator?(pm')(ac1') {-5} at' 'hierarchy' nodes(ac1') {-6} allocator?(pm')(ac2') {-7} at' 'hierarchy' nodes(ac2') {-8} at' 'hierarchy' edges(ac1', ac2') {-9} plain_memory?(pm') {-10} allocator?(pm')(ac') {-11} at' 'hierarchy' nodes(ac') {-12} size' > 0 {-13} pm' 'states'(s') {-14} allocation_ok?(pm')(at', s') {-15} OK?(ac' 'alloc(pm')(size')(s')) 	<hr/> <ul style="list-style-type: none"> {1} $(ac2' memory_pool(pm')(state(ac' 'alloc(pm')(size')(s')))) \subseteq ac1' allocated(pm')(state(ac' 'alloc(pm')(size')(s')))$ \wedge $block_is_free(ac2' memory_pool(pm')(state(ac' 'alloc(pm')(size')(s'))),$ $at' 'info$ $WITH [(ac')$ $:= cons((data(ac' 'alloc(pm')(size')(s')), size'),$ $at' 'info(ac'))]$ $(ac1')$ {2} $data(ac' 'alloc(pm')(size')(s')) = null_address$
---	---

C Proof scripts

Replacing using formula -1,

Simplifying, rewriting, and recording with decision procedures,

Using lemma `allocators_different_in_hierarchy`,

Simplifying, rewriting, and recording with decision procedures,

Using lemma `allocation_alloc_other_private_memory`,

Simplifying, rewriting, and recording with decision procedures,

Using lemma `allocator_const_private_memory_allocated`,

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `allocation_alloc.1.1`.

`allocation_alloc.1.2`:

{-1}	$(ac2' \text{'memory_pool}(pm')(s') \subseteq ac1' \text{'allocated}(pm')(s'))$
{-2}	$block_is_free(ac2' \text{'memory_pool}(pm')(s'), at' \text{'info}(ac1'))$
{-3}	$allocator?(pm')(ac1')$
{-4}	$at' \text{'hierarchy'nodes}(ac1')$
{-5}	$allocator?(pm')(ac2')$
{-6}	$at' \text{'hierarchy'nodes}(ac2')$
{-7}	$at' \text{'hierarchy'edges}(ac1', ac2')$
{-8}	$plain_memory?(pm')$
{-9}	$allocator?(pm')(ac')$
{-10}	$at' \text{'hierarchy'nodes}(ac')$
{-11}	$size' > 0$
{-12}	$pm' \text{'states}(s')$
{-13}	$allocation_ok?(pm')(at', s')$
{-14}	$OK?(ac' \text{'alloc}(pm')(size')(s'))$
{1}	$ac' = ac2'$
{2}	$(ac2' \text{'memory_pool}(pm')(state(ac' \text{'alloc}(pm')(size')(s')))) \subseteq ac1' \text{'allocated}(pm')(state(ac' \text{'alloc}(pm')(size')(s')))$ \wedge $block_is_free(ac2' \text{'memory_pool}(pm')(state(ac' \text{'alloc}(pm')(size')(s'))),$ $at' \text{'info}$ $WITH [(ac')$ $:= cons((data(ac' \text{'alloc}(pm')(size')(s')), size'),$ $at' \text{'info}(ac'))]$ $(ac1'))$
{3}	$data(ac' \text{'alloc}(pm')(size')(s')) = \text{null_address}$

Using lemma `allocation_alloc_other_private_memory`,

Simplifying, rewriting, and recording with decision procedures,

Using lemma `allocator_const_private_memory_memory_pool`,

Simplifying, rewriting, and recording with decision procedures,

Replacing using formula -1,

Hiding formulas: -1,

Case splitting on `ac1!1 = ac!1`,

we get 2 subgoals:

allocation_alloc.1.2.1:

<pre> {-1} ac1' = ac' {-2} stays_unchanged(pm') (s', state(ac' ' alloc(pm')(size')(s')), ac2' ' private_mem(pm')) {-3} (ac2' ' memory_pool(pm')(s') ⊆ ac1' ' allocated(pm')(s')) {-4} block_is_free(ac2' ' memory_pool(pm')(s'), at' ' info(ac1')) {-5} allocator?(pm')(ac1') {-6} at' ' hierarchy ' nodes(ac1') {-7} allocator?(pm')(ac2') {-8} at' ' hierarchy ' nodes(ac2') {-9} at' ' hierarchy ' edges(ac1', ac2') {-10} plain_memory?(pm') {-11} allocator?(pm')(ac') {-12} at' ' hierarchy ' nodes(ac') {-13} size' > 0 {-14} pm' ' states(s') {-15} allocation_ok?(pm')(at', s') {-16} OK?(ac' ' alloc(pm')(size')(s')) </pre>	<hr style="border: 0.5px solid black;"/> <pre> {1} ac' = ac2' {2} (ac2' ' memory_pool(pm')(s') ⊆ ac1' ' allocated(pm')(state(ac' ' alloc(pm')(size')(s')))) ^ block_is_free(ac2' ' memory_pool(pm')(s'), at' ' info WITH [(ac' := cons((data(ac' ' alloc(pm')(size')(s')), size'), at' ' info(ac'))] (ac1')) {3} data(ac' ' alloc(pm')(size')(s')) = null_address </pre>
--	---

Replacing using formula -1,

Simplifying, rewriting, and recording with decision procedures,

Rewriting using block_is_free_add, matching in *,

Using lemma allocator_alloc_disjoint,

Simplifying, rewriting, and recording with decision procedures,

Using lemma disjoint_mono,

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of allocation_alloc.1.2.1.

allocation_alloc.1.2.2:

{-1}	stays_unchanged(pm')
	(s', state(ac' alloc(pm')(size')(s')), ac2' private_mem(pm'))
{-2}	(ac2' memory_pool(pm')(s') \subseteq ac1' allocated(pm')(s'))
{-3}	block_is_free(ac2' memory_pool(pm')(s'), at' info(ac1'))
{-4}	allocator?(pm')(ac1')
{-5}	at' hierarchy' nodes(ac1')
{-6}	allocator?(pm')(ac2')
{-7}	at' hierarchy' nodes(ac2')
{-8}	at' hierarchy' edges(ac1', ac2')
{-9}	plain_memory?(pm')
{-10}	allocator?(pm')(ac')
{-11}	at' hierarchy' nodes(ac')
{-12}	size' > 0
{-13}	pm' states(s')
{-14}	allocation_ok?(pm')(at', s')
{-15}	OK?(ac' alloc(pm')(size')(s'))
{1}	ac1' = ac'
{2}	ac' = ac2'
{3}	(ac2' memory_pool(pm')(s') \subseteq ac1' allocated(pm')(state(ac' alloc(pm')(size')(s')))) \wedge block_is_free(ac2' memory_pool(pm')(s'), at' info WITH [(ac') := cons((data(ac' alloc(pm')(size')(s')), size'), at' info(ac'))] (ac1'))
{4}	data(ac' alloc(pm')(size')(s')) = null_address

Using lemma allocation_alloc_other_private_memory,
Simplifying, rewriting, and recording with decision procedures,
Using lemma allocator_const_private_memory_allocated,
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of allocation_alloc.1.2.2.

allocation_alloc.2:

{-1}	plain_memory?(pm')
{-2}	allocator?(pm')(ac')
{-3}	at' hierarchy' nodes(ac')
{-4}	size' > 0
{-5}	pm' states(s')
{-6}	allocation_ok?(pm')(at', s')
{-7}	OK?(ac' alloc(pm')(size')(s'))
{1}	\forall (ac1, ac2, ac3: (at' hierarchy' nodes)): at' hierarchy' edges(ac1, ac2) \wedge at' hierarchy' edges(ac1, ac3) \supset ac2 = ac3 \vee disjoint?(ac2' memory_pool(pm')(state(ac' alloc(pm')(size')(s'))), ac3' memory_pool(pm')(state(ac' alloc(pm')(size')(s'))))
{2}	data(ac' alloc(pm')(size')(s')) = null_address

Repeatedly Skolemizing and flattening,
Using lemma allocation_subnodes_disjoint,
Simplifying, rewriting, and recording with decision procedures,
Case splitting on ac2!1 = ac!1,

we get 2 subgoals:

allocation_alloc.2.1:

{-1}	$ac2' = ac'$
{-2}	$disjoint?(ac2' \text{ 'memory_pool}(pm')(s'), ac3' \text{ 'memory_pool}(pm')(s'))$
{-3}	$allocator?(pm')(ac1')$
{-4}	$at' \text{ 'hierarchy' nodes}(ac1')$
{-5}	$allocator?(pm')(ac2')$
{-6}	$at' \text{ 'hierarchy' nodes}(ac2')$
{-7}	$allocator?(pm')(ac3')$
{-8}	$at' \text{ 'hierarchy' nodes}(ac3')$
{-9}	$at' \text{ 'hierarchy' edges}(ac1', ac2')$
{-10}	$at' \text{ 'hierarchy' edges}(ac1', ac3')$
{-11}	$plain_memory?(pm')$
{-12}	$allocator?(pm')(ac')$
{-13}	$at' \text{ 'hierarchy' nodes}(ac')$
{-14}	$size' > 0$
{-15}	$pm' \text{ 'states}(s')$
{-16}	$allocation_ok?(pm')(at', s')$
{-17}	$OK?(ac' \text{ 'alloc}(pm')(size')(s'))$
{1}	$ac2' = ac3'$
{2}	$disjoint?(ac2' \text{ 'memory_pool}(pm')(state(ac' \text{ 'alloc}(pm')(size')(s'))),$ $ac3' \text{ 'memory_pool}(pm')(state(ac' \text{ 'alloc}(pm')(size')(s'))))$
{3}	$data(ac' \text{ 'alloc}(pm')(size')(s')) = \text{null_address}$

Replacing using formula -1,

Using lemma allocation_alloc_other_private_memory,

Simplifying, rewriting, and recording with decision procedures,

Using lemma allocator_const_private_memory_memory_pool,

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of allocation_alloc.2.1.

allocation_alloc.2.2:

{-1}	$disjoint?(ac2' \text{ 'memory_pool}(pm')(s'), ac3' \text{ 'memory_pool}(pm')(s'))$
{-2}	$allocator?(pm')(ac1')$
{-3}	$at' \text{ 'hierarchy' nodes}(ac1')$
{-4}	$allocator?(pm')(ac2')$
{-5}	$at' \text{ 'hierarchy' nodes}(ac2')$
{-6}	$allocator?(pm')(ac3')$
{-7}	$at' \text{ 'hierarchy' nodes}(ac3')$
{-8}	$at' \text{ 'hierarchy' edges}(ac1', ac2')$
{-9}	$at' \text{ 'hierarchy' edges}(ac1', ac3')$
{-10}	$plain_memory?(pm')$
{-11}	$allocator?(pm')(ac')$
{-12}	$at' \text{ 'hierarchy' nodes}(ac')$
{-13}	$size' > 0$
{-14}	$pm' \text{ 'states}(s')$
{-15}	$allocation_ok?(pm')(at', s')$
{-16}	$OK?(ac' \text{ 'alloc}(pm')(size')(s'))$
{1}	$ac2' = ac'$
{2}	$ac2' = ac3'$
{3}	$disjoint?(ac2' \text{ 'memory_pool}(pm')(state(ac' \text{ 'alloc}(pm')(size')(s'))),$ $ac3' \text{ 'memory_pool}(pm')(state(ac' \text{ 'alloc}(pm')(size')(s'))))$
{4}	$data(ac' \text{ 'alloc}(pm')(size')(s')) = \text{null_address}$

C Proof scripts

Case splitting on $ac3!1 = ac!1$,

we get 2 subgoals:

`allocation_alloc.2.2.1:`

{-1}	$ac3' = ac'$
{-2}	$disjoint?(ac2' \text{ memory_pool}(pm')(s'), ac3' \text{ memory_pool}(pm')(s'))$
{-3}	$allocator?(pm')(ac1')$
{-4}	$at' \text{ hierarchy nodes}(ac1')$
{-5}	$allocator?(pm')(ac2')$
{-6}	$at' \text{ hierarchy nodes}(ac2')$
{-7}	$allocator?(pm')(ac3')$
{-8}	$at' \text{ hierarchy nodes}(ac3')$
{-9}	$at' \text{ hierarchy edges}(ac1', ac2')$
{-10}	$at' \text{ hierarchy edges}(ac1', ac3')$
{-11}	$plain_memory?(pm')$
{-12}	$allocator?(pm')(ac')$
{-13}	$at' \text{ hierarchy nodes}(ac')$
{-14}	$size' > 0$
{-15}	$pm' \text{ states}(s')$
{-16}	$allocation_ok?(pm')(at', s')$
{-17}	$OK?(ac' \text{ alloc}(pm')(size')(s'))$
{1}	$ac2' = ac'$
{2}	$ac2' = ac3'$
{3}	$disjoint?(ac2' \text{ memory_pool}(pm')(state(ac' \text{ alloc}(pm')(size')(s'))),$ $ac3' \text{ memory_pool}(pm')(state(ac' \text{ alloc}(pm')(size')(s'))))$
{4}	$data(ac' \text{ alloc}(pm')(size')(s')) = \text{null_address}$

Replacing using formula -1,

Using lemma `allocation_alloc_other_private_memory`,

Simplifying, rewriting, and recording with decision procedures,

Using lemma `allocator_const_private_memory_memory_pool`,

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `allocation_alloc.2.2.1`.

allocation_alloc.2.2.2:

{-1}	disjoint?(ac2' 'memory_pool(pm')(s'), ac3' 'memory_pool(pm')(s'))
{-2}	allocator?(pm')(ac1')
{-3}	at' 'hierarchy' 'nodes(ac1')
{-4}	allocator?(pm')(ac2')
{-5}	at' 'hierarchy' 'nodes(ac2')
{-6}	allocator?(pm')(ac3')
{-7}	at' 'hierarchy' 'nodes(ac3')
{-8}	at' 'hierarchy' 'edges(ac1', ac2')
{-9}	at' 'hierarchy' 'edges(ac1', ac3')
{-10}	plain_memory?(pm')
{-11}	allocator?(pm')(ac')
{-12}	at' 'hierarchy' 'nodes(ac')
{-13}	size' > 0
{-14}	pm' 'states(s')
{-15}	allocation_ok?(pm')(at', s')
{-16}	OK?(ac' 'alloc(pm')(size')(s'))
{1}	ac3' = ac'
{2}	ac2' = ac'
{3}	ac2' = ac3'
{4}	disjoint?(ac2' 'memory_pool(pm')(state(ac' 'alloc(pm')(size')(s'))), ac3' 'memory_pool(pm')(state(ac' 'alloc(pm')(size')(s'))))
{5}	data(ac' 'alloc(pm')(size')(s')) = null_address

Using lemma allocation_alloc_other_private_memory,

Using lemma allocation_alloc_other_private_memory,

Simplifying, rewriting, and recording with decision procedures,

Using lemma allocator_const_private_memory_memory_pool,

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of allocation_alloc.2.2.2.

allocation_alloc.3:

{-1}	plain_memory?(pm')
{-2}	allocator?(pm')(ac')
{-3}	at' 'hierarchy' 'nodes(ac')
{-4}	size' > 0
{-5}	pm' 'states(s')
{-6}	allocation_ok?(pm')(at', s')
{-7}	OK?(ac' 'alloc(pm')(size')(s'))
{1}	\forall (ac1, ac2: (roots(at' 'hierarchy'))): ac1 = ac2 \vee disjoint?(ac1' 'memory_pool(pm')(state(ac' 'alloc(pm')(size')(s'))), ac2' 'memory_pool(pm')(state(ac' 'alloc(pm')(size')(s'))))
{2}	data(ac' 'alloc(pm')(size')(s')) = null_address

Repeatedly Skolemizing and flattening,

Using lemma allocation_roots_disjoint,

Simplifying, rewriting, and recording with decision procedures,

Case splitting on ac1!1 = ac!1,

we get 2 subgoals:

allocation_alloc.3.1:

{-1}	ac1' = ac'
{-2}	disjoint?(ac1' 'memory_pool(pm')(s'), ac2' 'memory_pool(pm')(s'))
{-3}	allocator?(pm')(ac1')
{-4}	at' 'hierarchy' nodes(ac1')
{-5}	roots(at' 'hierarchy')(ac1')
{-6}	allocator?(pm')(ac2')
{-7}	at' 'hierarchy' nodes(ac2')
{-8}	roots(at' 'hierarchy')(ac2')
{-9}	plain_memory?(pm')
{-10}	allocator?(pm')(ac')
{-11}	at' 'hierarchy' nodes(ac')
{-12}	size' > 0
{-13}	pm' 'states(s')
{-14}	allocation_ok?(pm')(at', s')
{-15}	OK?(ac' 'alloc(pm')(size')(s'))
{1}	ac1' = ac2'
{2}	disjoint?(ac1' 'memory_pool(pm')(state(ac' 'alloc(pm')(size')(s'))), ac2' 'memory_pool(pm')(state(ac' 'alloc(pm')(size')(s'))))
{3}	data(ac' 'alloc(pm')(size')(s')) = null_address

Replacing using formula -1,

Using lemma allocation_alloc_other_private_memory,

Simplifying, rewriting, and recording with decision procedures,

Using lemma allocator_const_private_memory_memory_pool,

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of allocation_alloc.3.1.

allocation_alloc.3.2:

{-1}	disjoint?(ac1' 'memory_pool(pm')(s'), ac2' 'memory_pool(pm')(s'))
{-2}	allocator?(pm')(ac1')
{-3}	at' 'hierarchy' nodes(ac1')
{-4}	roots(at' 'hierarchy')(ac1')
{-5}	allocator?(pm')(ac2')
{-6}	at' 'hierarchy' nodes(ac2')
{-7}	roots(at' 'hierarchy')(ac2')
{-8}	plain_memory?(pm')
{-9}	allocator?(pm')(ac')
{-10}	at' 'hierarchy' nodes(ac')
{-11}	size' > 0
{-12}	pm' 'states(s')
{-13}	allocation_ok?(pm')(at', s')
{-14}	OK?(ac' 'alloc(pm')(size')(s'))
{1}	ac1' = ac'
{2}	ac1' = ac2'
{3}	disjoint?(ac1' 'memory_pool(pm')(state(ac' 'alloc(pm')(size')(s'))), ac2' 'memory_pool(pm')(state(ac' 'alloc(pm')(size')(s'))))
{4}	data(ac' 'alloc(pm')(size')(s')) = null_address

Case splitting on ac2!1 = ac!1,

we get 2 subgoals:

allocation_alloc.3.2.1:

{-1}	$ac2' = ac'$
{-2}	$disjoint?(ac1' \text{ 'memory_pool}(pm')(s'), ac2' \text{ 'memory_pool}(pm')(s'))$
{-3}	$allocator?(pm')(ac1')$
{-4}	$at' \text{ 'hierarchy' nodes}(ac1')$
{-5}	$roots(at' \text{ 'hierarchy})(ac1')$
{-6}	$allocator?(pm')(ac2')$
{-7}	$at' \text{ 'hierarchy' nodes}(ac2')$
{-8}	$roots(at' \text{ 'hierarchy})(ac2')$
{-9}	$plain_memory?(pm')$
{-10}	$allocator?(pm')(ac')$
{-11}	$at' \text{ 'hierarchy' nodes}(ac')$
{-12}	$size' > 0$
{-13}	$pm' \text{ 'states}(s')$
{-14}	$allocation_ok?(pm')(at', s')$
{-15}	$OK?(ac' \text{ 'alloc}(pm')(size')(s'))$
{1}	$ac1' = ac'$
{2}	$ac1' = ac2'$
{3}	$disjoint?(ac1' \text{ 'memory_pool}(pm')(state(ac' \text{ 'alloc}(pm')(size')(s'))),$ $ac2' \text{ 'memory_pool}(pm')(state(ac' \text{ 'alloc}(pm')(size')(s'))))$
{4}	$data(ac' \text{ 'alloc}(pm')(size')(s')) = null_address$

Replacing using formula -1,

Using lemma allocation_alloc_other_private_memory,

Simplifying, rewriting, and recording with decision procedures,

Using lemma allocator_const_private_memory_memory_pool,

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of allocation_alloc.3.2.1.

allocation_alloc.3.2.2:

{-1}	$disjoint?(ac1' \text{ 'memory_pool}(pm')(s'), ac2' \text{ 'memory_pool}(pm')(s'))$
{-2}	$allocator?(pm')(ac1')$
{-3}	$at' \text{ 'hierarchy' nodes}(ac1')$
{-4}	$roots(at' \text{ 'hierarchy})(ac1')$
{-5}	$allocator?(pm')(ac2')$
{-6}	$at' \text{ 'hierarchy' nodes}(ac2')$
{-7}	$roots(at' \text{ 'hierarchy})(ac2')$
{-8}	$plain_memory?(pm')$
{-9}	$allocator?(pm')(ac')$
{-10}	$at' \text{ 'hierarchy' nodes}(ac')$
{-11}	$size' > 0$
{-12}	$pm' \text{ 'states}(s')$
{-13}	$allocation_ok?(pm')(at', s')$
{-14}	$OK?(ac' \text{ 'alloc}(pm')(size')(s'))$
{1}	$ac2' = ac'$
{2}	$ac1' = ac'$
{3}	$ac1' = ac2'$
{4}	$disjoint?(ac1' \text{ 'memory_pool}(pm')(state(ac' \text{ 'alloc}(pm')(size')(s'))),$ $ac2' \text{ 'memory_pool}(pm')(state(ac' \text{ 'alloc}(pm')(size')(s'))))$
{5}	$data(ac' \text{ 'alloc}(pm')(size')(s')) = null_address$

Using lemma allocation_alloc_other_private_memory,

Using lemma allocation_alloc_other_private_memory,

Simplifying, rewriting, and recording with decision procedures,

C Proof scripts

Using lemma `allocator_const_private_memory_memory_pool`,
 Simplifying, rewriting, and recording with decision procedures,
 This completes the proof of `allocation_alloc.3.2.2`.

`allocation_alloc.4`:

{-1}	<code>plain_memory?(pm')</code>
{-2}	<code>allocator?(pm')(ac')</code>
{-3}	<code>at' hierarchy nodes(ac')</code>
{-4}	<code>size' > 0</code>
{-5}	<code>pm' states(s')</code>
{-6}	<code>allocation_ok?(pm')(at', s')</code>
{-7}	<code>OK?(ac' alloc(pm')(size')(s'))</code>
{1}	$\forall (ac1, ac2: (at' hierarchy nodes)):$ $ac1 = ac2 \vee disjoint?(ac1 private_mem(pm'), ac2 private_mem(pm'))$
{2}	<code>data(ac' alloc(pm')(size')(s')) = null_address</code>

Repeatedly Skolemizing and flattening,
 Using lemma `allocation_private_memory_disjoint`,
 Simplifying, rewriting, and recording with decision procedures,
 This completes the proof of `allocation_alloc.4`.

`allocation_alloc.5`:

{-1}	<code>plain_memory?(pm')</code>
{-2}	<code>allocator?(pm')(ac')</code>
{-3}	<code>at' hierarchy nodes(ac')</code>
{-4}	<code>size' > 0</code>
{-5}	<code>pm' states(s')</code>
{-6}	<code>allocation_ok?(pm')(at', s')</code>
{-7}	<code>OK?(ac' alloc(pm')(size')(s'))</code>
{1}	$\forall (ac: (at' hierarchy nodes)):$ $consistent_allocator?(pm')$ $(ac,$ $at' info$ $WITH [(ac')$ $:= cons((data(ac' alloc(pm')(size')(s')),$ $size'),$ $at' info(ac'))]$ $(ac))$ $(state(ac' alloc(pm')(size')(s')))$
{2}	<code>data(ac' alloc(pm')(size')(s')) = null_address</code>

Repeatedly Skolemizing and flattening,
 Using lemma `allocation_consistent_allocator`,
 Simplifying, rewriting, and recording with decision procedures,
 Case splitting on `ac!1 = ac!2`,
 we get 2 subgoals:

allocation_alloc.5.1:

{-1}	$ac' = ac''$
{-2}	$consistent_allocator?(pm')(ac'', at'info(ac''))(s')$
{-3}	$allocator?(pm')(ac'')$
{-4}	$at' hierarchy' nodes(ac'')$
{-5}	$plain_memory?(pm')$
{-6}	$allocator?(pm')(ac')$
{-7}	$at' hierarchy' nodes(ac')$
{-8}	$size' > 0$
{-9}	$pm' states(s')$
{-10}	$allocation_ok?(pm')(at', s')$
{-11}	$OK?(ac' alloc(pm')(size')(s'))$
{1}	$consistent_allocator?(pm')$ $(ac'',$ $at'info$ $WITH [(ac')$ $:= cons((data(ac' alloc(pm')(size')(s')), size'),$ $at'info(ac'))]$ $(ac''))$ $(state(ac' alloc(pm')(size')(s')))$
{2}	$data(ac' alloc(pm')(size')(s')) = null_address$

Replacing using formula -1,
Simplifying, rewriting, and recording with decision procedures,
Rewriting using consistent_allocator_alloc, matching in *,
This completes the proof of allocation_alloc.5.1.

allocation_alloc.5.2:

{-1}	$consistent_allocator?(pm')(ac'', at'info(ac''))(s')$
{-2}	$allocator?(pm')(ac'')$
{-3}	$at' hierarchy' nodes(ac'')$
{-4}	$plain_memory?(pm')$
{-5}	$allocator?(pm')(ac')$
{-6}	$at' hierarchy' nodes(ac')$
{-7}	$size' > 0$
{-8}	$pm' states(s')$
{-9}	$allocation_ok?(pm')(at', s')$
{-10}	$OK?(ac' alloc(pm')(size')(s'))$
{1}	$ac' = ac''$
{2}	$consistent_allocator?(pm')$ $(ac'',$ $at'info$ $WITH [(ac')$ $:= cons((data(ac' alloc(pm')(size')(s')), size'),$ $at'info(ac'))]$ $(ac''))$ $(state(ac' alloc(pm')(size')(s')))$
{3}	$data(ac' alloc(pm')(size')(s')) = null_address$

Simplifying, rewriting, and recording with decision procedures,
Using lemma consistent_allocator_state_change,
Simplifying, rewriting, and recording with decision procedures,
Using lemma allocation_alloc_other_private_memory,
Simplifying, rewriting, and recording with decision procedures,

Using lemma `allocator_const_private_memory_allocated`,
 Simplifying, rewriting, and recording with decision procedures,
 This completes the proof of `allocation_alloc.5.2`.
 Q.E.D.

C.10 Proofs for Allocator (`allocators.pvs`)

C.10.1 `Allocator.null_address_TCC1`

Terse proof for `null_address_TCC1`.

`null_address_TCC1`:

{1} Mem?(Mem(0)'type_of)

Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of `null_address_TCC1`.
 Q.E.D.

C.10.2 `Allocator.allocator?_TCC1`

Terse proof for `allocator?_TCC1`.

`allocator?_TCC1`:

{1} \forall (pm: (plain_memory?[State]), ac: Allocator):
 (ac'private_mem(pm) \subseteq (pm'ro_addr \cup pm'rw_addr)) \supset
 (\forall (s: State):
 (ac'allocated(pm)(s) \subseteq ac'memory_pool(pm)(s)) \wedge pm'states(s) \supset
 (\forall (size: posnat):
 disjoint?(ac'allocated(pm)(s),
 address_block(data(ac'alloc(pm)(size)(s)), size))
 \wedge
 OK?(ac'alloc(pm)(size)(s)) \wedge
 \neg data(ac'alloc(pm)(size)(s)) = null_address
 \supset
 OK?[State, Address](ac'alloc(pm)(size)(s)) \vee
 Exception?[State, Address](ac'alloc(pm)(size)(s))))

Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of `allocator?_TCC1`.
 Q.E.D.

C.10.3 `Allocator.allocator?_TCC2`

Terse proof for `allocator?_TCC2`.

allocator?_TCC2:

```

{1}  ∇ (pm: (plain_memory?[State]), ac: Allocator):
      (ac'private_mem(pm) ⊆ (pm'ro_addr ∪ pm'rw_addr)) ⊃
      (∇ (s: State):
        (∇ (addr: Address):
          OK?(ac'free(pm)(addr)(s)) ≡ up?(ac'freed_size(pm)(addr)(s)))
        ∧
        (∇ (size: posnat):
          OK?(ac'alloc(pm)(size)(s)) ∧
          ¬ data(ac'alloc(pm)(size)(s)) = null_address
          ⊃
          disjoint?(ac'allocated(pm)(s),
                    address_block(data(ac'alloc(pm)(size)(s)), size))
          ∧
          ac'allocated(pm)(state(ac'alloc(pm)(size)(s))) =
            (ac'allocated(pm)(s) ∪ address_block(data(ac'alloc(pm)(size)(s)), size))
          ∧
          ac'memory_pool(pm)(state(ac'alloc(pm)(size)(s))) =
            ac'memory_pool(pm)(s)
          ∧ (ac'allocated(pm)(s) ⊆ ac'memory_pool(pm)(s)) ∧ pm'states(s)
          ⊃
          (∇ (addr1: Address):
            OK?(ac'free(pm)(addr1)(s)) ⊃
            OK?[State, Unit](ac'free(pm)(addr1)(s)) ∨
            Exception?[State, Unit](ac'free(pm)(addr1)(s))))

```

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of allocator?_TCC2.

Q.E.D.

C.10.4 Allocator.allocator?_TCC3

Terse proof for allocator?_TCC3.

allocator?_TCC3:

```

{1}  ∀ (pm: (plain_memory?[State]), ac: Allocator):
      (ac'private_mem(pm) ⊆ (pm'ro_addr ∪ pm'rw_addr)) ⊃
      (∀ (s: State):
        (∀ (addr: Address):
          OK?(ac'free(pm)(addr)(s)) ≡ up?(ac'freed_size(pm)(addr)(s)))
        ∧
        (∀ (size: posnat):
          OK?(ac'alloc(pm)(size)(s)) ∧
          ¬ data(ac'alloc(pm)(size)(s)) = null_address
          ⊃
          disjoint?(ac'allocated(pm)(s),
                    address_block(data(ac'alloc(pm)(size)(s)), size))
          ∧
          ac'allocated(pm)(state(ac'alloc(pm)(size)(s))) =
            (ac'allocated(pm)(s) ∪ address_block(data(ac'alloc(pm)(size)(s)), size))
          ∧
          ac'memory_pool(pm)(state(ac'alloc(pm)(size)(s))) =
            ac'memory_pool(pm)(s))
        ∧ (ac'allocated(pm)(s) ⊆ ac'memory_pool(pm)(s)) ∧ pm'states(s)
        ⊃
        (∀ (addr1: Address):
          OK?(ac'free(pm)(addr1)(s)) ⊃
          up?[posnat](ac'freed_size(pm)(addr1)(s))))

```

Repeatedly Skolemizing and flattening,

Rewriting using -3, matching in *,

This completes the proof of allocator?_TCC3.

Q.E.D.

C.10.5 Allocator.allocator?_TCC4

Terse proof for allocator?_TCC4.

allocator?_TCC4:

```

{1}  ∀ (pm: (plain_memory?[State]), ac: Allocator):
      (ac'private_mem(pm) ⊆ (pm'ro_addr ∪ pm'rw_addr)) ⊃
      (∀ (s: State):
        (∀ (s2: State):
          pm'states(s2) ∧ stays_unchanged(pm)(s, s2, ac'private_mem(pm)) ⊃
          ac'memory_pool(pm)(s) = ac'memory_pool(pm)(s2) ∧
          ac'allocated(pm)(s) = ac'allocated(pm)(s2))
        ∧
        (∀ (addr: Address):
          OK?(ac'free(pm)(addr)(s)) ⊃
          ac'allocated(pm)(state(ac'free(pm)(addr)(s))) =
          (ac'allocated(pm)(s) \ address_block(addr, down(ac'freed_size(pm)(addr)(s))))
          ∧
          ac'memory_pool(pm)(state(ac'free(pm)(addr)(s))) =
          ac'memory_pool(pm)(s))
        ∧
        (∀ (addr: Address):
          OK?(ac'free(pm)(addr)(s)) ≡ up?(ac'freed_size(pm)(addr)(s)))
        ∧
        (∀ (size: posnat):
          OK?(ac'alloc(pm)(size)(s)) ∧
          ¬ data(ac'alloc(pm)(size)(s)) = null_address
          ⊃
          disjoint?(ac'allocated(pm)(s),
                    address_block(data(ac'alloc(pm)(size)(s)), size))
          ∧
          ac'allocated(pm)(state(ac'alloc(pm)(size)(s))) =
          (ac'allocated(pm)(s) ∪ address_block(data(ac'alloc(pm)(size)(s)), size))
          ∧
          ac'memory_pool(pm)(state(ac'alloc(pm)(size)(s))) =
          ac'memory_pool(pm)(s))
        ∧ (ac'allocated(pm)(s) ⊆ ac'memory_pool(pm)(s)) ∧ pm'states(s)
        ⊃
        (∀ (size1: posnat):
          has_next_state(ac'alloc(pm)(size1)(s)) ⊃
          OK?[State, Address](ac'alloc(pm)(size1)(s)) ∨
          Exception?[State, Address](ac'alloc(pm)(size1)(s))))

```

Repeatedly Skolemizing and flattening,

Expanding the definition of has_next_state,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of allocator?_TCC4.

Q.E.D.

C.10.6 Allocator.allocator?_TCC5

Terse proof for allocator?_TCC5.

allocator?_TCC5:

$$\begin{array}{l}
 \{1\} \quad \forall (\text{pm}: (\text{plain_memory?}[\text{State}])), \text{ac}: \text{Allocator}: \\
 (\text{ac}'\text{private_mem}(\text{pm}) \subseteq (\text{pm}'\text{ro_addr} \cup \text{pm}'\text{rw_addr})) \supset \\
 (\forall (s: \text{State}): \\
 (\forall (\text{size}: \text{posnat}): \\
 \text{has_next_state}(\text{ac}'\text{alloc}(\text{pm})(\text{size})(s)) \supset \\
 \text{pm}'\text{states}(\text{state}(\text{ac}'\text{alloc}(\text{pm})(\text{size})(s)))) \\
 \wedge \\
 (\forall (s_2: \text{State}): \\
 \text{pm}'\text{states}(s_2) \wedge \text{stays_unchanged}(\text{pm})(s, s_2, \text{ac}'\text{private_mem}(\text{pm})) \supset \\
 \text{ac}'\text{memory_pool}(\text{pm})(s) = \text{ac}'\text{memory_pool}(\text{pm})(s_2) \wedge \\
 \text{ac}'\text{allocated}(\text{pm})(s) = \text{ac}'\text{allocated}(\text{pm})(s_2)) \\
 \wedge \\
 (\forall (\text{addr}: \text{Address}): \\
 \text{OK?}(\text{ac}'\text{free}(\text{pm})(\text{addr})(s)) \supset \\
 \text{ac}'\text{allocated}(\text{pm})(\text{state}(\text{ac}'\text{free}(\text{pm})(\text{addr})(s))) = \\
 (\text{ac}'\text{allocated}(\text{pm})(s) \setminus \text{address_block}(\text{addr}, \text{down}(\text{ac}'\text{freed_size}(\text{pm})(\text{addr})(s)))) \\
 \wedge \\
 \text{ac}'\text{memory_pool}(\text{pm})(\text{state}(\text{ac}'\text{free}(\text{pm})(\text{addr})(s))) = \\
 \text{ac}'\text{memory_pool}(\text{pm})(s)) \\
 \wedge \\
 (\forall (\text{addr}: \text{Address}): \\
 \text{OK?}(\text{ac}'\text{free}(\text{pm})(\text{addr})(s)) \equiv \text{up?}(\text{ac}'\text{freed_size}(\text{pm})(\text{addr})(s))) \\
 \wedge \\
 (\forall (\text{size}: \text{posnat}): \\
 \text{OK?}(\text{ac}'\text{alloc}(\text{pm})(\text{size})(s)) \wedge \\
 \neg \text{data}(\text{ac}'\text{alloc}(\text{pm})(\text{size})(s)) = \text{null_address} \\
 \supset \\
 \text{disjoint?}(\text{ac}'\text{allocated}(\text{pm})(s), \\
 \text{address_block}(\text{data}(\text{ac}'\text{alloc}(\text{pm})(\text{size})(s)), \text{size})) \\
 \wedge \\
 \text{ac}'\text{allocated}(\text{pm})(\text{state}(\text{ac}'\text{alloc}(\text{pm})(\text{size})(s))) = \\
 (\text{ac}'\text{allocated}(\text{pm})(s) \cup \text{address_block}(\text{data}(\text{ac}'\text{alloc}(\text{pm})(\text{size})(s)), \text{size})) \\
 \wedge \\
 \text{ac}'\text{memory_pool}(\text{pm})(\text{state}(\text{ac}'\text{alloc}(\text{pm})(\text{size})(s))) = \\
 \text{ac}'\text{memory_pool}(\text{pm})(s)) \\
 \wedge (\text{ac}'\text{allocated}(\text{pm})(s) \subseteq \text{ac}'\text{memory_pool}(\text{pm})(s)) \wedge \text{pm}'\text{states}(s) \\
 \supset \\
 (\forall (\text{addr1}: \text{Address}): \\
 \text{has_next_state}(\text{ac}'\text{free}(\text{pm})(\text{addr1})(s)) \supset \\
 \text{OK?}[\text{State}, \text{Unit}](\text{ac}'\text{free}(\text{pm})(\text{addr1})(s)) \vee \\
 \text{Exception?}[\text{State}, \text{Unit}](\text{ac}'\text{free}(\text{pm})(\text{addr1})(s))))
 \end{array}$$

Repeatedly Skolemizing and flattening,

Expanding the definition of has_next_state,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of allocator?_TCC5.

Q.E.D.

C.10.7 Allocator.allocatort_TCC6

Terse proof for allocatort_TCC6.

allocatort_TCC6:

```

{1}  ∇ (pm: (plain_memory?[State]), ac: Allocator):
      (ac'private_mem(pm) ⊆ (pm'ro_addr ∪ pm'rw_addr)) ⊃
      (∇ (s: State):
        (∇ (addr: Address):
          has_next_state(ac'free(pm)(addr)(s)) ⊃
          pm'states(state(ac'free(pm)(addr)(s))))
        ∧
        (∇ (size: posnat):
          has_next_state(ac'alloc(pm)(size)(s)) ⊃
          pm'states(state(ac'alloc(pm)(size)(s))))
        ∧
        (∇ (s2: State):
          pm'states(s2) ∧ stays_unchanged(pm)(s, s2, ac'private_mem(pm)) ⊃
          ac'memory_pool(pm)(s) = ac'memory_pool(pm)(s2) ∧
          ac'allocated(pm)(s) = ac'allocated(pm)(s2))
        ∧
        (∇ (addr: Address):
          OK?(ac'free(pm)(addr)(s)) ⊃
          ac'allocated(pm)(state(ac'free(pm)(addr)(s))) =
          (ac'allocated(pm)(s) \ address_block(addr, down(ac'freed_size(pm)(addr)(s))))
          ∧
          ac'memory_pool(pm)(state(ac'free(pm)(addr)(s))) =
          ac'memory_pool(pm)(s))
        ∧
        (∇ (addr: Address):
          OK?(ac'free(pm)(addr)(s)) ≡ up?(ac'freed_size(pm)(addr)(s)))
        ∧
        (∇ (size: posnat):
          OK?(ac'alloc(pm)(size)(s)) ∧
          ¬ data(ac'alloc(pm)(size)(s)) = null_address
          ⊃
          disjoint?(ac'allocated(pm)(s),
                    address_block(data(ac'alloc(pm)(size)(s)), size))
          ∧
          ac'allocated(pm)(state(ac'alloc(pm)(size)(s))) =
          (ac'allocated(pm)(s) ∪ address_block(data(ac'alloc(pm)(size)(s)), size))
          ∧
          ac'memory_pool(pm)(state(ac'alloc(pm)(size)(s))) =
          ac'memory_pool(pm)(s)
          ∧ (ac'allocated(pm)(s) ⊆ ac'memory_pool(pm)(s)) ∧ pm'states(s)
        ⊃
        (∇ (size1: posnat):
          OK?(ac'alloc(pm)(size1)(s)) ∧
          ¬ data(ac'alloc(pm)(size1)(s)) = null_address
          ⊃
          OK?[State, Address](ac'alloc(pm)(size1)(s)) ∨
          Exception?[State, Address](ac'alloc(pm)(size1)(s))))

```

C Proof scripts

Repeatedly Skolemizing and flattening,

This completes the proof of `allocator?_TCC6`.

Q.E.D.

C.10.8 Allocator.allocator?_TCC7

Terse proof for `allocator?_TCC7`.

allocator?_TCC7:

```

{1}  ∀ (pm: (plain_memory?[State]), ac: Allocator):
      (ac'private_mem(pm) ⊆ (pm'ro_addr ∪ pm'rw_addr)) ⊃
      (∀ (s: State):
        (∀ (addr: Address):
          has_next_state(ac'free(pm)(addr)(s)) ⊃
            pm'states(state(ac'free(pm)(addr)(s))))
        ∧
        (∀ (size: posnat):
          has_next_state(ac'alloc(pm)(size)(s)) ⊃
            pm'states(state(ac'alloc(pm)(size)(s))))
        ∧
        (∀ (s2: State):
          pm'states(s2) ∧ stays_unchanged(pm)(s, s2, ac'private_mem(pm)) ⊃
            ac'memory_pool(pm)(s) = ac'memory_pool(pm)(s2) ∧
            ac'allocated(pm)(s) = ac'allocated(pm)(s2))
        ∧
        (∀ (addr: Address):
          OK?(ac'free(pm)(addr)(s)) ⊃
            ac'allocated(pm)(state(ac'free(pm)(addr)(s))) =
              (ac'allocated(pm)(s) \ address_block(addr, down(ac'freed_size(pm)(addr)(s))))
          ∧
            ac'memory_pool(pm)(state(ac'free(pm)(addr)(s))) =
              ac'memory_pool(pm)(s))
        ∧
        (∀ (addr: Address):
          OK?(ac'free(pm)(addr)(s)) ≡ up?(ac'freed_size(pm)(addr)(s))
        ∧
        (∀ (size: posnat):
          OK?(ac'alloc(pm)(size)(s)) ∧
            ¬ data(ac'alloc(pm)(size)(s)) = null_address
          ⊃
            disjoint?(ac'allocated(pm)(s),
              address_block(data(ac'alloc(pm)(size)(s)), size))
          ∧
            ac'allocated(pm)(state(ac'alloc(pm)(size)(s))) =
              (ac'allocated(pm)(s) ∪ address_block(data(ac'alloc(pm)(size)(s)), size))
          ∧
            ac'memory_pool(pm)(state(ac'alloc(pm)(size)(s))) =
              ac'memory_pool(pm)(s)
          ∧ (ac'allocated(pm)(s) ⊆ ac'memory_pool(pm)(s)) ∧ pm'states(s)
        ⊃
        (∀ (size1: posnat):
          OK?(ac'alloc(pm)(size1)(s)) ∧
            ¬ data(ac'alloc(pm)(size1)(s)) = null_address
          ⊃ pm'states(state[State, Address](ac'alloc(pm)(size1)(s))))

```

Repeatedly Skolemizing and flattening,

Instantiating quantified variables,

Expanding the definition of has_next_state,

Simplifying, rewriting, and recording with decision procedures,

C Proof scripts

This completes the proof of `allocator?_TCC7`.

Q.E.D.

C.10.9 Allocator.allocator?_TCC8

Terse proof for `allocator?_TCC8`.

allocator?_TCC8:

```

{1}  ∀ (pm: (plain_memory?[State]), ac: Allocator):
      (ac'private_mem(pm) ⊆ (pm'ro_addr ∪ pm'rw_addr)) ⊃
      (∀ (s: State):
        (∀ (size: posnat):
          OK?(ac'alloc(pm)(size)(s)) ∧
          ¬ data(ac'alloc(pm)(size)(s)) = null_address
          ⊃
          only_changes(pm)(s, state(ac'alloc(pm)(size)(s)), ac'private_mem(pm)))
        ∧
        (∀ (addr: Address):
          has_next_state(ac'free(pm)(addr)(s)) ⊃
          pm'states(state(ac'free(pm)(addr)(s))))
        ∧
        (∀ (size: posnat):
          has_next_state(ac'alloc(pm)(size)(s)) ⊃
          pm'states(state(ac'alloc(pm)(size)(s))))
        ∧
        (∀ (s2: State):
          pm'states(s2) ∧ stays_unchanged(pm)(s, s2, ac'private_mem(pm)) ⊃
          ac'memory_pool(pm)(s) = ac'memory_pool(pm)(s2) ∧
          ac'allocated(pm)(s) = ac'allocated(pm)(s2))
        ∧
        (∀ (addr: Address):
          OK?(ac'free(pm)(addr)(s)) ⊃
          ac'allocated(pm)(state(ac'free(pm)(addr)(s))) =
          (ac'allocated(pm)(s) \ address_block(addr, down(ac'freed_size(pm)(addr)(s))))
          ∧
          ac'memory_pool(pm)(state(ac'free(pm)(addr)(s))) =
          ac'memory_pool(pm)(s))
        ∧
        (∀ (addr: Address):
          OK?(ac'free(pm)(addr)(s)) ≡ up?(ac'freed_size(pm)(addr)(s)))
        ∧
        (∀ (size: posnat):
          OK?(ac'alloc(pm)(size)(s)) ∧
          ¬ data(ac'alloc(pm)(size)(s)) = null_address
          ⊃
          disjoint?(ac'allocated(pm)(s),
            address_block(data(ac'alloc(pm)(size)(s)), size))
          ∧
          ac'allocated(pm)(state(ac'alloc(pm)(size)(s))) =
          (ac'allocated(pm)(s) ∪ address_block(data(ac'alloc(pm)(size)(s)), size))
          ∧
          ac'memory_pool(pm)(state(ac'alloc(pm)(size)(s))) =
          ac'memory_pool(pm)(s))
          ∧ (ac'allocated(pm)(s) ⊆ ac'memory_pool(pm)(s)) ∧ pm'states(s)
        ⊃
        (∀ (free_addr: Address):
          OK?(ac'free(pm)(free_addr)(s)) ⊃
          pm'states(state[State, Unit](ac'free(pm)(free_addr)(s))))
      )

```


Repeatedly Skolemizing and flattening,
 Instantiating quantified variables,
 Expanding the definition of `has_next_state`,
 Simplifying, rewriting, and recording with decision procedures,
 This completes the proof of `allocator?_TCC8`.
 Q.E.D.

C.10.10 `Allocator.allocator_subset_private_memory_plain_memory`

Terse proof for `allocator_subset_private_memory_plain_memory`.
`allocator_subset_private_memory_plain_memory`:

$$\{1\} \quad \forall (pm: (plain_memory?), ac: (allocator?(pm))):$$

$$(ac'private_mem(pm) \subseteq (pm'ro_addr \cup pm'rw_addr))$$

Repeatedly Skolemizing and flattening,
 Expanding the definition of `allocator?`,
 Simplifying, rewriting, and recording with decision procedures,
 This completes the proof of `allocator_subset_private_memory_plain_memory`.
 Q.E.D.

C.10.11 `Allocator.allocator_private_memory_in_plain_memory`

Terse proof for `allocator_private_memory_in_plain_memory`.
`allocator_private_memory_in_plain_memory`:

$$\{1\} \quad \forall (pm: (plain_memory?), ac: (allocator?(pm)), addr: Address):$$

$$ac'private_mem(pm)(addr) \supset union(pm'ro_addr, pm'rw_addr)(addr)$$

Repeatedly Skolemizing and flattening,
 Using lemma `allocator_subset_private_memory_plain_memory`,
 Using lemma `subset_member`,
 Simplifying, rewriting, and recording with decision procedures,
 This completes the proof of `allocator_private_memory_in_plain_memory`.
 Q.E.D.

C.10.12 `Allocator.allocator_subset_allocated_memory_pool`

Terse proof for `allocator_subset_allocated_memory_pool`.
`allocator_subset_allocated_memory_pool`:

$$\{1\} \quad \forall (pm: (plain_memory?), ac: (allocator?(pm)), s: State):$$

$$pm'states(s) \supset (ac'allocated(pm)(s) \subseteq ac'memory_pool(pm)(s))$$

Repeatedly Skolemizing and flattening,
 Expanding the definition of `allocator?`,
 Applying disjunctive simplification to flatten sequent,
 Instantiating quantified variables,
 Simplifying, rewriting, and recording with decision procedures,
 This completes the proof of `allocator_subset_allocated_memory_pool`.
 Q.E.D.

C.10.13 Allocator.allocator_alloc_disjoint

Terse proof for `allocator_alloc_disjoint`.

`allocator_alloc_disjoint`:

$\{1\} \quad \forall (pm: (\text{plain_memory?}), ac: (\text{allocator?}(pm)), size: \text{posnat}, s: \text{State}):$ $pm' \text{states}(s) \wedge$ $OK?(ac' \text{alloc}(pm)(size)(s)) \wedge \neg \text{data}(ac' \text{alloc}(pm)(size)(s)) = \text{null_address}$ \supset $\text{disjoint?}(ac' \text{allocated}(pm)(s),$ $\quad \text{address_block}(\text{data}(ac' \text{alloc}(pm)(size)(s)), size))$

Repeatedly Skolemizing and flattening,
 Expanding the definition of `allocator?`,
 Applying disjunctive simplification to flatten sequent,
 Instantiating the top quantifier in -3 with the terms: s' ,
 Simplifying, rewriting, and recording with decision procedures,
 Applying disjunctive simplification to flatten sequent,
 Instantiating quantified variables,
 Simplifying, rewriting, and recording with decision procedures,
 This completes the proof of `allocator_alloc_disjoint`.
 Q.E.D.

C.10.14 Allocator.allocator_allocated_alloc_TCC1

Terse proof for `allocator_allocated_alloc_TCC1`.

`allocator_allocated_alloc_TCC1`:

$\{1\} \quad \forall (pm: (\text{plain_memory?}[State]), ac: (\text{allocator?}(pm)), size: \text{posnat}, s: \text{State}):$ $pm' \text{states}(s) \wedge$ $OK?(ac' \text{alloc}(pm)(size)(s)) \wedge \neg \text{data}(ac' \text{alloc}(pm)(size)(s)) = \text{null_address}$ \supset $OK?[State, Address](ac' \text{alloc}(pm)(size)(s)) \vee$ $\text{Exception?}[State, Address](ac' \text{alloc}(pm)(size)(s))$
--

Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of `allocator_allocated_alloc_TCC1`.
 Q.E.D.

C.10.15 Allocator.allocator_allocated_alloc

Terse proof for `allocator_allocated_alloc`.

`allocator_allocated_alloc`:

$\{1\} \quad \forall (pm: (\text{plain_memory?}), ac: (\text{allocator?}(pm)), size: \text{posnat}, s: \text{State}):$ $pm' \text{states}(s) \wedge$ $OK?(ac' \text{alloc}(pm)(size)(s)) \wedge \neg \text{data}(ac' \text{alloc}(pm)(size)(s)) = \text{null_address}$ \supset $ac' \text{allocated}(pm)(\text{state}(ac' \text{alloc}(pm)(size)(s))) =$ $(\text{ac' allocated}(pm)(s) \cup \text{address_block}(\text{data}(ac' \text{alloc}(pm)(size)(s)), size))$

Repeatedly Skolemizing and flattening,

Expanding the definition of `allocator?`,
 Applying disjunctive simplification to flatten sequent,
 Instantiating the top quantifier in -3 with the terms: s' ,
 Simplifying, rewriting, and recording with decision procedures,
 Applying disjunctive simplification to flatten sequent,
 Instantiating quantified variables,
 Simplifying, rewriting, and recording with decision procedures,
 This completes the proof of `allocator_allocated_alloc`.
 Q.E.D.

C.10.16 `Allocator.allocator_memory_pool_alloc`

Terse proof for `allocator_memory_pool_alloc`.

`allocator_memory_pool_alloc`:

$\{1\} \quad \forall (pm: (plain_memory?), ac: (allocator?(pm)), size: posnat, s: State):$ $pm'states(s) \wedge$ $OK?(ac'alloc(pm)(size)(s)) \wedge \neg data(ac'alloc(pm)(size)(s)) = null_address$ $\supset ac'memory_pool(pm)(state(ac'alloc(pm)(size)(s))) = ac'memory_pool(pm)(s)$

Repeatedly Skolemizing and flattening,
 Expanding the definition of `allocator?`,
 Applying disjunctive simplification to flatten sequent,
 Instantiating the top quantifier in -3 with the terms: s' ,
 Simplifying, rewriting, and recording with decision procedures,
 Applying disjunctive simplification to flatten sequent,
 Instantiating quantified variables,
 Simplifying, rewriting, and recording with decision procedures,
 This completes the proof of `allocator_memory_pool_alloc`.
 Q.E.D.

C.10.17 `Allocator.allocator_subset_address_block_alloc_allocated`

Terse proof for `allocator_subset_address_block_alloc_allocated`.

`allocator_subset_address_block_alloc_allocated`:

$\{1\} \quad \forall (pm: (plain_memory?), ac: (allocator?(pm)), size: posnat, addr: Ad-$ $dress, s: State):$ $pm'states(s) \wedge$ $OK?(ac'alloc(pm)(size)(s)) \wedge \neg data(ac'alloc(pm)(size)(s)) = null_address$ \supset $(address_block(data(ac'alloc(pm)(size)(s)), size) \subseteq ac'allocated(pm)(state(ac'alloc(pm)(size)(s))))$
--

Repeatedly Skolemizing and flattening,
 Using lemma `allocator_allocated_alloc`,
 Simplifying, rewriting, and recording with decision procedures,
 Applying `union_subset3` where a gets `address_block(data(ac!1'alloc(pm!1)(size!1)(s!1)), size!1)`, b gets `ac!1'allocated(pm!1)(s!1)`,
 Simplifying, rewriting, and recording with decision procedures,
 This completes the proof of `allocator_subset_address_block_alloc_allocated`.
 Q.E.D.

C.10.18 Allocator.alloc_free_freed_size

Terse proof for `allocator_free_freed_size`.

`allocator_free_freed_size`:

$$\frac{\{1\} \quad \forall (pm: (plain_memory?), ac: (allocator?(pm)), addr: Address, s: State):}{\quad pm'states(s) \supset (OK?(ac'free(pm)(addr)(s)) \equiv up?(ac'freed_size(pm)(addr)(s)))}$$

Repeatedly Skolemizing and flattening,
 Expanding the definition of `allocator?`,
 Applying disjunctive simplification to flatten sequent,
 Instantiating quantified variables,
 Simplifying, rewriting, and recording with decision procedures,
 Applying disjunctive simplification to flatten sequent,
 Instantiating quantified variables,
 This completes the proof of `allocator_free_freed_size`.
 Q.E.D.

C.10.19 Allocator.alloc_allocated_free_TCC1

Terse proof for `allocator_allocated_free_TCC1`.

`allocator_allocated_free_TCC1`:

$$\frac{\{1\} \quad \forall (pm: (plain_memory?[State]), ac: (allocator?(pm)), addr: Address, s: State):}{\quad pm'states(s) \wedge OK?(ac'free(pm)(addr)(s)) \supset (OK?[State, Unit](ac'free(pm)(addr)(s)) \vee Exception?[State, Unit](ac'free(pm)(addr)(s)))}$$

Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of `allocator_allocated_free_TCC1`.
 Q.E.D.

C.10.20 Allocator.alloc_allocated_free_TCC2

Terse proof for `allocator_allocated_free_TCC2`.

`allocator_allocated_free_TCC2`:

$$\frac{\{1\} \quad \forall (pm: (plain_memory?[State]), ac: (allocator?(pm)), addr: Address, s: State):}{\quad pm'states(s) \wedge OK?(ac'free(pm)(addr)(s)) \supset up?[posnat](ac'freed_size(pm)(addr)(s))}$$

Repeatedly Skolemizing and flattening,
 Rewriting using `allocator_free_freed_size`, matching in *,
 This completes the proof of `allocator_allocated_free_TCC2`.
 Q.E.D.

C.10.21 Allocator.alloc_allocated_free

Terse proof for `allocator_allocated_free`.

allocator_allocated_free:

{1} \forall (pm: (plain_memory?), ac: (allocator?(pm)), addr: Address, s: State):
 $pm\text{'states}(s) \wedge OK?(ac\text{'free}(pm)(addr)(s)) \supset$
 $ac\text{'allocated}(pm)(state(ac\text{'free}(pm)(addr)(s))) =$
 $(ac\text{'allocated}(pm)(s) \setminus address_block(addr, down(ac\text{'freed_size}(pm)(addr)(s))))$

Repeatedly Skolemizing and flattening,
 Expanding the definition of allocator?,
 Applying disjunctive simplification to flatten sequent,
 Instantiating the top quantifier in -3 with the terms: s' ,
 Simplifying, rewriting, and recording with decision procedures,
 Applying disjunctive simplification to flatten sequent,
 Instantiating quantified variables,
 Simplifying, rewriting, and recording with decision procedures,
 This completes the proof of allocator_allocated_free.
 Q.E.D.

C.10.22 Allocator.allocator_memory_pool_free

Terse proof for allocator_memory_pool_free.

allocator_memory_pool_free:

{1} \forall (pm: (plain_memory?), ac: (allocator?(pm)), addr: Address, s: State):
 $pm\text{'states}(s) \wedge OK?(ac\text{'free}(pm)(addr)(s)) \supset$
 $ac\text{'memory_pool}(pm)(state(ac\text{'free}(pm)(addr)(s))) = ac\text{'memory_pool}(pm)(s)$

Repeatedly Skolemizing and flattening,
 Expanding the definition of allocator?,
 Applying disjunctive simplification to flatten sequent,
 Instantiating the top quantifier in -3 with the terms: s' ,
 Simplifying, rewriting, and recording with decision procedures,
 Applying disjunctive simplification to flatten sequent,
 Instantiating quantified variables,
 Simplifying, rewriting, and recording with decision procedures,
 This completes the proof of allocator_memory_pool_free.
 Q.E.D.

C.10.23 Allocator.allocator_const_private_memory_memory_pool

Terse proof for allocator_const_private_memory_memory_pool.

allocator_const_private_memory_memory_pool:

{1} \forall (pm: (plain_memory?), ac: (allocator?(pm)), s_1, s_2 : State):
 $pm\text{'states}(s_1) \wedge pm\text{'states}(s_2) \wedge stays_unchanged(pm)(s_1, s_2, ac\text{'private_mem}(pm)) \supset$
 $ac\text{'memory_pool}(pm)(s_1) = ac\text{'memory_pool}(pm)(s_2)$

Repeatedly Skolemizing and flattening,
 Expanding the definition of allocator?,
 Applying disjunctive simplification to flatten sequent,
 Instantiating the top quantifier in -3 with the terms: s'_1 ,
 Simplifying, rewriting, and recording with decision procedures,

Applying disjunctive simplification to flatten sequent,
 Instantiating quantified variables,
 Simplifying, rewriting, and recording with decision procedures,
 This completes the proof of `allocator_const_private_memory_memory_pool`.
 Q.E.D.

C.10.24 Allocator.allocator_const_private_memory_allocated

Terse proof for `allocator_const_private_memory_allocated`.
`allocator_const_private_memory_allocated:`

$$\frac{\{1\} \quad \forall (pm: (plain_memory?), ac: (allocator?(pm)), s_1, s_2: State): \quad pm'states(s_1) \wedge pm'states(s_2) \wedge stays_unchanged(pm)(s_1, s_2, ac'private_mem(pm)) \supset ac'allocated(pm)(s_1) = ac'allocated(pm)(s_2)}{}$$

Repeatedly Skolemizing and flattening,
 Expanding the definition of `allocator?`,
 Applying disjunctive simplification to flatten sequent,
 Instantiating the top quantifier in -3 with the terms: s'_1 ,
 Simplifying, rewriting, and recording with decision procedures,
 Applying disjunctive simplification to flatten sequent,
 Instantiating the top quantifier in -7 with the terms: s'_2 ,
 Simplifying, rewriting, and recording with decision procedures,
 This completes the proof of `allocator_const_private_memory_allocated`.
 Q.E.D.

C.10.25 Allocator.allocator_next_state_alloc_TCC1

Terse proof for `allocator_next_state_alloc_TCC1`.
`allocator_next_state_alloc_TCC1:`

$$\frac{\{1\} \quad \forall (pm: (plain_memory?[State]), ac: (allocator?(pm)), size: posnat, s: State): \quad pm'states(s) \wedge has_next_state(ac'alloc(pm)(size)(s)) \supset \quad OK?[State, Address](ac'alloc(pm)(size)(s)) \vee \quad Exception?[State, Address](ac'alloc(pm)(size)(s))}{}$$

Repeatedly Skolemizing and flattening,
 Expanding the definition of `has_next_state`,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 This completes the proof of `allocator_next_state_alloc_TCC1`.
 Q.E.D.

C.10.26 Allocator.allocator_next_state_alloc

Terse proof for `allocator_next_state_alloc`.
`allocator_next_state_alloc:`

$$\frac{\{1\} \quad \forall (pm: (plain_memory?), ac: (allocator?(pm)), size: posnat, s: State): \quad pm'states(s) \wedge has_next_state(ac'alloc(pm)(size)(s)) \supset \quad pm'states(state(ac'alloc(pm)(size)(s)))}{}$$

Repeatedly Skolemizing and flattening,

Expanding the definition of `allocator?`,
 Applying disjunctive simplification to flatten sequent,
 Instantiating quantified variables,
 Simplifying, rewriting, and recording with decision procedures,
 Applying disjunctive simplification to flatten sequent,
 Instantiating quantified variables,
 Simplifying, rewriting, and recording with decision procedures,
 This completes the proof of `allocator_next_state_alloc`.
 Q.E.D.

C.10.27 `Allocator.allocator_next_state_free_TCC1`

Terse proof for `allocator_next_state_free_TCC1`.

`allocator_next_state_free_TCC1`:

$\{1\} \quad \forall (pm: (\text{plain_memory?}[State]), ac: (\text{allocator?}(pm)), addr: \text{Address}, s: \text{State}):$ $pm \text{'states}(s) \wedge \text{has_next_state}(ac \text{'free}(pm)(addr)(s)) \supset$ $\text{OK?}[State, \text{Unit}](ac \text{'free}(pm)(addr)(s)) \vee$ $\text{Exception?}[State, \text{Unit}](ac \text{'free}(pm)(addr)(s))$

Repeatedly Skolemizing and flattening,
 Expanding the definition of `has_next_state`,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 This completes the proof of `allocator_next_state_free_TCC1`.
 Q.E.D.

C.10.28 `Allocator.allocator_next_state_free`

Terse proof for `allocator_next_state_free`.

`allocator_next_state_free`:

$\{1\} \quad \forall (pm: (\text{plain_memory?}), ac: (\text{allocator?}(pm)), addr: \text{Address}, s: \text{State}):$ $pm \text{'states}(s) \wedge \text{has_next_state}(ac \text{'free}(pm)(addr)(s)) \supset$ $pm \text{'states}(\text{state}(ac \text{'free}(pm)(addr)(s)))$
--

Repeatedly Skolemizing and flattening,
 Expanding the definition of `allocator?`,
 Applying disjunctive simplification to flatten sequent,
 Instantiating quantified variables,
 Simplifying, rewriting, and recording with decision procedures,
 Applying disjunctive simplification to flatten sequent,
 Instantiating quantified variables,
 Simplifying, rewriting, and recording with decision procedures,
 This completes the proof of `allocator_next_state_free`.
 Q.E.D.

C.10.29 `Allocator.allocator_only_changes_alloc_TCC1`

Terse proof for `allocator_only_changes_alloc_TCC1`.

allocator_only_changes_alloc_TCC1:

$$\{1\} \quad \forall (pm: (\text{plain_memory?}[State]), ac: (\text{allocator?}(pm)), size: \text{posnat}, s: State):$$

$$pm' \text{states}(s) \wedge$$

$$OK?(ac' \text{alloc}(pm)(size)(s)) \wedge \neg \text{data}(ac' \text{alloc}(pm)(size)(s)) = \text{null_address}$$

$$\supset pm' \text{states}(\text{state}[State, Address](ac' \text{alloc}(pm)(size)(s)))$$

Repeatedly Skolemizing and flattening,
 Rewriting using allocator_next_state_alloc, matching in *,
 Expanding the definition of has_next_state,
 which is trivially true.
 This completes the proof of allocator_only_changes_alloc_TCC1.
 Q.E.D.

C.10.30 Allocator.allocator_only_changes_alloc

Terse proof for allocator_only_changes_alloc.

allocator_only_changes_alloc:

$$\{1\} \quad \forall (pm: (\text{plain_memory?}), ac: (\text{allocator?}(pm)), size: \text{posnat}, s: State):$$

$$pm' \text{states}(s) \wedge$$

$$OK?(ac' \text{alloc}(pm)(size)(s)) \wedge \neg \text{data}(ac' \text{alloc}(pm)(size)(s)) = \text{null_address}$$

$$\supset \text{only_changes}(pm)(s, \text{state}(ac' \text{alloc}(pm)(size)(s)), ac' \text{private_mem}(pm))$$

Repeatedly Skolemizing and flattening,
 Expanding the definition of allocator?,
 Applying disjunctive simplification to flatten sequent,
 Instantiating quantified variables,
 Simplifying, rewriting, and recording with decision procedures,
 Applying disjunctive simplification to flatten sequent,
 Instantiating quantified variables,
 Simplifying, rewriting, and recording with decision procedures,
 This completes the proof of allocator_only_changes_alloc.
 Q.E.D.

C.10.31 Allocator.allocator_only_changes_free_TCC1

Terse proof for allocator_only_changes_free_TCC1.

allocator_only_changes_free_TCC1:

$$\{1\} \quad \forall (pm: (\text{plain_memory?}[State]), ac: (\text{allocator?}(pm)), free_addr: \text{Ad-}$$

$$\text{dress}, s: State):$$

$$pm' \text{states}(s) \wedge OK?(ac' \text{free}(pm)(free_addr)(s)) \supset$$

$$pm' \text{states}(\text{state}[State, Unit](ac' \text{free}(pm)(free_addr)(s)))$$

Repeatedly Skolemizing and flattening,
 Rewriting using allocator_next_state_free, matching in *,
 Expanding the definition of has_next_state,
 which is trivially true.
 This completes the proof of allocator_only_changes_free_TCC1.
 Q.E.D.

C.10.32 Allocator.allocator_only_changes_free

Terse proof for `allocator_only_changes_free`.

`allocator_only_changes_free`:

$\{1\} \quad \forall (pm: (\text{plain_memory?}), ac: (\text{allocator?}(pm)), \text{free_addr}: \text{Address}, s: \text{State}):$ $pm' \text{states}(s) \wedge \text{OK?}(ac' \text{free}(pm)(\text{free_addr})(s)) \supset$ $\text{only_changes}(pm)(s, \text{state}(ac' \text{free}(pm)(\text{free_addr})(s)), ac' \text{private_mem}(pm))$

Repeatedly Skolemizing and flattening,
 Expanding the definition of `allocator?`,
 Applying disjunctive simplification to flatten sequent,
 Instantiating quantified variables,
 Simplifying, rewriting, and recording with decision procedures,
 Applying disjunctive simplification to flatten sequent,
 Instantiating quantified variables,
 Simplifying, rewriting, and recording with decision procedures,
 This completes the proof of `allocator_only_changes_free`.
 Q.E.D.

C.10.33 Allocator.allocator_allocated_memory_read_TCC1

Terse proof for `allocator_allocated_memory_read_TCC1`.

`allocator_allocated_memory_read_TCC1`:

$\{1\} \quad \forall (pm: (\text{plain_memory?}[State]), ac: (\text{allocator?}(pm)), \text{addr}: \text{Address}, s: \text{State}):$ $pm' \text{states}(s) \wedge \text{union}(pm' \text{ro_addr}, pm' \text{rw_addr})(\text{addr}) \supset$ $\text{OK?}[State, \text{Byte}](\text{memory_read}(pm' \text{mem})(\text{addr})(s)) \vee$ $\text{Exception?}[State, \text{Byte}](\text{memory_read}(pm' \text{mem})(\text{addr})(s))$
--

Repeatedly Skolemizing and flattening,
 Rewriting using `plain_memory_memory_read_ok`, matching in *,
 This completes the proof of `allocator_allocated_memory_read_TCC1`.
 Q.E.D.

C.10.34 Allocator.allocator_allocated_memory_read

Terse proof for `allocator_allocated_memory_read`.

`allocator_allocated_memory_read`:

$\{1\} \quad \forall (pm: (\text{plain_memory?}), ac: (\text{allocator?}(pm)), \text{addr}: \text{Address}, s: \text{State}):$ $pm' \text{states}(s) \wedge \text{union}(pm' \text{ro_addr}, pm' \text{rw_addr})(\text{addr}) \supset$ $ac' \text{allocated}(pm)(\text{state}(\text{memory_read}(pm' \text{mem})(\text{addr})(s))) = ac' \text{allocated}(pm)(s)$

Installing automatic rewrites from: `plain_memory_states_memory_read`
 Repeatedly Skolemizing and flattening,
 Rewriting using `allocator_const_private_memory_allocated`, matching in *,
 we get 2 subgoals:

allocator_allocated_memory_read.1:

{-1}	plain_memory?(pm')
{-2}	allocator?(pm')(ac')
{-3}	pm' 'states(s')
{-4}	union(pm' 'ro_addr, pm' 'rw_addr)(addr')
{1}	pm' 'states(state(memory_read(pm' 'mem)(addr')(s')))
{2}	ac' 'allocated(pm')(state(memory_read(pm' 'mem)(addr')(s'))) = ac' 'allocated(pm')(s')

Simplifying, rewriting, and recording with decision procedures,
This completes the proof of allocator_allocated_memory_read.1.

allocator_allocated_memory_read.2:

{-1}	plain_memory?(pm')
{-2}	allocator?(pm')(ac')
{-3}	pm' 'states(s')
{-4}	union(pm' 'ro_addr, pm' 'rw_addr)(addr')
{1}	stays_unchanged(pm') (state(memory_read(pm' 'mem)(addr')(s')), s', ac' 'private_mem(pm'))
{2}	ac' 'allocated(pm')(state(memory_read(pm' 'mem)(addr')(s'))) = ac' 'allocated(pm')(s')

Rewriting using stays_unchanged_symmetric, matching in *,
Using lemma plain_memory_stays_unchanged_memory_read,
Simplifying, rewriting, and recording with decision procedures,
Using lemma stays_unchanged_mono,
Simplifying, rewriting, and recording with decision procedures,
Rewriting using allocator_subset_private_memory_plain_memory, matching in *,
This completes the proof of allocator_allocated_memory_read.2.
Q.E.D.

C.10.35 Allocator.allocator_memory_pool_memory_read

Terse proof for allocator_memory_pool_memory_read.

allocator_memory_pool_memory_read:

{1}	\forall (pm: (plain_memory?), ac: (allocator?(pm)), addr: Address, s: State): pm 'states(s) \wedge union(pm 'ro_addr, pm 'rw_addr)(addr) \supset ac 'memory_pool(pm)(state(memory_read(pm 'mem)(addr)(s))) = ac 'memory_pool(pm)(s)
-----	--

Installing automatic rewrites from: plain_memory_states_memory_read
Repeatedly Skolemizing and flattening,
Rewriting using allocator_const_private_memory_memory_pool, matching in *,
we get 2 subgoals:

allocator_memory_pool_memory_read.1:

{-1}	plain_memory?(pm')
{-2}	allocator?(pm')(ac')
{-3}	pm' 'states(s')
{-4}	union(pm' 'ro_addr, pm' 'rw_addr)(addr')
{1}	pm' 'states(state(memory_read(pm' 'mem)(addr')(s')))
{2}	ac' 'memory_pool(pm')(state(memory_read(pm' 'mem)(addr')(s'))) = ac' 'memory_pool(pm')(s')

Simplifying, rewriting, and recording with decision procedures,
 This completes the proof of `allocator_memory_pool_memory_read.1`.
`allocator_memory_pool_memory_read.2`:

{-1}	<code>plain_memory?(pm')</code>
{-2}	<code>allocator?(pm')(ac')</code>
{-3}	<code>pm' states(s')</code>
{-4}	<code>union(pm' ro_addr, pm' rw_addr)(addr')</code>
{1}	<code>stays_unchanged(pm')</code> $(\text{state}(\text{memory_read}(\text{pm}' \text{ mem})(\text{addr}')(\text{s}')), \text{s}', \text{ac}' \text{ private_mem}(\text{pm}'))$
{2}	$\text{ac}' \text{ memory_pool}(\text{pm}')(\text{state}(\text{memory_read}(\text{pm}' \text{ mem})(\text{addr}')(\text{s}'))) =$ $\text{ac}' \text{ memory_pool}(\text{pm}')(\text{s}')$

Rewriting using `stays_unchanged_symmetric`, matching in *,
 Using lemma `plain_memory_stays_unchanged_memory_read`,
 Simplifying, rewriting, and recording with decision procedures,
 Using lemma `stays_unchanged_mono`,
 Simplifying, rewriting, and recording with decision procedures,
 Rewriting using `allocator_subset_private_memory_plain_memory`, matching in *,
 This completes the proof of `allocator_memory_pool_memory_read.2`.
 Q.E.D.

C.10.36 Allocat- tor.allocator_stays_unchanged_memory_write_private_mem_TCC1

Terse proof for `allocator_stays_unchanged_memory_write_private_mem_TCC1`.
`allocator_stays_unchanged_memory_write_private_mem_TCC1`:

{1}	$\forall (\text{pm}: (\text{plain_memory?}[\text{State}]), \text{ac}: (\text{allocator?}(\text{pm})), \text{addr}: \text{Address}, \text{byte}: \text{Byte}, \text{s}: \text{State}):$ $\text{pm}' \text{ states}(\text{s}) \wedge \text{pm}' \text{ rw_addr}(\text{addr}) \wedge \neg \text{ac}' \text{ private_mem}(\text{pm})(\text{addr}) \supset$ $\text{OK?}[\text{State}, \text{Unit}](\text{memory_write}(\text{pm}' \text{ mem})(\text{addr}, \text{byte})(\text{s})) \vee$ $\text{Exception?}[\text{State}, \text{Unit}](\text{memory_write}(\text{pm}' \text{ mem})(\text{addr}, \text{byte})(\text{s}))$
-----	--

Repeatedly Skolemizing and flattening,
 Rewriting using `plain_memory_memory_write_ok`, matching in *,
 This completes the proof of `allocator_stays_unchanged_memory_write_private_mem_TCC1`.
 Q.E.D.

C.10.37 Allocat- tor.allocator_stays_unchanged_memory_write_private_mem_TCC2

Terse proof for `allocator_stays_unchanged_memory_write_private_mem_TCC2`.
`allocator_stays_unchanged_memory_write_private_mem_TCC2`:

{1}	$\forall (\text{pm}: (\text{plain_memory?}[\text{State}]), \text{ac}: (\text{allocator?}(\text{pm})), \text{addr}: \text{Address}, \text{byte}: \text{Byte}, \text{s}: \text{State}):$ $\text{pm}' \text{ states}(\text{s}) \wedge \text{pm}' \text{ rw_addr}(\text{addr}) \wedge \neg \text{ac}' \text{ private_mem}(\text{pm})(\text{addr}) \supset$ $\text{pm}' \text{ states}(\text{state}[\text{State}, \text{Unit}](\text{memory_write}(\text{pm}' \text{ mem})(\text{addr}, \text{byte})(\text{s})))$
-----	--

Repeatedly Skolemizing and flattening,
 Rewriting using `plain_memory_states_memory_write`, matching in *,

This completes the proof of `allocator_stays_unchanged_memory_write_private_mem_TCC2`.
Q.E.D.

C.10.38

Allocator.allocator_stays_unchanged_memory_write_private_mem

Terse proof for `allocator_stays_unchanged_memory_write_private_mem`.

`allocator_stays_unchanged_memory_write_private_mem`:

$\{1\} \quad \forall (\text{pm}: (\text{plain_memory?}), \text{ac}: (\text{allocator?}(\text{pm})), \text{addr}: \text{Address}, \text{byte}: \text{Byte}, s: \text{State}):$ $\text{pm}'\text{states}(s) \wedge \text{pm}'\text{rw_addr}(\text{addr}) \wedge \neg \text{ac}'\text{private_mem}(\text{pm})(\text{addr}) \supset$ $\text{stays_unchanged}(\text{pm})$ $(s, \text{state}(\text{memory_write}(\text{pm}'\text{mem})(\text{addr}, \text{byte})(s)), \text{ac}'\text{private_mem}(\text{pm}))$

Installing automatic rewrites from: `plain_memory_states_memory_write`
 Repeatedly Skolemizing and flattening,
 Simplifying, rewriting, and recording with decision procedures,
 Using lemma `plain_memory_stays_unchanged_memory_write`,
 Simplifying, rewriting, and recording with decision procedures,
 Using lemma `stays_unchanged_mono`,
 Simplifying, rewriting, and recording with decision procedures,
 Hiding formulas: -1, 3,
 Using lemma `allocator_subset_private_memory_plain_memory`,
 Keeping (-1 1 2) and hiding *,
 Rewriting relative to the theory: sets,
 Simplifying, rewriting, and recording with decision procedures,
 Repeatedly Skolemizing and flattening,
 Repeatedly simplifying with decision procedures, rewriting, propositional reasoning, quantifier instantiation, skolemization, if-lifting and equality replacement,
 This completes the proof of `allocator_stays_unchanged_memory_write_private_mem`.
 Q.E.D.

C.10.39 Allocator.allocator_allocated_memory_write

Terse proof for `allocator_allocated_memory_write`.

`allocator_allocated_memory_write`:

$\{1\} \quad \forall (\text{pm}: (\text{plain_memory?}), \text{ac}: (\text{allocator?}(\text{pm})), \text{addr}: \text{Address}, \text{byte}: \text{Byte}, s: \text{State}):$ $\text{pm}'\text{states}(s) \wedge \text{pm}'\text{rw_addr}(\text{addr}) \wedge \neg \text{ac}'\text{private_mem}(\text{pm})(\text{addr}) \supset$ $\text{ac}'\text{allocated}(\text{pm})(\text{state}(\text{memory_write}(\text{pm}'\text{mem})(\text{addr}, \text{byte})(s))) =$ $\text{ac}'\text{allocated}(\text{pm})(s)$

Installing automatic rewrites from: `plain_memory_states_memory_write`
 Repeatedly Skolemizing and flattening,
 Simplifying, rewriting, and recording with decision procedures,
 Rewriting using `allocator_const_private_memory_allocated`, matching in *,
 Rewriting using `stays_unchanged_symmetric`, matching in *,
 Rewriting using `allocator_stays_unchanged_memory_write_private_mem`, matching in *,
 This completes the proof of `allocator_allocated_memory_write`.
 Q.E.D.

C.10.40 Allocator.allocator_memory_pool_memory_write

Terse proof for allocator_memory_pool_memory_write.

allocator_memory_pool_memory_write:

$$\{1\} \quad \forall (pm: (plain_memory?), ac: (allocator?(pm)), addr: Address, byte: Byte, s: State):$$

$$pm'states(s) \wedge pm'rw_addr(addr) \wedge \neg ac'private_mem(pm)(addr) \supset$$

$$ac'memory_pool(pm)(state(memory_write(pm'mem)(addr, byte)(s))) =$$

$$ac'memory_pool(pm)(s)$$

Installing automatic rewrites from: plain_memory_states_memory_write
 Repeatedly Skolemizing and flattening,
 Simplifying, rewriting, and recording with decision procedures,
 Rewriting using allocator_const_private_memory_memory_pool, matching in *,
 Rewriting using stays_unchanged_symmetric, matching in *,
 Rewriting using allocator_stays_unchanged_memory_write_private_mem, matching in *,
 This completes the proof of allocator_memory_pool_memory_write.
 Q.E.D.

C.10.41

Allocator.allocator_subset_address_block_alloc_memory_pool

Terse proof for allocator_subset_address_block_alloc_memory_pool.

allocator_subset_address_block_alloc_memory_pool:

$$\{1\} \quad \forall (pm: (plain_memory?), ac: (allocator?(pm)), size: posnat, addr: Address, s: State):$$

$$pm'states(s) \wedge$$

$$OK?(ac'alloc(pm)(size)(s)) \wedge \neg data(ac'alloc(pm)(size)(s)) = null_address$$

$$\supset$$

$$(address_block(data(ac'alloc(pm)(size)(s)), size) \subseteq ac'memory_pool(pm)(s))$$

Repeatedly Skolemizing and flattening,
 Using lemma allocator_subset_address_block_alloc_allocated,
 Simplifying, rewriting, and recording with decision procedures,
 Using lemma allocator_next_state_alloc,
 Installing automatic rewrites from: has_next_state
 Simplifying, rewriting, and recording with decision procedures,
 Using lemma allocator_subset_allocated_memory_pool,
 Simplifying, rewriting, and recording with decision procedures,
 Rewriting using allocator_memory_pool_alloc, matching in *,
 Forward chaining on subset_transitive,
 This completes the proof of allocator_subset_address_block_alloc_memory_pool.
 Q.E.D.

C.11 Proofs for ArithmeticExpressions (expressions.pvs)

C.11.1 ArithmeticExpressions.mod_impl_TCC1

Terse proof for mod_impl_TCC1.

mod_impl_TCC1:

$$\frac{}{\{1\} \exists (x: [d: [\text{int}, \{k: \text{int} \mid k \neq 0\}] \rightarrow \{r: \text{int} \mid \text{abs}(r) < \text{abs}(d'2)]): \text{TRUE}}$$

Instantiating the top quantifier in 1 with the terms: $(\lambda (a: \text{int}, b: \{k: \text{int} \mid k \neq 0\}): 0)$, we get 2 subgoals:

mod_impl_TCC1.1:

$$\frac{}{\{1\} \text{TRUE}}$$

which is trivially true.

This completes the proof of mod_impl_TCC1.1.

mod_impl_TCC1.2:

$$\frac{}{\{1\} \forall (b: \{k: \text{int} \mid k \neq 0\}): \text{abs}(0) < \text{abs}(b)}$$

Repeatedly Skolemizing and flattening,

Expanding the definition of abs,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of mod_impl_TCC1.2.

Q.E.D.

C.11.2 ArithmeticExpressions.div_impl_pos_range_TCC1

Terse proof for div_impl_pos_range_TCC1.

div_impl_pos_range_TCC1:

$$\frac{}{\{1\} \forall (j: \text{nat}): j \neq 0 \supset j > 0}$$

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of div_impl_pos_range_TCC1.

Q.E.D.

C.11.3 ArithmeticExpressions.plus_ptr_TCC1

Terse proof for plus_ptr_TCC1.

plus_ptr_TCC1:

$$\frac{}{\{1\} \forall (\text{typ}: \text{Cpp_Subtype}(\text{cv}(\text{pointer?})), \text{i_typ}: \text{Cpp_Subtype}(\text{cv}(\text{non_bool_integral_enum?})), \text{ptr_val}: (\text{range_pointer}(\text{cv_base}(\text{typ}))), \text{n}: (\text{range}(\text{i_typ}))) : \text{not_null?}(\text{typ})(\text{ptr_val}) \supset \text{array?}(\text{cv_base}(\text{typ})) \vee \text{pointer?}(\text{cv_base}(\text{typ})) \vee \text{reference?}(\text{cv_base}(\text{typ})) \vee \text{bitfield?}(\text{cv_base}(\text{typ})) \vee \text{enum?}(\text{cv_base}(\text{typ})) \vee \text{pointer_to_member?}(\text{cv_base}(\text{typ})) \vee \text{const?}(\text{cv_base}(\text{typ})) \vee \text{volatile?}(\text{cv_base}(\text{typ}))}$$

Repeatedly Skolemizing and flattening,

Keeping (-2 2) and hiding *,

C Proof scripts

Rewriting using `cv_base_result`, matching in `*`,

Keeping (1) and hiding `*`,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `plus_ptr_TCC1`.

Q.E.D.

C.11.4 ArithmeticExpressions.plus_ptr_TCC2

Terse proof for `plus_ptr_TCC2`.

`plus_ptr_TCC2`:

$\{1\} \quad \forall (\text{typ}: \text{Cpp_Subtype}(\text{cv}(\text{pointer?})), \text{i_typ}: \text{Cpp_Subtype}(\text{cv}(\text{non_bool_integral_enum?})), \\ \text{ptr_val}: (\text{range_pointer}(\text{cv_base}(\text{typ}))), \text{n}: (\text{range}(\text{i_typ}))) : \\ \text{not_null?}(\text{typ})(\text{ptr_val}) \supset \text{Cpp_Type?}(\text{typ}(\text{cv_base}(\text{typ})))$

Repeatedly Skolemizing and flattening,

Keeping (-1 -2 1) and hiding `*`,

Expanding the definition of `Cpp_Type?`,

Repeatedly Skolemizing and flattening,

Instantiating the top quantifier in -1 with the terms: (t!1),

Splitting conjunctions,

we get 2 subgoals:

plus_ptr_TCC2.1:

<pre> {-1} no_pointers_to_bitfield?(t') ∧ no_pointers_to_references?(t') ∧ no_cv_references?(t') ∧ no_reference_to_reference?(t') ∧ no_reference_to_bitfields?(t') ∧ no_pointer_to_member_to_reference?(t') ∧ no_pointer_to_member_to_cv_void?(t') ∧ no_cv_void_parameter?(t') ∧ no_array_of_references?(t') ∧ no_array_of_cv_void?(t') ∧ no_array_of_function?(t') ∧ no_array_of_abstract_class?(t') ∧ no_pointer_or_ref_to_incomplete_array_parameter?(t') ∧ no_array_return_type?(t') ∧ no_function_return_type?(t') ∧ bitfield_underlying_integral_or_enum_type?(t') ∧ cv_array?(t') ∧ enum_underlying_integral?(t') ∧ enum_constants?(t') ∧ const_volatile?(t') ∧ const_stutter?(t') ∧ volatile_stutter?(t') ∧ no_cv_class?(t') ∧ no_cv_union?(t') ∧ no_cv_function?(t') ∧ no_reference_to_void?(t') ∧ no_cv_void?(t') ∧ no_array_of_bitfields?(t') </pre>	<pre> {-2} cv(pointer?)(typ') {-3} subterm(t', typ(cv_base(typ'))) </pre>
<pre> {1} no_pointers_to_bitfield?(t') ∧ no_pointers_to_references?(t') ∧ no_cv_references?(t') ∧ no_reference_to_reference?(t') ∧ no_reference_to_bitfields?(t') ∧ no_pointer_to_member_to_reference?(t') ∧ no_pointer_to_member_to_cv_void?(t') ∧ no_cv_void_parameter?(t') ∧ no_array_of_references?(t') ∧ no_array_of_cv_void?(t') ∧ no_array_of_function?(t') ∧ no_array_of_abstract_class?(t') ∧ no_pointer_or_ref_to_incomplete_array_parameter?(t') ∧ no_array_return_type?(t') ∧ no_function_return_type?(t') ∧ bitfield_underlying_integral_or_enum_type?(t') ∧ cv_array?(t') ∧ enum_underlying_integral?(t') ∧ enum_constants?(t') ∧ const_volatile?(t') ∧ const_stutter?(t') ∧ volatile_stutter?(t') ∧ no_cv_class?(t') ∧ no_cv_union?(t') ∧ no_cv_function?(t') ∧ no_reference_to_void?(t') ∧ no_cv_void?(t') ∧ no_array_of_bitfields?(t') </pre>	

which is trivially true.

This completes the proof of `plus_ptr_TCC2.1`.

`plus_ptr_TCC2.2`:

{-1}	<code>cv(pointer?)(typ')</code>
{-2}	<code>subterm(t', typ(cv_base(typ')))</code>
{1}	<code>subterm(t', typ')</code>
{2}	$ \begin{aligned} &\text{no_pointers_to_bitfield?}(t') \wedge \\ &\text{no_pointers_to_references?}(t') \wedge \\ &\text{no_cv_references?}(t') \wedge \\ &\text{no_reference_to_reference?}(t') \wedge \\ &\text{no_reference_to_bitfields?}(t') \wedge \\ &\text{no_pointer_to_member_to_reference?}(t') \wedge \\ &\text{no_pointer_to_member_to_cv_void?}(t') \wedge \\ &\text{no_cv_void_parameter?}(t') \wedge \\ &\text{no_array_of_references?}(t') \wedge \\ &\text{no_array_of_cv_void?}(t') \wedge \\ &\text{no_array_of_function?}(t') \wedge \\ &\text{no_array_of_abstract_class?}(t') \wedge \\ &\text{no_pointer_or_ref_to_incomplete_array_parameter?}(t') \wedge \\ &\text{no_array_return_type?}(t') \wedge \\ &\text{no_function_return_type?}(t') \wedge \\ &\text{bitfield_underlying_integral_or_enum_type?}(t') \wedge \\ &\text{cv_array?}(t') \wedge \\ &\text{enum_underlying_integral?}(t') \wedge \\ &\text{enum_constants?}(t') \wedge \\ &\text{const_volatile?}(t') \wedge \\ &\text{const_stutter?}(t') \wedge \\ &\text{volatile_stutter?}(t') \wedge \\ &\text{no_cv_class?}(t') \wedge \\ &\text{no_cv_union?}(t') \wedge \\ &\text{no_cv_function?}(t') \wedge \\ &\text{no_reference_to_void?}(t') \wedge \\ &\text{no_cv_void?}(t') \wedge \text{no_array_of_bitfields?}(t') \end{aligned} $

Hiding formulas: 2,

Using lemma `subterm_cv_base`,

Using lemma `cv_base_result`,

we get 2 subgoals:

`plus_ptr_TCC2.2.1`:

{-1}	<code>pointer?(cv_base(typ'))</code>
{-2}	<code>subterm(cv_base(typ'), typ')</code>
{-3}	<code>cv(pointer?)(typ')</code>
{-4}	<code>subterm(t', typ(cv_base(typ')))</code>
{1}	<code>subterm(t', typ')</code>

Using lemma `subterm_transitive`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Keeping (-2 -4 1) and hiding *,

Expanding the definition of `subterm`,

which is trivially true.

This completes the proof of `plus_ptr_TCC2.2.1`.

plus_ptr_TCC2.2.2:

{-1}	subterm(cv_base(typ'), typ')
{-2}	cv(pointer?)(typ')
{-3}	subterm(t', typ(cv_base(typ')))
{1}	(pointer? \subseteq interpreted?)
{2}	subterm(t', typ')

Keeping (1) and hiding *,
 Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of plus_ptr_TCC2.2.2.
 Q.E.D.

C.11.5 ArithmeticExpressions.plus_ptr_TCC3

Terse proof for plus_ptr_TCC3.

plus_ptr_TCC3:

{1}	\forall (typ: Cpp_Subtype(cv(pointer?)), i_typ: Cpp_Subtype(cv(non_bool_integral_enum?)), ptr_val: (range_pointer(cv_base(typ))), n: (range(i_typ))): not_null?(typ)(ptr_val) \supset pointer?(cv_base(typ))
-----	--

Repeatedly Skolemizing and flattening,
 Rewriting using cv_base_result, matching in * where P gets pointer?,
 Keeping (1) and hiding *,
 Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of plus_ptr_TCC3.
 Q.E.D.

C.11.6 ArithmeticExpressions.plus_ptr_TCC4

Terse proof for plus_ptr_TCC4.

plus_ptr_TCC4:

{1}	\forall (typ: Cpp_Subtype(cv(pointer?)), i_typ: Cpp_Subtype(cv(non_bool_integral_enum?)), ptr_val: (range_pointer(cv_base(typ))), n: (range(i_typ))): \neg (not_null?(typ)(ptr_val) \wedge range_pointer(cv_base(typ)) (add(typ)(ptr_val, n \times size_of(typ(cv_base(typ)))))) \wedge check_bounds(typ) (add(typ)(ptr_val, n \times size_of(typ(cv_base(typ)))))) \supset pointer?(cv_base(typ))
-----	--

Repeatedly Skolemizing and flattening,
 Rewriting using cv_base_result, matching in *,
 Keeping (1) and hiding *,
 Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of plus_ptr_TCC4.
 Q.E.D.

C.11.7 ArithmeticExpressions.plus_ptr_TCC5

Terse proof for plus_ptr_TCC5.

plus_ptr_TCC5:

$$\frac{\{1\} \quad \forall (\text{typ}: \text{Cpp_Subtype}(\text{cv}(\text{pointer?})), \text{i_typ}: \text{Cpp_Subtype}(\text{cv}(\text{non_bool_integral_enum?})), \text{ptr_val}: (\text{range_pointer}(\text{cv_base}(\text{typ})))): \text{pointer?}(\text{cv_base}(\text{typ}))}{}$$

Repeatedly Skolemizing and flattening,
 Rewriting using cv_base_result, matching in *,
 Keeping (1) and hiding *,
 Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of plus_ptr_TCC5.
 Q.E.D.

C.11.8 ArithmeticExpressions.plus_ptr_TCC6

Terse proof for plus_ptr_TCC6.

plus_ptr_TCC6:

$$\frac{\{1\} \quad \forall (\text{typ}: \text{Cpp_Subtype}(\text{cv}(\text{pointer?})), \text{i_typ}: \text{Cpp_Subtype}(\text{cv}(\text{non_bool_integral_enum?}))) : \text{pointer?}(\text{cv_base}(\text{typ}))}{}$$

Repeatedly Skolemizing and flattening,
 Rewriting using cv_base_result, matching in *,
 Keeping (1) and hiding *,
 Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of plus_ptr_TCC6.
 Q.E.D.

C.11.9 ArithmeticExpressions.postinc_ptr_TCC1

Terse proof for postinc_ptr_TCC1.

postinc_ptr_TCC1:

$$\frac{\{1\} \quad \forall ((\text{typ}: \text{Cpp_Subtype}(v(\text{pointer?})))): \text{cv}(\text{pointer?})(\text{typ})}{}$$

Repeatedly Skolemizing and flattening,
 Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of postinc_ptr_TCC1.
 Q.E.D.

C.11.10 ArithmeticExpressions.postinc_ptr_TCC2

Terse proof for postinc_ptr_TCC2.

postinc_ptr_TCC2:

$$\frac{\{1\} \quad \forall ((\text{typ}: \text{Cpp_Subtype}(v(\text{pointer?})))): \text{pointer?}(\text{cv_base}(\text{typ}))}{}$$

Repeatedly Skolemizing and flattening,
 Using lemma postinc_ptr_TCC1,

Rewriting using `cv_base_result`, matching in `*`,
 Keeping (1) and hiding `*`,
 Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of `postinc_ptr_TCC2`.
 Q.E.D.

C.11.11 ArithmeticExpressions.postinc_ptr_TCC3

Terse proof for `postinc_ptr_TCC3`.

`postinc_ptr_TCC3`:

{1} $\forall ((\text{typ: Cpp_Subtype}(v(\text{pointer?})))): \text{range}(\text{uchar})(1)$

Repeatedly Skolemizing and flattening,
 Keeping (1) and hiding `*`,
 Expanding the definition of `range`,
 Expanding the definition of `range_integral`,
 Expanding the definition of `extend`,
 Using lemma `binary_range_uchar`,
 Using lemma `max_uchar`,
 Using lemma `pos_expt_gt`,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 This completes the proof of `postinc_ptr_TCC3`.
 Q.E.D.

C.11.12 ArithmeticExpressions.postinc_ptr_TCC4

Terse proof for `postinc_ptr_TCC4`.

`postinc_ptr_TCC4`:

{1} $\forall ((\text{typ: Cpp_Subtype}(v(\text{pointer?})))): \text{Cpp_Type?}(\text{uchar}) \wedge \text{cv}(\text{non_bool_integral_enum?})(\text{uchar})$

Repeatedly Skolemizing and flattening,
 Keeping (1) and hiding `*`,
 Applying propositional simplification,
 we get 2 subgoals:
`postinc_ptr_TCC4.1`:

{1} `Cpp_Type?(uchar)`

Expanding the definition of `Cpp_Type?`,
 Expanding the definition of `subterm`,
 Repeatedly Skolemizing and flattening,
 Replacing using formula -1,
 Installing automatic rewrites from: `no_pointers_to_bitfield?` `no_pointers_to_references?`
`no_cv_references?` `no_reference_to_reference?` `no_reference_to_bitfields?` `no_pointer_to_member_to_reference?`
`no_pointer_to_member_to_cv_void?` `no_cv_void_parameter?` `no_array_of_references?` `no_array_of_bitfields?`
`no_array_of_cv_void?` `no_array_of_function?` `no_array_of_abstract_class?` `no_pointer_or_ref_to_incomplete_array_parameter?`
`no_array_return_type?` `no_function_return_type?` `bitfield_underlying_integral_or_enum_type?` `cv_array?`
`enum_underlying_integral?` `enum_constants?` `const_volatile?` `const_stutter?` `volatile_stutter?`
`no_cv_class?` `no_cv_union?` `no_cv_function?` `no_reference_to_void?` `no_cv_void?`

C Proof scripts

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `postinc_ptr_TCC4.1`.
`postinc_ptr_TCC4.2`:

```
{1} cv(non_bool_integral_enum?)(uchar)
```

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `postinc_ptr_TCC4.2`.
Q.E.D.

C.11.13 ArithmeticExpressions.subscript_TCC1

Terse proof for `subscript_TCC1`.

`subscript_TCC1`:

```
{1}  ∨ ((typ: Cpp_Subtype(array?)), i_typ: Cpp_Subtype(cv(non_bool_integral_enum?))):  
      array?(typ) ∨  
      pointer?(typ) ∨  
      reference?(typ) ∨  
      bitfield?(typ) ∨  
      enum?(typ) ∨ pointer_to_member?(typ) ∨ const?(typ) ∨ volatile?(typ)
```

Repeatedly Skolemizing and flattening,
This completes the proof of `subscript_TCC1`.
Q.E.D.

C.11.14 ArithmeticExpressions.subscript_TCC2

Terse proof for `subscript_TCC2`.

`subscript_TCC2`:

```
{1}  ∨ ((typ: Cpp_Subtype(array?)), i_typ: Cpp_Subtype(cv(non_bool_integral_enum?))):  
      Cpp_Type?(pointer(typ(typ))) ∧ cv(pointer?)(pointer(typ(typ)))
```

Repeatedly Skolemizing and flattening,
Applying propositional simplification,
we get 2 subgoals:

`subscript_TCC2.1`:

```
{-1} Cpp_Type?(typ')  
{-2} array?(typ')  
{-3} Cpp_Type?(i_typ')  
{-4} cv(non_bool_integral_enum?)(i_typ')  
{1}  Cpp_Type?(pointer(typ(typ')))
```

Keeping (-1 -2 1) and hiding *,
Expanding the definition of `Cpp_Type?`,
Repeatedly Skolemizing and flattening,
Expanding the definition of subterm,
Splitting conjunctions,
we get 2 subgoals:

subscript_TCC2.1.1.1:

{-1}	$t' = \text{pointer}(\text{typ}(\text{typ}'))$
{-2}	$\forall (t: \text{Cpp_Type_}):$ $\text{subterm}(t, \text{typ}') \supset$ $\text{no_pointers_to_bitfield?}(t) \wedge$ $\text{no_pointers_to_references?}(t) \wedge$ $\text{no_cv_references?}(t) \wedge$ $\text{no_reference_to_reference?}(t) \wedge$ $\text{no_reference_to_bitfields?}(t) \wedge$ $\text{no_pointer_to_member_to_reference?}(t) \wedge$ $\text{no_pointer_to_member_to_cv_void?}(t) \wedge$ $\text{no_cv_void_parameter?}(t) \wedge$ $\text{no_array_of_references?}(t) \wedge$ $\text{no_array_of_cv_void?}(t) \wedge$ $\text{no_array_of_function?}(t) \wedge$ $\text{no_array_of_abstract_class?}(t) \wedge$ $\text{no_pointer_or_ref_to_incomplete_array_parameter?}(t) \wedge$ $\text{no_array_return_type?}(t) \wedge$ $\text{no_function_return_type?}(t) \wedge$ $\text{bitfield_underlying_integral_or_enum_type?}(t) \wedge$ $\text{cv_array?}(t) \wedge$ $\text{enum_underlying_integral?}(t) \wedge$ $\text{enum_constants?}(t) \wedge$ $\text{const_volatile?}(t) \wedge$ $\text{const_stutter?}(t) \wedge$ $\text{volatile_stutter?}(t) \wedge$ $\text{no_cv_class?}(t) \wedge$ $\text{no_cv_union?}(t) \wedge$ $\text{no_cv_function?}(t) \wedge$ $\text{no_reference_to_void?}(t) \wedge$ $\text{no_cv_void?}(t) \wedge \text{no_array_of_bitfields?}(t)$
{-3}	$\text{array?}(\text{typ}')$
{1}	$\text{no_pointers_to_bitfield?}(t') \wedge$ $\text{no_pointers_to_references?}(t') \wedge$ $\text{no_cv_references?}(t') \wedge$ $\text{no_reference_to_reference?}(t') \wedge$ $\text{no_reference_to_bitfields?}(t') \wedge$ $\text{no_pointer_to_member_to_reference?}(t') \wedge$ $\text{no_pointer_to_member_to_cv_void?}(t') \wedge$ $\text{no_cv_void_parameter?}(t') \wedge$ $\text{no_array_of_references?}(t') \wedge$ $\text{no_array_of_cv_void?}(t') \wedge$ $\text{no_array_of_function?}(t') \wedge$ $\text{no_array_of_abstract_class?}(t') \wedge$ $\text{no_pointer_or_ref_to_incomplete_array_parameter?}(t') \wedge$ $\text{no_array_return_type?}(t') \wedge$ $\text{no_function_return_type?}(t') \wedge$ $\text{bitfield_underlying_integral_or_enum_type?}(t') \wedge$ $\text{cv_array?}(t') \wedge$ $\text{enum_underlying_integral?}(t') \wedge$ $\text{enum_constants?}(t') \wedge$ $\text{const_volatile?}(t') \wedge$ $\text{const_stutter?}(t') \wedge$ $\text{volatile_stutter?}(t') \wedge$ $\text{no_cv_class?}(t') \wedge$ $\text{no_cv_union?}(t') \wedge$ $\text{no_cv_function?}(t') \wedge$ $\text{no_reference_to_void?}(t') \wedge$ $\text{no_cv_void?}(t') \wedge \text{no_array_of_bitfields?}(t')$

C Proof scripts

Replacing using formula -1,

Installing automatic rewrites from: `no_pointers_to_bitfield? no_pointers_to_references?
no_cv_references? no_reference_to_reference? no_reference_to_bitfields? no_pointer_to_member_to_reference?
no_pointer_to_member_to_cv_void? no_cv_void_parameter? no_array_of_references? no_array_of_cv_void?
no_array_of_function? no_array_of_abstract_class? no_array_of_bitfields? no_pointer_or_ref_to_incomplete_array_pa
no_array_return_type? no_function_return_type? bitfield_underlying_integral_or_enum_type? cv_array?
enum_underlying_integral? enum_constants? const_volatile? const_stutter? volatile_stutter?
no_cv_class? no_cv_union? no_cv_function? no_reference_to_void? no_cv_void?`

Instantiating the top quantifier in -2 with the terms: `(typ!1)`,

Expanding the definition of subterm,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `subscript_TCC2.1.1`.

subscript_TCC2.1.1.2:

{-1}	subterm(t' , typ(typ'))
{-2}	$\forall (t: \text{Cpp_Type_}):$ subterm(t , typ') \supset no_pointers_to_bitfield?(t) \wedge no_pointers_to_references?(t) \wedge no_cv_references?(t) \wedge no_reference_to_reference?(t) \wedge no_reference_to_bitfields?(t) \wedge no_pointer_to_member_to_reference?(t) \wedge no_pointer_to_member_to_cv_void?(t) \wedge no_cv_void_parameter?(t) \wedge no_array_of_references?(t) \wedge no_array_of_cv_void?(t) \wedge no_array_of_function?(t) \wedge no_array_of_abstract_class?(t) \wedge no_pointer_or_ref_to_incomplete_array_parameter?(t) \wedge no_array_return_type?(t) \wedge no_function_return_type?(t) \wedge bitfield_underlying_integral_or_enum_type?(t) \wedge cv_array?(t) \wedge enum_underlying_integral?(t) \wedge enum_constants?(t) \wedge const_volatile?(t) \wedge const_stutter?(t) \wedge volatile_stutter?(t) \wedge no_cv_class?(t) \wedge no_cv_union?(t) \wedge no_cv_function?(t) \wedge no_reference_to_void?(t) \wedge no_cv_void?(t) \wedge no_array_of_bitfields?(t)
{-3}	array?(typ')
{1}	no_pointers_to_bitfield?(t') \wedge no_pointers_to_references?(t') \wedge no_cv_references?(t') \wedge no_reference_to_reference?(t') \wedge no_reference_to_bitfields?(t') \wedge no_pointer_to_member_to_reference?(t') \wedge no_pointer_to_member_to_cv_void?(t') \wedge no_cv_void_parameter?(t') \wedge no_array_of_references?(t') \wedge no_array_of_cv_void?(t') \wedge no_array_of_function?(t') \wedge no_array_of_abstract_class?(t') \wedge no_pointer_or_ref_to_incomplete_array_parameter?(t') \wedge no_array_return_type?(t') \wedge no_function_return_type?(t') \wedge bitfield_underlying_integral_or_enum_type?(t') \wedge cv_array?(t') \wedge enum_underlying_integral?(t') \wedge enum_constants?(t') \wedge const_volatile?(t') \wedge const_stutter?(t') \wedge volatile_stutter?(t') \wedge no_cv_class?(t') \wedge no_cv_union?(t') \wedge no_cv_function?(t') \wedge no_reference_to_void?(t') \wedge no_cv_void?(t') \wedge no_array_of_bitfields?(t')

C Proof scripts

Instantiating quantified variables,
Expanding the definition of subterm,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `subscript_TCC2.1.2`.

`subscript_TCC2.2`:

{-1}	Cpp_Type?(typ')
{-2}	array?(typ')
{-3}	Cpp_Type?(i_typ')
{-4}	cv(non_bool_integral_enum?)(i_typ')
{1}	cv(pointer?)(pointer(typ(typ')))

Expanding the definition of cv,
Expanding the definition of c,
which is trivially true.
This completes the proof of `subscript_TCC2.2`.
Q.E.D.

C.11.15 ArithmeticExpressions.subscript_TCC3

Terse proof for `subscript_TCC3`.

`subscript_TCC3`:

{1}	$\forall ((\text{typ}: \text{Cpp_Subtype}(\text{array?})), \text{i_typ}: \text{Cpp_Subtype}(\text{cv}(\text{non_bool_integral_enum?})))$ $\text{Cpp_Type?}(\text{pointer}(\text{typ}(\text{typ})))$
-----	---

Repeatedly Skolemizing and flattening,
Using lemma `subscript_TCC2`,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `subscript_TCC3`.
Q.E.D.

C.11.16 ArithmeticExpressions.subscript_TCC4

Terse proof for `subscript_TCC4`.

`subscript_TCC4`:

{1}	$\forall ((\text{typ}: \text{Cpp_Subtype}(\text{array?})), \text{i_typ}: \text{Cpp_Subtype}(\text{cv}(\text{non_bool_integral_enum?})))$ $\text{pointer?}(\text{cv_base}(\text{pointer}(\text{typ}(\text{typ}))))$
-----	--

Repeatedly Skolemizing and flattening,
Rewriting using `cv_base_result2`, matching in *,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `subscript_TCC4`.
Q.E.D.

C.11.17 ArithmeticExpressions.ptr_to_nonzero_object_TCC1

Terse proof for `ptr_to_nonzero_object_TCC1`.

ptr_to_nonzero_object_TCC1:

{1}	\forall (typ: Cpp_Subtype(extend[Cpp_Type_, (pointer?), bool, FALSE] (λ (p: (pointer?): \neg void?(typ(p)) \wedge \neg function?(typ(p))))): array?(cv_base(typ)) \vee pointer?(cv_base(typ)) \vee reference?(cv_base(typ)) \vee bitfield?(cv_base(typ)) \vee enum?(cv_base(typ)) \vee pointer_to_member?(cv_base(typ)) \vee const?(cv_base(typ)) \vee volatile?(cv_base(typ))
-----	---

Repeatedly Skolemizing and flattening,
 Expanding the definition of extend,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 Keeping (-2 4) and hiding *,
 Rewriting using cv_base_result, matching in *,
 we get 2 subgoals:

ptr_to_nonzero_object_TCC1.1:

{-1}	pointer?(typ')
{1}	cv(pointer?)(typ')
{2}	pointer?(cv_base(typ'))

Hiding formulas: 2,
 Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of ptr_to_nonzero_object_TCC1.1.

ptr_to_nonzero_object_TCC1.2:

{-1}	pointer?(typ')
{1}	(pointer? \subseteq interpreted?)
{2}	pointer?(cv_base(typ'))

Keeping (1) and hiding *,
 Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of ptr_to_nonzero_object_TCC1.2.
 Q.E.D.

C.11.18 ArithmeticExpressions.ptr_to_nonzero_object_TCC2

Terse proof for ptr_to_nonzero_object_TCC2.

ptr_to_nonzero_object_TCC2:

{1}	\forall (typ: Cpp_Subtype(extend[Cpp_Type_, (pointer?), bool, FALSE] (λ (p: (pointer?): \neg void?(typ(p)) \wedge \neg function?(typ(p))))): array?(typ(cv_base(typ))) \supset Cpp_Type?(typ(cv_base(typ)))
-----	--

Repeatedly Skolemizing and flattening,
 Expanding the definition of extend,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Expanding the definition of cv_base,
 Keeping (-1 -2 3) and hiding *,
 Expanding the definition of Cpp_Type?,
 Repeatedly Skolemizing and flattening,
 Instantiating quantified variables,
 Expanding the definition of subterm,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 This completes the proof of ptr_to_nonzero_object_TCC2.
 Q.E.D.

C.11.19 ArithmeticExpressions.ptr_to_nonzero_object_TCC3

Terse proof for ptr_to_nonzero_object_TCC3.

ptr_to_nonzero_object_TCC3:

$\{1\} \quad \forall (\text{typ}: \\ \text{Cpp_Subtype}(\text{extend}[\text{Cpp_Type_}, (\text{pointer?}), \text{bool}, \text{FALSE}] \\ (\lambda (p: (\text{pointer?})): \\ \neg \text{void?}(\text{typ}(p)) \wedge \neg \text{function?}(\text{typ}(p))))): \\ (\text{array?}(\text{typ}(\text{cv_base}(\text{typ}))) \supset \text{size_of}(\text{typ}(\text{cv_base}(\text{typ}))) > 0) \supset \\ \text{Cpp_Type?}(\text{typ}(\text{cv_base}(\text{typ}))))$

Repeatedly Skolemizing and flattening,
 Hiding formulas: -3,
 Expanding the definition of extend,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 Expanding the definition of cv_base,
 Keeping (-1 -2 3) and hiding *,
 Expanding the definition of Cpp_Type?,
 Repeatedly Skolemizing and flattening,
 Instantiating quantified variables,
 Expanding the definition of subterm,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 This completes the proof of ptr_to_nonzero_object_TCC3.
 Q.E.D.

C.11.20 ArithmeticExpressions.ptr_to_nonzero_object

Terse proof for ptr_to_nonzero_object.

ptr_to_nonzero_object:

$\{1\} \quad \forall (\text{typ}: \\ \text{Cpp_Subtype}(\text{extend}[\text{Cpp_Type_}, (\text{pointer?}), \text{bool}, \text{FALSE}] \\ (\lambda (p: (\text{pointer?})): \\ \neg \text{void?}(\text{typ}(p)) \wedge \neg \text{function?}(\text{typ}(p))))): \\ (\text{array?}(\text{typ}(\text{cv_base}(\text{typ}))) \supset \text{size_of}(\text{typ}(\text{cv_base}(\text{typ}))) > 0) \supset \\ \text{size_of}(\text{typ}(\text{cv_base}(\text{typ}))) > 0$
--

Repeatedly Skolemizing and flattening,
 Expanding the definition of extend,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 Expanding the definition of cv_base,

Expanding the definition of size_of,
 Expanding the definition of uidt,
 Expanding the definition of Cpp_Type?,
 Instantiating the top quantifier in -1 with the terms: (typ!1),
 Expanding the definition of subterm,
 Applying disjunctive simplification to flatten sequent,
 Hiding formulas: (-4 -5 -6 -7 -8 -9 -10 -11 -12 -13 -14 -15 -16),
 Hiding formulas: (-13 -15),
 Expanding the definition of no_pointers_to_bitfield?,
 Expanding the definition of no_pointers_to_references?,
 Installing automatic rewrites from: pod_size
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 we get 13 subgoals:

`ptr_to_nonzero_object.1:`

{-1}	no_cv_references?(typ')
{-2}	cv_array?(typ')
{-3}	enum_underlying_integral?(typ')
{-4}	enum_constants?(typ')
{-5}	const_volatile?(typ')
{-6}	const_stutter?(typ')
{-7}	volatile_stutter?(typ')
{-8}	no_cv_class?(typ')
{-9}	no_cv_union?(typ')
{-10}	no_cv_function?(typ')
{-11}	no_cv_void?(typ')
{-12}	pointer?(typ')
{-13}	pointer?(typ(typ'))
<hr/>	
{1}	cv(bitfield?)(typ(typ'))
{2}	size(uidt(dt_cv_pointer(typ(typ')))) > 0

Expanding the definition of dt_cv_pointer,
 Adding type constraints for dt_pointer(typ(typ!1)),
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 This completes the proof of `ptr_to_nonzero_object.1`.

`ptr_to_nonzero_object.2:`

{-1}	no_cv_references?(typ')
{-2}	cv_array?(typ')
{-3}	enum_underlying_integral?(typ')
{-4}	enum_constants?(typ')
{-5}	const_volatile?(typ')
{-6}	const_stutter?(typ')
{-7}	volatile_stutter?(typ')
{-8}	no_cv_class?(typ')
{-9}	no_cv_union?(typ')
{-10}	no_cv_function?(typ')
{-11}	no_cv_void?(typ')
{-12}	pointer?(typ')
{-13}	pointer_to_member?(typ(typ'))
<hr/>	
{1}	cv(bitfield?)(typ(typ'))
{2}	size(uidt(dt_cv_ptm(typ(typ')))) > 0

Expanding the definition of dt_cv_ptm,

C Proof scripts

Adding type constraints for `dt_ptm(typ(typ1))`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `ptr_to_nonzero_object.2`.

`ptr_to_nonzero_object.3`:

{-1}	<code>no_cv_references?(typ')</code>
{-2}	<code>cv_array?(typ')</code>
{-3}	<code>enum_underlying_integral?(typ')</code>
{-4}	<code>enum_constants?(typ')</code>
{-5}	<code>const_volatile?(typ')</code>
{-6}	<code>const_stutter?(typ')</code>
{-7}	<code>volatile_stutter?(typ')</code>
{-8}	<code>no_cv_class?(typ')</code>
{-9}	<code>no_cv_union?(typ')</code>
{-10}	<code>no_cv_function?(typ')</code>
{-11}	<code>no_cv_void?(typ')</code>
{-12}	<code>pointer?(typ')</code>
{-13}	<code>bitfield?(typ(typ'))</code>
<hr/>	
{1}	<code>cv(bitfield?)(typ(typ'))</code>
{2}	<code>size(uidt(dt_cv_bitfield(typ(typ')))) > 0</code>

Expanding the definition of `cv`,

Expanding the definition of `c`,

which is trivially true.

This completes the proof of `ptr_to_nonzero_object.3`.

`ptr_to_nonzero_object.4`:

{-1}	<code>no_cv_references?(typ')</code>
{-2}	<code>cv_array?(typ')</code>
{-3}	<code>enum_underlying_integral?(typ')</code>
{-4}	<code>enum_constants?(typ')</code>
{-5}	<code>const_volatile?(typ')</code>
{-6}	<code>const_stutter?(typ')</code>
{-7}	<code>volatile_stutter?(typ')</code>
{-8}	<code>no_cv_class?(typ')</code>
{-9}	<code>no_cv_union?(typ')</code>
{-10}	<code>no_cv_function?(typ')</code>
{-11}	<code>no_cv_void?(typ')</code>
{-12}	<code>pointer?(typ')</code>
{-13}	<code>const?(typ(typ'))</code>
{-14}	<code>volatile?(typ(typ(typ')))</code>
<hr/>	
{1}	<code>cv(bitfield?)(typ(typ'))</code>
{2}	<code>size(uidt(typ(typ(typ')))) > 0</code>

Expanding the definition of `uidt`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 4 subgoals:

ptr_to_nonzero_object.4.1:

{-1}	no_cv_references?(typ')
{-2}	cv_array?(typ')
{-3}	enum_underlying_integral?(typ')
{-4}	enum_constants?(typ')
{-5}	const_volatile?(typ')
{-6}	const_stutter?(typ')
{-7}	volatile_stutter?(typ')
{-8}	no_cv_class?(typ')
{-9}	no_cv_union?(typ')
{-10}	no_cv_function?(typ')
{-11}	no_cv_void?(typ')
{-12}	pointer?(typ')
{-13}	const?(typ(typ'))
{-14}	volatile?(typ(typ(typ')))
{-15}	pointer?(typ(typ(typ(typ'))))
{1}	cv(bitfield?)(typ(typ'))
{2}	size(uidt(dt_cv_pointer(typ(typ(typ'))))) > 0

Expanding the definition of dt_cv_pointer,

Rewriting using dt_volatile_size, matching in *,

we get 2 subgoals:

ptr_to_nonzero_object.4.1.1:

{-1}	no_cv_references?(typ')
{-2}	cv_array?(typ')
{-3}	enum_underlying_integral?(typ')
{-4}	enum_constants?(typ')
{-5}	const_volatile?(typ')
{-6}	const_stutter?(typ')
{-7}	volatile_stutter?(typ')
{-8}	no_cv_class?(typ')
{-9}	no_cv_union?(typ')
{-10}	no_cv_function?(typ')
{-11}	no_cv_void?(typ')
{-12}	pointer?(typ')
{-13}	const?(typ(typ'))
{-14}	volatile?(typ(typ(typ')))
{-15}	pointer?(typ(typ(typ(typ'))))
{1}	cv(bitfield?)(typ(typ'))
{2}	size(uidt(dt_cv_pointer(typ(typ(typ'))))) > 0

Expanding the definition of dt_cv_pointer,

Adding type constraints for dt_pointer(typ(typ(typ!1))),

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of ptr_to_nonzero_object.4.1.1.

ptr_to_nonzero_object.4.1.2:

{-1}	no_cv_references?(typ')
{-2}	cv_array?(typ')
{-3}	enum_underlying_integral?(typ')
{-4}	enum_constants?(typ')
{-5}	const_volatile?(typ')
{-6}	const_stutter?(typ')
{-7}	volatile_stutter?(typ')
{-8}	no_cv_class?(typ')
{-9}	no_cv_union?(typ')
{-10}	no_cv_function?(typ')
{-11}	no_cv_void?(typ')
{-12}	pointer?(typ')
{-13}	const?(typ(typ'))
{-14}	volatile?(typ(typ(typ')))
{-15}	pointer?(typ(typ(typ(typ'))))
{1}	$\exists (x_1 :$ <div style="padding-left: 20px;">(interpreted_data_type? <div style="padding-left: 20px;">[[(range_pointer(cv_base(typ(typ(typ(typ'))))))]])) :</div> </div>
{2}	$\text{cv}(\text{bitfield?})(\text{typ}(\text{typ}'))$
{3}	$\text{size}(\text{uidt}(\text{dt_volatile}(\text{dt_cv_pointer}(\text{typ}(\text{typ}(\text{typ}(\text{typ}')))))) > 0$

Instantiating the top quantifier in 1 with the terms: (dt_pointer(cv_base(typ(typ(typ(typ!1)))))),

we get 2 subgoals:

ptr_to_nonzero_object.4.1.2.1:

{-1}	no_cv_references?(typ')
{-2}	cv_array?(typ')
{-3}	enum_underlying_integral?(typ')
{-4}	enum_constants?(typ')
{-5}	const_volatile?(typ')
{-6}	const_stutter?(typ')
{-7}	volatile_stutter?(typ')
{-8}	no_cv_class?(typ')
{-9}	no_cv_union?(typ')
{-10}	no_cv_function?(typ')
{-11}	no_cv_void?(typ')
{-12}	pointer?(typ')
{-13}	const?(typ(typ'))
{-14}	volatile?(typ(typ(typ')))
{-15}	pointer?(typ(typ(typ(typ'))))
{1}	TRUE
{2}	$\text{cv}(\text{bitfield?})(\text{typ}(\text{typ}'))$
{3}	$\text{size}(\text{uidt}(\text{dt_volatile}(\text{dt_cv_pointer}(\text{typ}(\text{typ}(\text{typ}(\text{typ}')))))) > 0$

which is trivially true.

This completes the proof of ptr_to_nonzero_object.4.1.2.1.

ptr_to_nonzero_object.4.1.2.2:

{-1}	no_cv_references?(typ')
{-2}	cv_array?(typ')
{-3}	enum_underlying_integral?(typ')
{-4}	enum_constants?(typ')
{-5}	const_volatile?(typ')
{-6}	const_stutter?(typ')
{-7}	volatile_stutter?(typ')
{-8}	no_cv_class?(typ')
{-9}	no_cv_union?(typ')
{-10}	no_cv_function?(typ')
{-11}	no_cv_void?(typ')
{-12}	pointer?(typ')
{-13}	const?(typ(typ'))
{-14}	volatile?(typ(typ(typ')))
{-15}	pointer?(typ(typ(typ(typ'))))
{1}	pointer?(cv_base(typ(typ(typ(typ')))))
{2}	cv(bitfield?)(typ(typ'))
{3}	size(uidt(dt_volatile(dt_cv_pointer(typ(typ(typ(typ')))))))) > 0

Keeping (-15 1) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of ptr_to_nonzero_object.4.1.2.2.

ptr_to_nonzero_object.4.2:

{-1}	no_cv_references?(typ')
{-2}	cv_array?(typ')
{-3}	enum_underlying_integral?(typ')
{-4}	enum_constants?(typ')
{-5}	const_volatile?(typ')
{-6}	const_stutter?(typ')
{-7}	volatile_stutter?(typ')
{-8}	no_cv_class?(typ')
{-9}	no_cv_union?(typ')
{-10}	no_cv_function?(typ')
{-11}	no_cv_void?(typ')
{-12}	pointer?(typ')
{-13}	const?(typ(typ'))
{-14}	volatile?(typ(typ(typ')))
{-15}	pointer_to_member?(typ(typ(typ(typ'))))
{1}	cv(bitfield?)(typ(typ'))
{2}	size(uidt(dt_cv_ptm(typ(typ(typ'))))) > 0

Expanding the definition of dt_cv_ptm,

Rewriting using dt_volatile_size, matching in *,

we get 2 subgoals:

`ptr_to_nonzero_object.4.2.1:`

{-1}	<code>no_cv_references?(typ')</code>
{-2}	<code>cv_array?(typ')</code>
{-3}	<code>enum_underlying_integral?(typ')</code>
{-4}	<code>enum_constants?(typ')</code>
{-5}	<code>const_volatile?(typ')</code>
{-6}	<code>const_stutter?(typ')</code>
{-7}	<code>volatile_stutter?(typ')</code>
{-8}	<code>no_cv_class?(typ')</code>
{-9}	<code>no_cv_union?(typ')</code>
{-10}	<code>no_cv_function?(typ')</code>
{-11}	<code>no_cv_void?(typ')</code>
{-12}	<code>pointer?(typ')</code>
{-13}	<code>const?(typ(typ'))</code>
{-14}	<code>volatile?(typ(typ(typ')))</code>
{-15}	<code>pointer_to_member?(typ(typ(typ(typ'))))</code>
{1}	<code>cv(bitfield?)(typ(typ'))</code>
{2}	<code>size(uidt(dt_cv_ptm(typ(typ(typ(typ')))))) > 0</code>

Expanding the definition of `dt_cv_ptm`,

Adding type constraints for `dt_ptm(typ(typ(typ(typ!1))))`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `ptr_to_nonzero_object.4.2.1`.

`ptr_to_nonzero_object.4.2.2:`

{-1}	<code>no_cv_references?(typ')</code>
{-2}	<code>cv_array?(typ')</code>
{-3}	<code>enum_underlying_integral?(typ')</code>
{-4}	<code>enum_constants?(typ')</code>
{-5}	<code>const_volatile?(typ')</code>
{-6}	<code>const_stutter?(typ')</code>
{-7}	<code>volatile_stutter?(typ')</code>
{-8}	<code>no_cv_class?(typ')</code>
{-9}	<code>no_cv_union?(typ')</code>
{-10}	<code>no_cv_function?(typ')</code>
{-11}	<code>no_cv_void?(typ')</code>
{-12}	<code>pointer?(typ')</code>
{-13}	<code>const?(typ(typ'))</code>
{-14}	<code>volatile?(typ(typ(typ')))</code>
{-15}	<code>pointer_to_member?(typ(typ(typ(typ'))))</code>
{1}	$\exists (x_1:$ $(\text{interpreted_data_type?}[(\text{range_ptm}(\text{cv_base}(\text{typ}(\text{typ}(\text{typ}(\text{typ}))))))]))):$
{2}	<code>cv(bitfield?)(typ(typ'))</code>
{3}	<code>size(uidt(dt_volatile(dt_cv_ptm(typ(typ(typ(typ')))))) > 0</code>

Instantiating the top quantifier in 1 with the terms: `(dt_ptm(cv_base(typ(typ(typ(typ!1))))))`,

we get 2 subgoals:

ptr_to_nonzero_object.4.2.2.1:

{-1}	no_cv_references?(typ')
{-2}	cv_array?(typ')
{-3}	enum_underlying_integral?(typ')
{-4}	enum_constants?(typ')
{-5}	const_volatile?(typ')
{-6}	const_stutter?(typ')
{-7}	volatile_stutter?(typ')
{-8}	no_cv_class?(typ')
{-9}	no_cv_union?(typ')
{-10}	no_cv_function?(typ')
{-11}	no_cv_void?(typ')
{-12}	pointer?(typ')
{-13}	const?(typ(typ'))
{-14}	volatile?(typ(typ(typ')))
{-15}	pointer_to_member?(typ(typ(typ(typ'))))
{1}	TRUE
{2}	cv(bitfield?)(typ(typ'))
{3}	size(uidt(dt_volatile(dt_cv_ptm(typ(typ(typ(typ')))))))) > 0

which is trivially true.

This completes the proof of ptr_to_nonzero_object.4.2.2.1.

ptr_to_nonzero_object.4.2.2.2:

{-1}	no_cv_references?(typ')
{-2}	cv_array?(typ')
{-3}	enum_underlying_integral?(typ')
{-4}	enum_constants?(typ')
{-5}	const_volatile?(typ')
{-6}	const_stutter?(typ')
{-7}	volatile_stutter?(typ')
{-8}	no_cv_class?(typ')
{-9}	no_cv_union?(typ')
{-10}	no_cv_function?(typ')
{-11}	no_cv_void?(typ')
{-12}	pointer?(typ')
{-13}	const?(typ(typ'))
{-14}	volatile?(typ(typ(typ')))
{-15}	pointer_to_member?(typ(typ(typ(typ'))))
{1}	pointer_to_member?(cv_base(typ(typ(typ(typ')))))
{2}	cv(bitfield?)(typ(typ'))
{3}	size(uidt(dt_volatile(dt_cv_ptm(typ(typ(typ(typ')))))))) > 0

Keeping (-15 1) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of ptr_to_nonzero_object.4.2.2.2.

`ptr_to_nonzero_object.4.3:`

{-1}	<code>no_cv_references?(typ')</code>
{-2}	<code>cv_array?(typ')</code>
{-3}	<code>enum_underlying_integral?(typ')</code>
{-4}	<code>enum_constants?(typ')</code>
{-5}	<code>const_volatile?(typ')</code>
{-6}	<code>const_stutter?(typ')</code>
{-7}	<code>volatile_stutter?(typ')</code>
{-8}	<code>no_cv_class?(typ')</code>
{-9}	<code>no_cv_union?(typ')</code>
{-10}	<code>no_cv_function?(typ')</code>
{-11}	<code>no_cv_void?(typ')</code>
{-12}	<code>pointer?(typ')</code>
{-13}	<code>const?(typ(typ'))</code>
{-14}	<code>volatile?(typ(typ(typ')))</code>
{-15}	<code>bitfield?(typ(typ(typ(typ'))))</code>
{1}	<code>cv(bitfield?)(typ(typ'))</code>
{2}	<code>size(uidt(dt_cv_bitfield(typ(typ(typ'))))) > 0</code>

Expanding the definition of `cv`,

which is trivially true.

This completes the proof of `ptr_to_nonzero_object.4.3`.

`ptr_to_nonzero_object.4.4:`

{-1}	<code>no_cv_references?(typ')</code>
{-2}	<code>cv_array?(typ')</code>
{-3}	<code>enum_underlying_integral?(typ')</code>
{-4}	<code>enum_constants?(typ')</code>
{-5}	<code>const_volatile?(typ')</code>
{-6}	<code>const_stutter?(typ')</code>
{-7}	<code>volatile_stutter?(typ')</code>
{-8}	<code>no_cv_class?(typ')</code>
{-9}	<code>no_cv_union?(typ')</code>
{-10}	<code>no_cv_function?(typ')</code>
{-11}	<code>no_cv_void?(typ')</code>
{-12}	<code>pointer?(typ')</code>
{-13}	<code>const?(typ(typ'))</code>
{-14}	<code>volatile?(typ(typ(typ')))</code>
{-15}	<code>floating_point?(typ(typ(typ(typ'))))</code>
{1}	<code>cv(bitfield?)(typ(typ'))</code>
{2}	<code>bool?(typ(typ(typ(typ'))))</code>
{3}	<code>pointer?(typ(typ(typ(typ'))))</code>
{4}	<code>pointer_to_member?(typ(typ(typ(typ'))))</code>
{5}	<code>bitfield?(typ(typ(typ(typ'))))</code>
{6}	<code>size(uidt(dt_cv_float(typ(typ(typ'))))) > 0</code>

Expanding the definition of `dt_cv_float`,

Rewriting using `dt_volatile_size`, matching in `*`,

we get 2 subgoals:

ptr_to_nonzero_object.4.4.1:

{-1}	no_cv_references?(typ')
{-2}	cv_array?(typ')
{-3}	enum_underlying_integral?(typ')
{-4}	enum_constants?(typ')
{-5}	const_volatile?(typ')
{-6}	const_stutter?(typ')
{-7}	volatile_stutter?(typ')
{-8}	no_cv_class?(typ')
{-9}	no_cv_union?(typ')
{-10}	no_cv_function?(typ')
{-11}	no_cv_void?(typ')
{-12}	pointer?(typ')
{-13}	const?(typ(typ'))
{-14}	volatile?(typ(typ(typ')))
{-15}	floating_point?(typ(typ(typ(typ'))))
{1}	cv(bitfield?)(typ(typ'))
{2}	bool?(typ(typ(typ(typ'))))
{3}	pointer?(typ(typ(typ(typ'))))
{4}	pointer_to_member?(typ(typ(typ(typ'))))
{5}	bitfield?(typ(typ(typ(typ'))))
{6}	size(uidt(dt_cv_float(typ(typ(typ(typ')))))) > 0

Expanding the definition of dt_cv_float,

Adding type constraints for dt_floating_point(typ(typ(typ(typ!1)))),

Expanding the definition of floating_point?,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of ptr_to_nonzero_object.4.4.1.

ptr_to_nonzero_object.4.4.2:

{-1}	no_cv_references?(typ')
{-2}	cv_array?(typ')
{-3}	enum_underlying_integral?(typ')
{-4}	enum_constants?(typ')
{-5}	const_volatile?(typ')
{-6}	const_stutter?(typ')
{-7}	volatile_stutter?(typ')
{-8}	no_cv_class?(typ')
{-9}	no_cv_union?(typ')
{-10}	no_cv_function?(typ')
{-11}	no_cv_void?(typ')
{-12}	pointer?(typ')
{-13}	const?(typ(typ'))
{-14}	volatile?(typ(typ(typ')))
{-15}	floating_point?(typ(typ(typ(typ'))))
{1}	$\exists (x_1:$ <div style="padding-left: 40px;">(interpreted_data_type? <div style="padding-left: 40px;">[[((range_floating_point(cv_base(typ(typ(typ(typ'))))))]]))):</div> <div style="padding-left: 40px;">TRUE</div> </div>
{2}	cv(bitfield?)(typ(typ'))
{3}	bool?(typ(typ(typ(typ'))))
{4}	pointer?(typ(typ(typ(typ'))))
{5}	pointer_to_member?(typ(typ(typ(typ'))))
{6}	bitfield?(typ(typ(typ(typ'))))
{7}	size(uidt(dt_volatile(dt_cv_float(typ(typ(typ(typ')))))))) > 0

Instantiating the top quantifier in 1 with the terms: (dt_floating_point(cv_base(typ(typ(typ(typ!1)))))), we get 2 subgoals:

ptr_to_nonzero_object.4.4.2.1:

{-1}	no_cv_references?(typ')
{-2}	cv_array?(typ')
{-3}	enum_underlying_integral?(typ')
{-4}	enum_constants?(typ')
{-5}	const_volatile?(typ')
{-6}	const_stutter?(typ')
{-7}	volatile_stutter?(typ')
{-8}	no_cv_class?(typ')
{-9}	no_cv_union?(typ')
{-10}	no_cv_function?(typ')
{-11}	no_cv_void?(typ')
{-12}	pointer?(typ')
{-13}	const?(typ(typ'))
{-14}	volatile?(typ(typ(typ')))
{-15}	floating_point?(typ(typ(typ(typ'))))
{1}	TRUE
{2}	cv(bitfield?)(typ(typ'))
{3}	bool?(typ(typ(typ(typ'))))
{4}	pointer?(typ(typ(typ(typ'))))
{5}	pointer_to_member?(typ(typ(typ(typ'))))
{6}	bitfield?(typ(typ(typ(typ'))))
{7}	size(uidt(dt_volatile(dt_cv_float(typ(typ(typ(typ')))))))) > 0

which is trivially true.

This completes the proof of `ptr_to_nonzero_object.4.4.2.1`.

`ptr_to_nonzero_object.4.4.2.2`:

{-1}	no_cv_references?(typ')
{-2}	cv_array?(typ')
{-3}	enum_underlying_integral?(typ')
{-4}	enum_constants?(typ')
{-5}	const_volatile?(typ')
{-6}	const_stutter?(typ')
{-7}	volatile_stutter?(typ')
{-8}	no_cv_class?(typ')
{-9}	no_cv_union?(typ')
{-10}	no_cv_function?(typ')
{-11}	no_cv_void?(typ')
{-12}	pointer?(typ')
{-13}	const?(typ(typ'))
{-14}	volatile?(typ(typ(typ')))
{-15}	floating_point?(typ(typ(typ(typ'))))
{1}	floating_point?(cv_base(typ(typ(typ(typ')))))
{2}	cv(bitfield?)(typ(typ'))
{3}	bool?(typ(typ(typ(typ'))))
{4}	pointer?(typ(typ(typ(typ'))))
{5}	pointer_to_member?(typ(typ(typ(typ'))))
{6}	bitfield?(typ(typ(typ(typ'))))
{7}	size(uidt(dt_volatile(dt_cv_float(typ(typ(typ(typ')))))))) > 0

Keeping (-15 1) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `ptr_to_nonzero_object.4.4.2.2`.

`ptr_to_nonzero_object.5`:

{-1}	no_cv_references?(typ')
{-2}	cv_array?(typ')
{-3}	enum_underlying_integral?(typ')
{-4}	enum_constants?(typ')
{-5}	const_volatile?(typ')
{-6}	const_stutter?(typ')
{-7}	volatile_stutter?(typ')
{-8}	no_cv_class?(typ')
{-9}	no_cv_union?(typ')
{-10}	no_cv_function?(typ')
{-11}	no_cv_void?(typ')
{-12}	pointer?(typ')
{-13}	const?(typ(typ'))
{-14}	floating_point?(typ(typ(typ')))
{1}	cv(bitfield?)(typ(typ'))
{2}	bool?(typ(typ(typ')))
{3}	pointer?(typ(typ(typ')))
{4}	pointer_to_member?(typ(typ(typ')))
{5}	bitfield?(typ(typ(typ')))
{6}	volatile?(typ(typ(typ')))
{7}	size(uidt(dt_cv_float(typ(typ')))) > 0

C Proof scripts

Expanding the definition of `dt_cv_float`,

Rewriting using `dt_const_size`, matching in `*`,

we get 2 subgoals:

`ptr_to_nonzero_object.5.1:`

{-1}	<code>no_cv_references?(typ')</code>
{-2}	<code>cv_array?(typ')</code>
{-3}	<code>enum_underlying_integral?(typ')</code>
{-4}	<code>enum_constants?(typ')</code>
{-5}	<code>const_volatile?(typ')</code>
{-6}	<code>const_stutter?(typ')</code>
{-7}	<code>volatile_stutter?(typ')</code>
{-8}	<code>no_cv_class?(typ')</code>
{-9}	<code>no_cv_union?(typ')</code>
{-10}	<code>no_cv_function?(typ')</code>
{-11}	<code>no_cv_void?(typ')</code>
{-12}	<code>pointer?(typ')</code>
{-13}	<code>const?(typ(typ'))</code>
{-14}	<code>floating_point?(typ(typ(typ')))</code>
{1}	<code>cv(bitfield?)(typ(typ'))</code>
{2}	<code>bool?(typ(typ(typ')))</code>
{3}	<code>pointer?(typ(typ(typ')))</code>
{4}	<code>pointer_to_member?(typ(typ(typ')))</code>
{5}	<code>bitfield?(typ(typ(typ')))</code>
{6}	<code>volatile?(typ(typ(typ')))</code>
{7}	<code>size(uidt(dt_cv_float(typ(typ(typ'))))) > 0</code>

Expanding the definition of `dt_cv_float`,

Adding type constraints for `dt_floating_point(typ(typ(typ!1)))`,

Expanding the definition of `floating_point?`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `ptr_to_nonzero_object.5.1`.

ptr_to_nonzero_object.5.2:

{-1}	no_cv_references?(typ')
{-2}	cv_array?(typ')
{-3}	enum_underlying_integral?(typ')
{-4}	enum_constants?(typ')
{-5}	const_volatile?(typ')
{-6}	const_stutter?(typ')
{-7}	volatile_stutter?(typ')
{-8}	no_cv_class?(typ')
{-9}	no_cv_union?(typ')
{-10}	no_cv_function?(typ')
{-11}	no_cv_void?(typ')
{-12}	pointer?(typ')
{-13}	const?(typ(typ'))
{-14}	floating_point?(typ(typ(typ)))
{1}	$\exists (x_1 :$ <div style="padding-left: 40px;">(interpreted_data_type? <div style="padding-left: 40px;">[[(range_floating_point(cv_base(typ(typ(typ))))])])]) :</div></div>
{2}	$\text{cv}(\text{bitfield}^{\text{TRUE}})(\text{typ}(\text{typ}'))$
{3}	bool?(typ(typ'))
{4}	pointer?(typ(typ'))
{5}	pointer_to_member?(typ(typ'))
{6}	bitfield?(typ(typ'))
{7}	volatile?(typ(typ'))
{8}	size(uidt(dt_const(dt_cv_float(typ(typ(typ)))))) > 0

Instantiating the top quantifier in 1 with the terms: (dt_floating_point(cv_base(typ(typ(typ!1))))), we get 2 subgoals:

ptr_to_nonzero_object.5.2.1:

{-1}	no_cv_references?(typ')
{-2}	cv_array?(typ')
{-3}	enum_underlying_integral?(typ')
{-4}	enum_constants?(typ')
{-5}	const_volatile?(typ')
{-6}	const_stutter?(typ')
{-7}	volatile_stutter?(typ')
{-8}	no_cv_class?(typ')
{-9}	no_cv_union?(typ')
{-10}	no_cv_function?(typ')
{-11}	no_cv_void?(typ')
{-12}	pointer?(typ')
{-13}	const?(typ(typ'))
{-14}	floating_point?(typ(typ(typ)))
{1}	TRUE
{2}	cv(bitfield?)(typ(typ'))
{3}	bool?(typ(typ'))
{4}	pointer?(typ(typ'))
{5}	pointer_to_member?(typ(typ'))
{6}	bitfield?(typ(typ'))
{7}	volatile?(typ(typ'))
{8}	size(uidt(dt_const(dt_cv_float(typ(typ(typ)))))) > 0

C Proof scripts

which is trivially true.

This completes the proof of `ptr_to_nonzero_object.5.2.1`.

`ptr_to_nonzero_object.5.2.2`:

{-1}	<code>no_cv_references?(typ')</code>
{-2}	<code>cv_array?(typ')</code>
{-3}	<code>enum_underlying_integral?(typ')</code>
{-4}	<code>enum_constants?(typ')</code>
{-5}	<code>const_volatile?(typ')</code>
{-6}	<code>const_stutter?(typ')</code>
{-7}	<code>volatile_stutter?(typ')</code>
{-8}	<code>no_cv_class?(typ')</code>
{-9}	<code>no_cv_union?(typ')</code>
{-10}	<code>no_cv_function?(typ')</code>
{-11}	<code>no_cv_void?(typ')</code>
{-12}	<code>pointer?(typ')</code>
{-13}	<code>const?(typ(typ'))</code>
{-14}	<code>floating_point?(typ(typ(typ')))</code>
{1}	<code>floating_point?(cv_base(typ(typ(typ'))))</code>
{2}	<code>cv(bitfield?)(typ(typ'))</code>
{3}	<code>bool?(typ(typ(typ')))</code>
{4}	<code>pointer?(typ(typ(typ')))</code>
{5}	<code>pointer_to_member?(typ(typ(typ')))</code>
{6}	<code>bitfield?(typ(typ(typ')))</code>
{7}	<code>volatile?(typ(typ'))</code>
{8}	<code>size(uidt(dt_const(dt_cv_float(typ(typ(typ')))))) > 0</code>

Keeping (-14 1) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `ptr_to_nonzero_object.5.2.2`.

ptr_to_nonzero_object.6:

{-1}	no_cv_references?(typ')
{-2}	cv_array?(typ')
{-3}	enum_underlying_integral?(typ')
{-4}	enum_constants?(typ')
{-5}	const_volatile?(typ')
{-6}	const_stutter?(typ')
{-7}	volatile_stutter?(typ')
{-8}	no_cv_class?(typ')
{-9}	no_cv_union?(typ')
{-10}	no_cv_function?(typ')
{-11}	no_cv_void?(typ')
{-12}	pointer?(typ')
{-13}	floating_point?(typ(typ'))
{1}	cv(bitfield?)(typ(typ'))
{2}	reference?(typ(typ'))
{3}	void?(typ(typ'))
{4}	function?(typ(typ'))
{5}	array?(typ(typ'))
{6}	class?(typ(typ'))
{7}	union?(typ(typ'))
{8}	pointer?(typ(typ'))
{9}	pointer_to_member?(typ(typ'))
{10}	bitfield?(typ(typ'))
{11}	bool?(typ(typ'))
{12}	const?(typ(typ'))
{13}	size(uidt(dt_cv_float(typ(typ')))) > 0
{14}	volatile?(typ(typ'))

Expanding the definition of dt_cv_float,

Adding type constraints for dt_floating_point(typ(typ!1)),

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of ptr_to_nonzero_object.6.

`ptr_to_nonzero_object.7:`

{-1}	<code>no_cv_references?(typ')</code>
{-2}	<code>cv_array?(typ')</code>
{-3}	<code>enum_underlying_integral?(typ')</code>
{-4}	<code>enum_constants?(typ')</code>
{-5}	<code>const_volatile?(typ')</code>
{-6}	<code>const_stutter?(typ')</code>
{-7}	<code>volatile_stutter?(typ')</code>
{-8}	<code>no_cv_class?(typ')</code>
{-9}	<code>no_cv_union?(typ')</code>
{-10}	<code>no_cv_function?(typ')</code>
{-11}	<code>no_cv_void?(typ')</code>
{-12}	<code>pointer?(typ')</code>
{-13}	<code>floating_point?(typ(typ(typ')))</code>
{-14}	<code>volatile?(typ(typ'))</code>
{1}	<code>cv(bitfield?)(typ(typ'))</code>
{2}	<code>bool?(typ(typ(typ')))</code>
{3}	<code>pointer?(typ(typ(typ')))</code>
{4}	<code>pointer_to_member?(typ(typ(typ')))</code>
{5}	<code>bitfield?(typ(typ(typ')))</code>
{6}	<code>size(uidt(dt_cv_float(typ(typ')))) > 0</code>

Expanding the definition of `dt_cv_float`,
 Rewriting using `dt_volatile_size`, matching in `*`,
 we get 2 subgoals:

`ptr_to_nonzero_object.7.1:`

{-1}	<code>no_cv_references?(typ')</code>
{-2}	<code>cv_array?(typ')</code>
{-3}	<code>enum_underlying_integral?(typ')</code>
{-4}	<code>enum_constants?(typ')</code>
{-5}	<code>const_volatile?(typ')</code>
{-6}	<code>const_stutter?(typ')</code>
{-7}	<code>volatile_stutter?(typ')</code>
{-8}	<code>no_cv_class?(typ')</code>
{-9}	<code>no_cv_union?(typ')</code>
{-10}	<code>no_cv_function?(typ')</code>
{-11}	<code>no_cv_void?(typ')</code>
{-12}	<code>pointer?(typ')</code>
{-13}	<code>floating_point?(typ(typ(typ')))</code>
{-14}	<code>volatile?(typ(typ'))</code>
{1}	<code>cv(bitfield?)(typ(typ'))</code>
{2}	<code>bool?(typ(typ(typ')))</code>
{3}	<code>pointer?(typ(typ(typ')))</code>
{4}	<code>pointer_to_member?(typ(typ(typ')))</code>
{5}	<code>bitfield?(typ(typ(typ')))</code>
{6}	<code>size(uidt(dt_cv_float(typ(typ(typ'))))) > 0</code>

Expanding the definition of `dt_cv_float`,
 Adding type constraints for `dt_floating_point(typ(typ(typ!1)))`,
 Expanding the definition of `floating_point?`,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 This completes the proof of `ptr_to_nonzero_object.7.1`.

ptr_to_nonzero_object.7.2:

{-1}	no_cv_references?(typ')
{-2}	cv_array?(typ')
{-3}	enum_underlying_integral?(typ')
{-4}	enum_constants?(typ')
{-5}	const_volatile?(typ')
{-6}	const_stutter?(typ')
{-7}	volatile_stutter?(typ')
{-8}	no_cv_class?(typ')
{-9}	no_cv_union?(typ')
{-10}	no_cv_function?(typ')
{-11}	no_cv_void?(typ')
{-12}	pointer?(typ')
{-13}	floating_point?(typ(typ(typ')))
{-14}	volatile?(typ(typ'))
{1} $\exists (x_1 :$	
	(interpreted_data_type?
	[(((range_floating_point(cv_base(typ(typ(typ'))))))))]):
{2}	^{TRUE} cv(bitfield?)(typ(typ'))
{3}	bool?(typ(typ(typ')))
{4}	pointer?(typ(typ(typ')))
{5}	pointer_to_member?(typ(typ(typ')))
{6}	bitfield?(typ(typ(typ')))
{7}	size(uidt(dt_volatile(dt_cv_float(typ(typ(typ')))))) > 0

Instantiating the top quantifier in 1 with the terms: (dt_floating_point(cv_base(typ(typ(typ!1))))), we get 2 subgoals:

ptr_to_nonzero_object.7.2.1:

{-1}	no_cv_references?(typ')
{-2}	cv_array?(typ')
{-3}	enum_underlying_integral?(typ')
{-4}	enum_constants?(typ')
{-5}	const_volatile?(typ')
{-6}	const_stutter?(typ')
{-7}	volatile_stutter?(typ')
{-8}	no_cv_class?(typ')
{-9}	no_cv_union?(typ')
{-10}	no_cv_function?(typ')
{-11}	no_cv_void?(typ')
{-12}	pointer?(typ')
{-13}	floating_point?(typ(typ(typ')))
{-14}	volatile?(typ(typ'))
{1} TRUE	
{2}	cv(bitfield?)(typ(typ'))
{3}	bool?(typ(typ(typ')))
{4}	pointer?(typ(typ(typ')))
{5}	pointer_to_member?(typ(typ(typ')))
{6}	bitfield?(typ(typ(typ')))
{7}	size(uidt(dt_volatile(dt_cv_float(typ(typ(typ')))))) > 0

which is trivially true.

This completes the proof of `ptr_to_nonzero_object.7.2.1`.

`ptr_to_nonzero_object.7.2.2`:

{-1}	<code>no_cv_references?(typ')</code>
{-2}	<code>cv_array?(typ')</code>
{-3}	<code>enum_underlying_integral?(typ')</code>
{-4}	<code>enum_constants?(typ')</code>
{-5}	<code>const_volatile?(typ')</code>
{-6}	<code>const_stutter?(typ')</code>
{-7}	<code>volatile_stutter?(typ')</code>
{-8}	<code>no_cv_class?(typ')</code>
{-9}	<code>no_cv_union?(typ')</code>
{-10}	<code>no_cv_function?(typ')</code>
{-11}	<code>no_cv_void?(typ')</code>
{-12}	<code>pointer?(typ')</code>
{-13}	<code>floating_point?(typ(typ(typ')))</code>
{-14}	<code>volatile?(typ(typ'))</code>
{1}	<code>floating_point?(cv_base(typ(typ(typ'))))</code>
{2}	<code>cv(bitfield?)(typ(typ'))</code>
{3}	<code>bool?(typ(typ(typ')))</code>
{4}	<code>pointer?(typ(typ(typ')))</code>
{5}	<code>pointer_to_member?(typ(typ(typ')))</code>
{6}	<code>bitfield?(typ(typ(typ')))</code>
{7}	<code>size(uidt(dt_volatile(dt_cv_float(typ(typ(typ')))))) > 0</code>

Keeping (-13 1) and hiding *,

Expanding the definition of `cv_base`,

Expanding the definition of `floating_point?`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `ptr_to_nonzero_object.7.2.2`.

`ptr_to_nonzero_object.8`:

{-1}	<code>no_cv_references?(typ')</code>
{-2}	<code>cv_array?(typ')</code>
{-3}	<code>enum_underlying_integral?(typ')</code>
{-4}	<code>enum_constants?(typ')</code>
{-5}	<code>const_volatile?(typ')</code>
{-6}	<code>const_stutter?(typ')</code>
{-7}	<code>volatile_stutter?(typ')</code>
{-8}	<code>no_cv_class?(typ')</code>
{-9}	<code>no_cv_union?(typ')</code>
{-10}	<code>no_cv_function?(typ')</code>
{-11}	<code>no_cv_void?(typ')</code>
{-12}	<code>pointer?(typ')</code>
{-13}	<code>const?(typ(typ'))</code>
{-14}	<code>bitfield?(typ(typ(typ')))</code>
{1}	<code>cv(bitfield?)(typ(typ'))</code>
{2}	<code>size(uidt(dt_cv_bitfield(typ(typ')))) > 0</code>

Expanding the definition of `cv`,

Expanding the definition of `c`,

which is trivially true.

This completes the proof of `ptr_to_nonzero_object.8`.

ptr_to_nonzero_object.9:

{-1}	no_cv_references?(typ')
{-2}	cv_array?(typ')
{-3}	enum_underlying_integral?(typ')
{-4}	enum_constants?(typ')
{-5}	const_volatile?(typ')
{-6}	const_stutter?(typ')
{-7}	volatile_stutter?(typ')
{-8}	no_cv_class?(typ')
{-9}	no_cv_union?(typ')
{-10}	no_cv_function?(typ')
{-11}	no_cv_void?(typ')
{-12}	pointer?(typ')
{-13}	bitfield?(typ(typ(typ')))
{-14}	volatile?(typ(typ'))
{1}	cv(bitfield?)(typ(typ'))
{2}	size(uidt(dt_cv_bitfield(typ(typ')))) > 0

Expanding the definition of cv,

Expanding the definition of c,

Expanding the definition of v,

which is trivially true.

This completes the proof of ptr_to_nonzero_object.9.

ptr_to_nonzero_object.10:

{-1}	no_cv_references?(typ')
{-2}	cv_array?(typ')
{-3}	enum_underlying_integral?(typ')
{-4}	enum_constants?(typ')
{-5}	const_volatile?(typ')
{-6}	const_stutter?(typ')
{-7}	volatile_stutter?(typ')
{-8}	no_cv_class?(typ')
{-9}	no_cv_union?(typ')
{-10}	no_cv_function?(typ')
{-11}	no_cv_void?(typ')
{-12}	pointer?(typ')
{-13}	const?(typ(typ'))
{-14}	pointer_to_member?(typ(typ(typ')))
{1}	cv(bitfield?)(typ(typ'))
{2}	size(uidt(dt_cv_ptm(typ(typ')))) > 0

Expanding the definition of dt_cv_ptm,

Rewriting using dt_const_size, matching in *,

we get 2 subgoals:

C Proof scripts

`ptr_to_nonzero_object.10.1:`

{-1}	<code>no_cv_references?(typ')</code>
{-2}	<code>cv_array?(typ')</code>
{-3}	<code>enum_underlying_integral?(typ')</code>
{-4}	<code>enum_constants?(typ')</code>
{-5}	<code>const_volatile?(typ')</code>
{-6}	<code>const_stutter?(typ')</code>
{-7}	<code>volatile_stutter?(typ')</code>
{-8}	<code>no_cv_class?(typ')</code>
{-9}	<code>no_cv_union?(typ')</code>
{-10}	<code>no_cv_function?(typ')</code>
{-11}	<code>no_cv_void?(typ')</code>
{-12}	<code>pointer?(typ')</code>
{-13}	<code>const?(typ(typ'))</code>
{-14}	<code>pointer_to_member?(typ(typ(typ')))</code>
{1}	<code>cv(bitfield?)(typ(typ'))</code>
{2}	<code>size(uidt(dt_cv_ptm(typ(typ(typ'))))) > 0</code>

Expanding the definition of `dt_cv_ptm`,

Adding type constraints for `dt_ptm(typ(typ(typ!1)))`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `ptr_to_nonzero_object.10.1`.

`ptr_to_nonzero_object.10.2:`

{-1}	<code>no_cv_references?(typ')</code>
{-2}	<code>cv_array?(typ')</code>
{-3}	<code>enum_underlying_integral?(typ')</code>
{-4}	<code>enum_constants?(typ')</code>
{-5}	<code>const_volatile?(typ')</code>
{-6}	<code>const_stutter?(typ')</code>
{-7}	<code>volatile_stutter?(typ')</code>
{-8}	<code>no_cv_class?(typ')</code>
{-9}	<code>no_cv_union?(typ')</code>
{-10}	<code>no_cv_function?(typ')</code>
{-11}	<code>no_cv_void?(typ')</code>
{-12}	<code>pointer?(typ')</code>
{-13}	<code>const?(typ(typ'))</code>
{-14}	<code>pointer_to_member?(typ(typ(typ')))</code>
{1}	$\exists (x_1: \text{interpreted_data_type?}[\text{((range_ptm(cv_base(typ(typ(typ'))))))}]): \text{TRUE}$
{2}	<code>cv(bitfield?)(typ(typ'))</code>
{3}	<code>size(uidt(dt_const(dt_cv_ptm(typ(typ(typ'))))) > 0</code>

Instantiating the top quantifier in 1 with the terms: `(dt_ptm(cv_base(typ(typ(typ!1)))))`,

we get 2 subgoals:

ptr_to_nonzero_object.10.2.1:

{-1}	no_cv_references?(typ')
{-2}	cv_array?(typ')
{-3}	enum_underlying_integral?(typ')
{-4}	enum_constants?(typ')
{-5}	const_volatile?(typ')
{-6}	const_stutter?(typ')
{-7}	volatile_stutter?(typ')
{-8}	no_cv_class?(typ')
{-9}	no_cv_union?(typ')
{-10}	no_cv_function?(typ')
{-11}	no_cv_void?(typ')
{-12}	pointer?(typ')
{-13}	const?(typ(typ'))
{-14}	pointer_to_member?(typ(typ(typ')))
{1}	TRUE
{2}	cv(bitfield?)(typ(typ'))
{3}	size(uidt(dt_const(dt_cv_ptm(typ(typ(typ')))))) > 0

which is trivially true.

This completes the proof of ptr_to_nonzero_object.10.2.1.

ptr_to_nonzero_object.10.2.2:

{-1}	no_cv_references?(typ')
{-2}	cv_array?(typ')
{-3}	enum_underlying_integral?(typ')
{-4}	enum_constants?(typ')
{-5}	const_volatile?(typ')
{-6}	const_stutter?(typ')
{-7}	volatile_stutter?(typ')
{-8}	no_cv_class?(typ')
{-9}	no_cv_union?(typ')
{-10}	no_cv_function?(typ')
{-11}	no_cv_void?(typ')
{-12}	pointer?(typ')
{-13}	const?(typ(typ'))
{-14}	pointer_to_member?(typ(typ(typ')))
{1}	pointer_to_member?(cv_base(typ(typ(typ'))))
{2}	cv(bitfield?)(typ(typ'))
{3}	size(uidt(dt_const(dt_cv_ptm(typ(typ(typ')))))) > 0

Keeping (-14 1) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of ptr_to_nonzero_object.10.2.2.

C Proof scripts

`ptr_to_nonzero_object.11:`

{-1}	<code>no_cv_references?(typ')</code>
{-2}	<code>cv_array?(typ')</code>
{-3}	<code>enum_underlying_integral?(typ')</code>
{-4}	<code>enum_constants?(typ')</code>
{-5}	<code>const_volatile?(typ')</code>
{-6}	<code>const_stutter?(typ')</code>
{-7}	<code>volatile_stutter?(typ')</code>
{-8}	<code>no_cv_class?(typ')</code>
{-9}	<code>no_cv_union?(typ')</code>
{-10}	<code>no_cv_function?(typ')</code>
{-11}	<code>no_cv_void?(typ')</code>
{-12}	<code>pointer?(typ')</code>
{-13}	<code>pointer_to_member?(typ(typ(typ')))</code>
{-14}	<code>volatile?(typ(typ'))</code>
<hr/>	
{1}	<code>cv(bitfield?)(typ(typ'))</code>
{2}	<code>size(uidt(dt_cv_ptm(typ(typ')))) > 0</code>

Expanding the definition of `dt_cv_ptm`,

Rewriting using `dt_volatile_size`, matching in `*`,

we get 2 subgoals:

`ptr_to_nonzero_object.11.1:`

{-1}	<code>no_cv_references?(typ')</code>
{-2}	<code>cv_array?(typ')</code>
{-3}	<code>enum_underlying_integral?(typ')</code>
{-4}	<code>enum_constants?(typ')</code>
{-5}	<code>const_volatile?(typ')</code>
{-6}	<code>const_stutter?(typ')</code>
{-7}	<code>volatile_stutter?(typ')</code>
{-8}	<code>no_cv_class?(typ')</code>
{-9}	<code>no_cv_union?(typ')</code>
{-10}	<code>no_cv_function?(typ')</code>
{-11}	<code>no_cv_void?(typ')</code>
{-12}	<code>pointer?(typ')</code>
{-13}	<code>pointer_to_member?(typ(typ(typ')))</code>
{-14}	<code>volatile?(typ(typ'))</code>
<hr/>	
{1}	<code>cv(bitfield?)(typ(typ'))</code>
{2}	<code>size(uidt(dt_cv_ptm(typ(typ')))) > 0</code>

Expanding the definition of `dt_cv_ptm`,

Adding type constraints for `dt_ptm(typ(typ!1))`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `ptr_to_nonzero_object.11.1`.

ptr_to_nonzero_object.11.2:

{-1}	no_cv_references?(typ')
{-2}	cv_array?(typ')
{-3}	enum_underlying_integral?(typ')
{-4}	enum_constants?(typ')
{-5}	const_volatile?(typ')
{-6}	const_stutter?(typ')
{-7}	volatile_stutter?(typ')
{-8}	no_cv_class?(typ')
{-9}	no_cv_union?(typ')
{-10}	no_cv_function?(typ')
{-11}	no_cv_void?(typ')
{-12}	pointer?(typ')
{-13}	pointer_to_member?(typ(typ(typ')))
{-14}	volatile?(typ(typ'))
{1}	$\exists (x_1 : (\text{interpreted_data_type?}[\text{((range_ptm(cv_base(typ(typ(typ'))))))}])) : \text{TRUE}$
{2}	cv(bitfield?)(typ(typ'))
{3}	size(uidt(dt_volatile(dt_cv_ptm(typ(typ(typ')))))) > 0

Instantiating the top quantifier in 1 with the terms: (dt_ptm(cv_base(typ(typ(typ!1))))),

we get 2 subgoals:

ptr_to_nonzero_object.11.2.1:

{-1}	no_cv_references?(typ')
{-2}	cv_array?(typ')
{-3}	enum_underlying_integral?(typ')
{-4}	enum_constants?(typ')
{-5}	const_volatile?(typ')
{-6}	const_stutter?(typ')
{-7}	volatile_stutter?(typ')
{-8}	no_cv_class?(typ')
{-9}	no_cv_union?(typ')
{-10}	no_cv_function?(typ')
{-11}	no_cv_void?(typ')
{-12}	pointer?(typ')
{-13}	pointer_to_member?(typ(typ(typ')))
{-14}	volatile?(typ(typ'))
{1}	TRUE
{2}	cv(bitfield?)(typ(typ'))
{3}	size(uidt(dt_volatile(dt_cv_ptm(typ(typ(typ')))))) > 0

which is trivially true.

This completes the proof of ptr_to_nonzero_object.11.2.1.

C Proof scripts

`ptr_to_nonzero_object.11.2.2:`

{-1}	<code>no_cv_references?(typ')</code>
{-2}	<code>cv_array?(typ')</code>
{-3}	<code>enum_underlying_integral?(typ')</code>
{-4}	<code>enum_constants?(typ')</code>
{-5}	<code>const_volatile?(typ')</code>
{-6}	<code>const_stutter?(typ')</code>
{-7}	<code>volatile_stutter?(typ')</code>
{-8}	<code>no_cv_class?(typ')</code>
{-9}	<code>no_cv_union?(typ')</code>
{-10}	<code>no_cv_function?(typ')</code>
{-11}	<code>no_cv_void?(typ')</code>
{-12}	<code>pointer?(typ')</code>
{-13}	<code>pointer_to_member?(typ(typ(typ')))</code>
{-14}	<code>volatile?(typ(typ'))</code>
<hr/>	
{1}	<code>pointer_to_member?(cv_base(typ(typ(typ'))))</code>
{2}	<code>cv(bitfield?)(typ(typ'))</code>
{3}	<code>size(uidt(dt_volatile(dt_cv_ptm(typ(typ(typ')))))) > 0</code>

Keeping (-13 1) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `ptr_to_nonzero_object.11.2.2`.

`ptr_to_nonzero_object.12:`

{-1}	<code>no_cv_references?(typ')</code>
{-2}	<code>cv_array?(typ')</code>
{-3}	<code>enum_underlying_integral?(typ')</code>
{-4}	<code>enum_constants?(typ')</code>
{-5}	<code>const_volatile?(typ')</code>
{-6}	<code>const_stutter?(typ')</code>
{-7}	<code>volatile_stutter?(typ')</code>
{-8}	<code>no_cv_class?(typ')</code>
{-9}	<code>no_cv_union?(typ')</code>
{-10}	<code>no_cv_function?(typ')</code>
{-11}	<code>no_cv_void?(typ')</code>
{-12}	<code>pointer?(typ')</code>
{-13}	<code>const?(typ(typ'))</code>
{-14}	<code>pointer?(typ(typ(typ')))</code>
<hr/>	
{1}	<code>cv(bitfield?)(typ(typ'))</code>
{2}	<code>size(uidt(dt_cv_pointer(typ(typ')))) > 0</code>

Expanding the definition of `dt_cv_pointer`,

Rewriting using `dt_const_size`, matching in *,

we get 2 subgoals:

ptr_to_nonzero_object.12.1:

{-1}	no_cv_references?(typ')
{-2}	cv_array?(typ')
{-3}	enum_underlying_integral?(typ')
{-4}	enum_constants?(typ')
{-5}	const_volatile?(typ')
{-6}	const_stutter?(typ')
{-7}	volatile_stutter?(typ')
{-8}	no_cv_class?(typ')
{-9}	no_cv_union?(typ')
{-10}	no_cv_function?(typ')
{-11}	no_cv_void?(typ')
{-12}	pointer?(typ')
{-13}	const?(typ(typ'))
{-14}	pointer?(typ(typ(typ')))
{1}	cv(bitfield?)(typ(typ'))
{2}	size(uidt(dt_cv_pointer(typ(typ(typ'))))) > 0

Expanding the definition of dt_cv_pointer,

Adding type constraints for dt_pointer(typ(typ(typ!1))),

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of ptr_to_nonzero_object.12.1.

ptr_to_nonzero_object.12.2:

{-1}	no_cv_references?(typ')
{-2}	cv_array?(typ')
{-3}	enum_underlying_integral?(typ')
{-4}	enum_constants?(typ')
{-5}	const_volatile?(typ')
{-6}	const_stutter?(typ')
{-7}	volatile_stutter?(typ')
{-8}	no_cv_class?(typ')
{-9}	no_cv_union?(typ')
{-10}	no_cv_function?(typ')
{-11}	no_cv_void?(typ')
{-12}	pointer?(typ')
{-13}	const?(typ(typ'))
{-14}	pointer?(typ(typ(typ')))
{1}	$\exists (x_1: \text{interpreted_data_type?}[\text{((range_pointer(cv_base(typ(typ(typ'))))))}]):$
	TRUE
{2}	cv(bitfield?)(typ(typ'))
{3}	size(uidt(dt_const(dt_cv_pointer(typ(typ(typ'))))) > 0

Instantiating the top quantifier in 1 with the terms: (dt_pointer(cv_base(typ(typ(typ!1))))),

we get 2 subgoals:

C Proof scripts

`ptr_to_nonzero_object.12.2.1:`

{-1}	<code>no_cv_references?(typ')</code>
{-2}	<code>cv_array?(typ')</code>
{-3}	<code>enum_underlying_integral?(typ')</code>
{-4}	<code>enum_constants?(typ')</code>
{-5}	<code>const_volatile?(typ')</code>
{-6}	<code>const_stutter?(typ')</code>
{-7}	<code>volatile_stutter?(typ')</code>
{-8}	<code>no_cv_class?(typ')</code>
{-9}	<code>no_cv_union?(typ')</code>
{-10}	<code>no_cv_function?(typ')</code>
{-11}	<code>no_cv_void?(typ')</code>
{-12}	<code>pointer?(typ')</code>
{-13}	<code>const?(typ(typ'))</code>
{-14}	<code>pointer?(typ(typ(typ')))</code>
{1}	<code>TRUE</code>
{2}	<code>cv(bitfield?)(typ(typ'))</code>
{3}	<code>size(uidt(dt_const(dt_cv_pointer(typ(typ(typ')))))) > 0</code>

which is trivially true.

This completes the proof of `ptr_to_nonzero_object.12.2.1`.

`ptr_to_nonzero_object.12.2.2:`

{-1}	<code>no_cv_references?(typ')</code>
{-2}	<code>cv_array?(typ')</code>
{-3}	<code>enum_underlying_integral?(typ')</code>
{-4}	<code>enum_constants?(typ')</code>
{-5}	<code>const_volatile?(typ')</code>
{-6}	<code>const_stutter?(typ')</code>
{-7}	<code>volatile_stutter?(typ')</code>
{-8}	<code>no_cv_class?(typ')</code>
{-9}	<code>no_cv_union?(typ')</code>
{-10}	<code>no_cv_function?(typ')</code>
{-11}	<code>no_cv_void?(typ')</code>
{-12}	<code>pointer?(typ')</code>
{-13}	<code>const?(typ(typ'))</code>
{-14}	<code>pointer?(typ(typ(typ')))</code>
{1}	<code>pointer?(cv_base(typ(typ(typ'))))</code>
{2}	<code>cv(bitfield?)(typ(typ'))</code>
{3}	<code>size(uidt(dt_const(dt_cv_pointer(typ(typ(typ')))))) > 0</code>

Keeping (-14 1) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `ptr_to_nonzero_object.12.2.2`.

ptr_to_nonzero_object.13:

{-1}	no_cv_references?(typ')
{-2}	cv_array?(typ')
{-3}	enum_underlying_integral?(typ')
{-4}	enum_constants?(typ')
{-5}	const_volatile?(typ')
{-6}	const_stutter?(typ')
{-7}	volatile_stutter?(typ')
{-8}	no_cv_class?(typ')
{-9}	no_cv_union?(typ')
{-10}	no_cv_function?(typ')
{-11}	no_cv_void?(typ')
{-12}	pointer?(typ')
{-13}	pointer?(typ(typ(typ')))
{-14}	volatile?(typ(typ'))
<hr/>	
{1}	cv(bitfield?)(typ(typ'))
{2}	size(uidt(dt_cv_pointer(typ(typ')))) > 0

Expanding the definition of dt_cv_pointer,

Rewriting using dt_volatile_size, matching in *,

we get 2 subgoals:

ptr_to_nonzero_object.13.1:

{-1}	no_cv_references?(typ')
{-2}	cv_array?(typ')
{-3}	enum_underlying_integral?(typ')
{-4}	enum_constants?(typ')
{-5}	const_volatile?(typ')
{-6}	const_stutter?(typ')
{-7}	volatile_stutter?(typ')
{-8}	no_cv_class?(typ')
{-9}	no_cv_union?(typ')
{-10}	no_cv_function?(typ')
{-11}	no_cv_void?(typ')
{-12}	pointer?(typ')
{-13}	pointer?(typ(typ(typ')))
{-14}	volatile?(typ(typ'))
<hr/>	
{1}	cv(bitfield?)(typ(typ'))
{2}	size(uidt(dt_cv_pointer(typ(typ')))) > 0

Expanding the definition of dt_cv_pointer,

Adding type constraints for dt_pointer(typ(typ(typ!))),

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of ptr_to_nonzero_object.13.1.

C Proof scripts

`ptr_to_nonzero_object.13.2:`

{-1}	<code>no_cv_references?(typ')</code>
{-2}	<code>cv_array?(typ')</code>
{-3}	<code>enum_underlying_integral?(typ')</code>
{-4}	<code>enum_constants?(typ')</code>
{-5}	<code>const_volatile?(typ')</code>
{-6}	<code>const_stutter?(typ')</code>
{-7}	<code>volatile_stutter?(typ')</code>
{-8}	<code>no_cv_class?(typ')</code>
{-9}	<code>no_cv_union?(typ')</code>
{-10}	<code>no_cv_function?(typ')</code>
{-11}	<code>no_cv_void?(typ')</code>
{-12}	<code>pointer?(typ')</code>
{-13}	<code>pointer?(typ(typ(typ')))</code>
{-14}	<code>volatile?(typ(typ'))</code>
{1} $\exists (x_1 : (\text{interpreted_data_type?} [((\text{range_pointer}(\text{cv_base}(\text{typ}(\text{typ}(\text{typ}'))))))])) :$	
{2}	$\text{cv}(\overset{\text{TRUE}}{\text{bitfield?}}(\text{typ}(\text{typ}')))$
{3}	$\text{size}(\text{uidt}(\text{dt_volatile}(\text{dt_cv_pointer}(\text{typ}(\text{typ}(\text{typ}')))))) > 0$

Instantiating the top quantifier in 1 with the terms: $(\text{dt_pointer}(\text{cv_base}(\text{typ}(\text{typ}(\text{typ}1))))$,

we get 2 subgoals:

`ptr_to_nonzero_object.13.2.1:`

{-1}	<code>no_cv_references?(typ')</code>
{-2}	<code>cv_array?(typ')</code>
{-3}	<code>enum_underlying_integral?(typ')</code>
{-4}	<code>enum_constants?(typ')</code>
{-5}	<code>const_volatile?(typ')</code>
{-6}	<code>const_stutter?(typ')</code>
{-7}	<code>volatile_stutter?(typ')</code>
{-8}	<code>no_cv_class?(typ')</code>
{-9}	<code>no_cv_union?(typ')</code>
{-10}	<code>no_cv_function?(typ')</code>
{-11}	<code>no_cv_void?(typ')</code>
{-12}	<code>pointer?(typ')</code>
{-13}	<code>pointer?(typ(typ(typ')))</code>
{-14}	<code>volatile?(typ(typ'))</code>
{1} TRUE	
{2}	$\text{cv}(\text{bitfield?})(\text{typ}(\text{typ}'))$
{3}	$\text{size}(\text{uidt}(\text{dt_volatile}(\text{dt_cv_pointer}(\text{typ}(\text{typ}(\text{typ}')))))) > 0$

which is trivially true.

This completes the proof of `ptr_to_nonzero_object.13.2.1`.

ptr_to_nonzero_object.13.2.2:

{-1}	no_cv_references?(typ')
{-2}	cv_array?(typ')
{-3}	enum_underlying_integral?(typ')
{-4}	enum_constants?(typ')
{-5}	const_volatile?(typ')
{-6}	const_stutter?(typ')
{-7}	volatile_stutter?(typ')
{-8}	no_cv_class?(typ')
{-9}	no_cv_union?(typ')
{-10}	no_cv_function?(typ')
{-11}	no_cv_void?(typ')
{-12}	pointer?(typ')
{-13}	pointer?(typ(typ(typ')))
{-14}	volatile?(typ(typ'))
{1}	pointer?(cv_base(typ(typ(typ'))))
{2}	cv(bitfield?)(typ(typ'))
{3}	size(uidt(dt_volatile(dt_cv_pointer(typ(typ(typ')))))) > 0

Keeping (-13 1) and hiding *,
 Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of ptr_to_nonzero_object.13.2.2.
 Q.E.D.

C.11.21 ArithmeticExpressions.minus_ptr_typ_is_Cpp_Type_TCC1

Terse proof for minus_ptr_typ_is_Cpp_Type_TCC1.

minus_ptr_typ_is_Cpp_Type_TCC1:

{1}	\forall (typ: Cpp_Subtype(cv(extend[Cpp_Type_, (pointer?), bool, FALSE] (λ (p: (pointer?): \neg void?(typ(p)) \wedge \neg function?(typ(p)))))): array?(cv_base(typ)) \vee pointer?(cv_base(typ)) \vee reference?(cv_base(typ)) \vee bitfield?(cv_base(typ)) \vee enum?(cv_base(typ)) \vee pointer_to_member?(cv_base(typ)) \vee const?(cv_base(typ)) \vee volatile?(cv_base(typ))
-----	---

Repeatedly Skolemizing and flattening,
 Hiding formulas: -1,
 Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of minus_ptr_typ_is_Cpp_Type_TCC1.
 Q.E.D.

C.11.22 ArithmeticExpressions.minus_ptr_typ_is_Cpp_Type

Terse proof for minus_ptr_typ_is_Cpp_Type.

C Proof scripts

minus_ptr_typ_is_Cpp_Type:

$\{1\} \quad \forall (\text{typ}:$ $\text{Cpp_Subtype}(\text{cv}(\text{extend}[\text{Cpp_Type_}, (\text{pointer?}), \text{bool}, \text{FALSE}]$ $(\lambda (p: (\text{pointer?})): \neg \text{void?}(\text{typ}(p)) \wedge \neg \text{function?}(\text{typ}(p))))):$ $\text{Cpp_Type?}(\text{typ}(\text{cv_base}(\text{typ})))$
--

Repeatedly Skolemizing and flattening,

Expanding the definition of Cpp_Type?,

Repeatedly Skolemizing and flattening,

Instantiating the top quantifier in -1 with the terms: (t!1),

Splitting conjunctions,

we get 2 subgoals:

minus_ptr_typ_is_Cpp_Type.1:

{-1}	no_pointers_to_bitfield?(<i>t'</i>) ∧ no_pointers_to_references?(<i>t'</i>) ∧ no_cv_references?(<i>t'</i>) ∧ no_reference_to_reference?(<i>t'</i>) ∧ no_reference_to_bitfields?(<i>t'</i>) ∧ no_pointer_to_member_to_reference?(<i>t'</i>) ∧ no_pointer_to_member_to_cv_void?(<i>t'</i>) ∧ no_cv_void_parameter?(<i>t'</i>) ∧ no_array_of_references?(<i>t'</i>) ∧ no_array_of_cv_void?(<i>t'</i>) ∧ no_array_of_function?(<i>t'</i>) ∧ no_array_of_abstract_class?(<i>t'</i>) ∧ no_pointer_or_ref_to_incomplete_array_parameter?(<i>t'</i>) ∧ no_array_return_type?(<i>t'</i>) ∧ no_function_return_type?(<i>t'</i>) ∧ bitfield_underlying_integral_or_enum_type?(<i>t'</i>) ∧ cv_array?(<i>t'</i>) ∧ enum_underlying_integral?(<i>t'</i>) ∧ enum_constants?(<i>t'</i>) ∧ const_volatile?(<i>t'</i>) ∧ const_stutter?(<i>t'</i>) ∧ volatile_stutter?(<i>t'</i>) ∧ no_cv_class?(<i>t'</i>) ∧ no_cv_union?(<i>t'</i>) ∧ no_cv_function?(<i>t'</i>) ∧ no_reference_to_void?(<i>t'</i>) ∧ no_cv_void?(<i>t'</i>) ∧ no_array_of_bitfields?(<i>t'</i>)
{-2}	cv(extend[Cpp_Type_, (pointer?), bool, FALSE] (λ (<i>p</i> : (pointer?):) : ¬ void?(typ(<i>p</i>)) ∧ ¬ function?(typ(<i>p</i>)))) (typ')
{-3}	subterm(<i>t'</i> , typ(cv_base(typ')))
{1}	no_pointers_to_bitfield?(<i>t'</i>) ∧ no_pointers_to_references?(<i>t'</i>) ∧ no_cv_references?(<i>t'</i>) ∧ no_reference_to_reference?(<i>t'</i>) ∧ no_reference_to_bitfields?(<i>t'</i>) ∧ no_pointer_to_member_to_reference?(<i>t'</i>) ∧ no_pointer_to_member_to_cv_void?(<i>t'</i>) ∧ no_cv_void_parameter?(<i>t'</i>) ∧ no_array_of_references?(<i>t'</i>) ∧ no_array_of_cv_void?(<i>t'</i>) ∧ no_array_of_function?(<i>t'</i>) ∧ no_array_of_abstract_class?(<i>t'</i>) ∧ no_pointer_or_ref_to_incomplete_array_parameter?(<i>t'</i>) ∧ no_array_return_type?(<i>t'</i>) ∧ no_function_return_type?(<i>t'</i>) ∧ bitfield_underlying_integral_or_enum_type?(<i>t'</i>) ∧ cv_array?(<i>t'</i>) ∧ enum_underlying_integral?(<i>t'</i>) ∧ enum_constants?(<i>t'</i>) ∧ const_volatile?(<i>t'</i>) ∧ const_stutter?(<i>t'</i>) ∧ volatile_stutter?(<i>t'</i>) ∧ no_cv_class?(<i>t'</i>) ∧ no_cv_union?(<i>t'</i>) ∧ no_cv_function?(<i>t'</i>) ∧ no_reference_to_void?(<i>t'</i>) ∧ no_cv_void?(<i>t'</i>) ∧ no_array_of_bitfields?(<i>t'</i>)

C Proof scripts

which is trivially true.

This completes the proof of `minus_ptr_typ_is_Cpp_Type.1`.

`minus_ptr_typ_is_Cpp_Type.2`:

<pre>{-1} cv(extend[Cpp_Type_, (pointer?), bool, FALSE] (λ (p: (pointer?): ¬ void?(typ(p)) ∧ ¬ function?(typ(p)))) (typ')) {-2} subterm(t', typ(cv_base(typ')))</pre>	<hr style="border: 0.5px solid black;"/> <pre>{1} subterm(t', typ') {2} no_pointers_to_bitfield?(t') ∧ no_pointers_to_references?(t') ∧ no_cv_references?(t') ∧ no_reference_to_reference?(t') ∧ no_reference_to_bitfields?(t') ∧ no_pointer_to_member_to_reference?(t') ∧ no_pointer_to_member_to_cv_void?(t') ∧ no_cv_void_parameter?(t') ∧ no_array_of_references?(t') ∧ no_array_of_cv_void?(t') ∧ no_array_of_function?(t') ∧ no_array_of_abstract_class?(t') ∧ no_pointer_or_ref_to_incomplete_array_parameter?(t') ∧ no_array_return_type?(t') ∧ no_function_return_type?(t') ∧ bitfield_underlying_integral_or_enum_type?(t') ∧ cv_array?(t') ∧ enum_underlying_integral?(t') ∧ enum_constants?(t') ∧ const_volatile?(t') ∧ const_stutter?(t') ∧ volatile_stutter?(t') ∧ no_cv_class?(t') ∧ no_cv_union?(t') ∧ no_cv_function?(t') ∧ no_reference_to_void?(t') ∧ no_cv_void?(t') ∧ no_array_of_bitfields?(t')</pre>
---	---

Hiding formulas: 2,

Using lemma `subterm_cv_base`,

Using lemma `subterm_transitive`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Keeping (-2 -3 1 2) and hiding *,

Using lemma `cv_base_result`,

we get 2 subgoals:

minus_ptr_typ_is_Cpp_Type.2.1:

{-1}	extend[Cpp_Type_, (pointer?), bool, FALSE]
	(λ (p: (pointer?): ¬ void?(typ(p)) ∧ ¬ function?(typ(p)))
	(cv_base(typ'))
{-2}	cv(extend[Cpp_Type_, (pointer?), bool, FALSE]
	(λ (p: (pointer?): ¬ void?(typ(p)) ∧ ¬ function?(typ(p)))
	(typ'))
{-3}	subterm(t', typ(cv_base(typ')))
{1}	subterm(t', cv_base(typ'))
{2}	subterm(t', typ')

Expanding the definition of extend,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 Expanding the definition of subterm,
 which is trivially true.

This completes the proof of minus_ptr_typ_is_Cpp_Type.2.1.

minus_ptr_typ_is_Cpp_Type.2.2:

{-1}	cv(extend[Cpp_Type_, (pointer?), bool, FALSE]
	(λ (p: (pointer?): ¬ void?(typ(p)) ∧ ¬ function?(typ(p)))
	(typ'))
{-2}	subterm(t', typ(cv_base(typ')))
{1}	(extend [Cpp_Type_, (pointer?), bool, FALSE](λ (p: (pointer?): ¬ void?(typ(p)) ∧ ¬ function?(typ(p))
{2}	subterm(t', cv_base(typ'))
{3}	subterm(t', typ')

Keeping (1) and hiding *,
 Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of minus_ptr_typ_is_Cpp_Type.2.2.
 Q.E.D.

C.11.23 ArithmeticExpressions.minus_ptr_ptr_TCC1

Terse proof for minus_ptr_ptr_TCC1.

minus_ptr_ptr_TCC1:

{1}	∃ (typ:
	Cpp_Subtype(cv(extend[Cpp_Type_, (pointer?), bool, FALSE]
	(λ (p: (pointer?):
	¬ void?(typ(p)) ∧ ¬ function?(typ(p))))),
	ptr_val1: (range_pointer(cv_base(typ))),
	ptr_val2: (range_pointer(cv_base(typ))))):
	array?(typ(cv_base(typ))) ⊃ Cpp_Type?(typ(cv_base(typ)))

Repeatedly Skolemizing and flattening,
 Using lemma minus_ptr_typ_is_Cpp_Type,
 This completes the proof of minus_ptr_ptr_TCC1.
 Q.E.D.

C.11.24 ArithmeticExpressions.minus_ptr_ptr_TCC2

Terse proof for minus_ptr_ptr_TCC2.

minus_ptr_ptr_TCC2:

```
{1}  ∇ (typ:
      Cpp_Subtype(cv(extend[Cpp_Type_, (pointer?), bool, FALSE]
                        (λ (p: (pointer?):
                          ¬ void?(typ(p)) ∧ ¬ function?(typ(p))))),
      ptr_val1: (range_pointer(cv_base(typ))),
      ptr_val2: (range_pointer(cv_base(typ))):
      (array?(typ(cv_base(typ))) ⊃ size_of(typ(cv_base(typ))) > 0) ⊃
      cv(pointer?)(typ)
```

Repeatedly Skolemizing and flattening,

Keeping (-2 1) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of minus_ptr_ptr_TCC2.

Q.E.D.

C.11.25 ArithmeticExpressions.minus_ptr_ptr_TCC3

Terse proof for minus_ptr_ptr_TCC3.

minus_ptr_ptr_TCC3:

```
{1}  ∇ (typ:
      Cpp_Subtype(cv(extend[Cpp_Type_, (pointer?), bool, FALSE]
                        (λ (p: (pointer?):
                          ¬ void?(typ(p)) ∧ ¬ function?(typ(p))))),
      ptr_val1: (range_pointer(cv_base(typ))),
      ptr_val2: (range_pointer(cv_base(typ))):
      (array?(typ(cv_base(typ))) ⊃ size_of(typ(cv_base(typ))) > 0) ∧
      not_null?(typ)(ptr_val1) ∧
      not_null?(typ)(ptr_val2) ∧
      same_array(typ)(ptr_val1, ptr_val2) ∧
      up?(address_of(typ)(ptr_val1)) ∧ up?(address_of(typ)(ptr_val2))
      ⊃ Cpp_Type?(typ(cv_base(typ)))
```

Repeatedly Skolemizing and flattening,

Using lemma minus_ptr_typ_is_Cpp_Type,

This completes the proof of minus_ptr_ptr_TCC3.

Q.E.D.

C.11.26 ArithmeticExpressions.minus_ptr_ptr_TCC4

Terse proof for minus_ptr_ptr_TCC4.

minus_ptr_ptr_TCC4:

{1}	\forall (typ: Cpp_Subtype(cv(extend[Cpp_Type_, (pointer?), bool, FALSE] (λ (p: (pointer?): \neg void?(typ(p)) \wedge \neg function?(typ(p))))), ptr_val1: (range_pointer(cv_base(typ))), ptr_val2: (range_pointer(cv_base(typ)))): (array?(typ(cv_base(typ))) \supset size_of(typ(cv_base(typ))) > 0) \wedge not_null?(typ)(ptr_val1) \wedge not_null?(typ)(ptr_val2) \wedge same_array(typ)(ptr_val1, ptr_val2) \wedge up?(address_of(typ)(ptr_val1)) \wedge up?(address_of(typ)(ptr_val2)) \supset size_of(typ(cv_base(typ))) > 0
-----	---

Repeatedly Skolemizing and flattening,

Using lemma ptr_to_nonzero_object,

we get 2 subgoals:

minus_ptr_ptr_TCC4.1:

{-1}	(array?(typ(cv_base(cv_base(typ')))) \supset size_of(typ(cv_base(cv_base(typ')))) > 0) \supset size_of(typ(cv_base(cv_base(typ')))) > 0
{-2}	Cpp_Type?(typ')
{-3}	cv(extend[Cpp_Type_, (pointer?), bool, FALSE] (λ (p: (pointer?): \neg void?(typ(p)) \wedge \neg function?(typ(p)))) (typ')
{-4}	range_pointer(cv_base(typ'))(ptr_val1')
{-5}	range_pointer(cv_base(typ'))(ptr_val2')
{-6}	(array?(typ(cv_base(typ')))) \supset size_of(typ(cv_base(typ')))) > 0
{-7}	not_null?(typ')(ptr_val1')
{-8}	not_null?(typ')(ptr_val2')
{-9}	same_array(typ')(ptr_val1', ptr_val2')
{-10}	up?(address_of(typ')(ptr_val1'))
{-11}	up?(address_of(typ')(ptr_val2'))
{1}	size_of(typ(cv_base(typ')))) > 0

Expanding the definition of cv_base,

Using lemma cv_base_not_const,

Using lemma cv_base_not_volatile,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Expanding the definition of cv_base,

which is trivially true.

This completes the proof of minus_ptr_ptr_TCC4.1.

minus_ptr_ptr_TCC4.2:

{-1}	Cpp_Type?(typ')
{-2}	cv(extend[Cpp_Type_, (pointer?), bool, FALSE] ($\lambda (p: (pointer?)): \neg \text{void?}(\text{typ}(p)) \wedge \neg \text{function?}(\text{typ}(p))$)) (typ')
{-3}	range_pointer(cv_base(typ'))(ptr_val1')
{-4}	range_pointer(cv_base(typ'))(ptr_val2')
{-5}	(array?(typ(cv_base(typ')))) \supset size_of(typ(cv_base(typ'))) $>$ 0
{-6}	not_null?(typ')(ptr_val1')
{-7}	not_null?(typ')(ptr_val2')
{-8}	same_array(typ')(ptr_val1', ptr_val2')
{-9}	up?(address_of(typ')(ptr_val1'))
{-10}	up?(address_of(typ')(ptr_val2'))
{1}	extend[Cpp_Type_, (pointer?), bool, FALSE] ($\lambda (p: (pointer?)): \neg \text{void?}(\text{typ}(p)) \wedge \neg \text{function?}(\text{typ}(p))$)) (cv_base(typ'))
{2}	size_of(typ(cv_base(typ'))) $>$ 0

Keeping (-2 1) and hiding *,

Using lemma cv_base_result,

Keeping (1) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of minus_ptr_ptr_TCC4.2.

Q.E.D.

C.11.27 ArithmeticExpressions.minus_ptr_ptr_TCC5

Terse proof for minus_ptr_ptr_TCC5.

minus_ptr_ptr_TCC5:

{1}	\forall (typ: Cpp_Subtype(cv(extend[Cpp_Type_, (pointer?), bool, FALSE] ($\lambda (p: (pointer?)): \neg \text{void?}(\text{typ}(p)) \wedge \neg \text{function?}(\text{typ}(p))$)), ptr_val1: (range_pointer(cv_base(typ))), ptr_val2: (range_pointer(cv_base(typ))): (array?(typ(cv_base(typ))) \supset size_of(typ(cv_base(typ))) $>$ 0) \wedge not_null?(typ)(ptr_val1) \wedge not_null?(typ)(ptr_val2) \wedge same_array(typ)(ptr_val1, ptr_val2) \wedge up?(address_of(typ)(ptr_val1)) \wedge up?(address_of(typ)(ptr_val2)) \supset (\forall (difference: {q: int offset(down [Memory_Address](address_of(typ)(ptr_val1))) - offset(down = rem(size_of(typ(cv_base(typ))))(offset(down [Memory_Address](address difference = ndiv(offset(down(address_of(typ)(ptr_val1))) - offset(down(address_of(typ)(ptr_val2))), size_of(typ(cv_base(typ)))) \supset Cpp_Type?(int) \wedge cv(non_bool_integral_enum?)(int))
-----	--

Repeatedly Skolemizing and flattening,
 Keeping (1) and hiding *,
 Applying propositional simplification,
 we get 2 subgoals:
 minus_ptr_ptr_TCC5.1:

{1} Cpp_Type?(int)

Expanding the definition of Cpp_Type?,
 Repeatedly Skolemizing and flattening,
 Expanding the definition of subterm,
 Replacing using formula -1,

Installing automatic rewrites from: no_pointers_to_bitfield? no_pointers_to_references?
 no_cv_references? no_reference_to_reference? no_reference_to_bitfields? no_pointer_to_member_to_reference?
 no_pointer_to_member_to_cv_void? no_cv_void_parameter? no_array_of_references? no_array_of_bitfields?
 no_array_of_cv_void? no_array_of_function? no_array_of_abstract_class? no_pointer_or_ref_to_incomplete_array_parameter?
 no_array_return_type? no_function_return_type? bitfield_underlying_integral_or_enum_type? cv_array?
 enum_underlying_integral? enum_constants? const_volatile? const_stutter? volatile_stutter?
 no_cv_class? no_cv_union? no_cv_function? no_reference_to_void? no_cv_void?

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 This completes the proof of minus_ptr_ptr_TCC5.1.

minus_ptr_ptr_TCC5.2:

{1} cv(non_bool_integral_enum?)(int)

Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of minus_ptr_ptr_TCC5.2.
 Q.E.D.

C.11.28 ArithmeticExpressions.minus_ptr_ptr_TCC6

Terse proof for minus_ptr_ptr_TCC6.

minus_ptr_ptr_TCC6:

{1} \forall (typ:
 Cpp_Subtype(cv(extend[Cpp_Type_, (pointer?), bool, FALSE]
 (λ (p: (pointer?):
 \neg void?(typ(p)) \wedge \neg function?(typ(p))))),
 ptr_val1: (range_pointer(cv_base(typ))),
 ptr_val2: (range_pointer(cv_base(typ))):
 \neg ((array?(typ(cv_base(typ))) \supset size_of(typ(cv_base(typ))) > 0) \wedge
 not_null?(typ)(ptr_val1) \wedge
 not_null?(typ)(ptr_val2) \wedge
 same_array(typ)(ptr_val1, ptr_val2) \wedge
 up?(address_of(typ)(ptr_val1)) \wedge up?(address_of(typ)(ptr_val2)))
 \supset Cpp_Type?(int) \wedge cv(non_bool_integral_enum?)(int)

Repeatedly Skolemizing and flattening,
 Keeping (2) and hiding *,
 Applying propositional simplification,
 we get 2 subgoals:

minus_ptr_ptr_TCC6.1:

{1} Cpp_Type?(int)

Expanding the definition of Cpp_Type?,
 Repeatedly Skolemizing and flattening,
 Expanding the definition of subterm,
 Replacing using formula -1,

Installing automatic rewrites from: no_pointers_to_bitfield? no_pointers_to_references?
 no_cv_references? no_reference_to_reference? no_reference_to_bitfields? no_pointer_to_member_to_reference?
 no_pointer_to_member_to_cv_void? no_cv_void_parameter? no_array_of_references? no_array_of_bitfields?
 no_array_of_cv_void? no_array_of_function? no_array_of_abstract_class? no_pointer_or_ref_to_incomplete_array_par
 no_array_return_type? no_function_return_type? bitfield_underlying_integral_or_enum_type? cv_array?
 enum_underlying_integral? enum_constants? const_volatile? const_stutter? volatile_stutter?
 no_cv_class? no_cv_union? no_cv_function? no_reference_to_void? no_cv_void?

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 This completes the proof of minus_ptr_ptr_TCC6.1.

minus_ptr_ptr_TCC6.2:

{1} cv(non_bool_integral_enum?)(int)

Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of minus_ptr_ptr_TCC6.2.
 Q.E.D.

C.11.29 ArithmeticExpressions.minus_ptr_ptr_TCC7

Terse proof for minus_ptr_ptr_TCC7.

minus_ptr_ptr_TCC7:

{1} \forall (typ:
 Cpp_Subtype(cv(extend[Cpp_Type_, (pointer?), bool, FALSE]
 (λ (p: (pointer?):
 \neg void?(typ(p)) \wedge \neg function?(typ(p))))),
 ptr_val1: (range_pointer(cv_base(typ))))):
 pointer?(cv_base(typ))

Repeatedly Skolemizing and flattening,
 Rewriting using cv_base_result, matching in *,
 we get 2 subgoals:

minus_ptr_ptr_TCC7.1:

{-1} Cpp_Type?(typ')
 {-2} cv(extend[Cpp_Type_, (pointer?), bool, FALSE]
 (λ (p: (pointer?): \neg void?(typ(p)) \wedge \neg function?(typ(p))))
 (typ')
 {-3} range_pointer(cv_base(typ'))(ptr_val1')

 {1} cv(pointer?)(typ')
 {2} pointer?(cv_base(typ'))

Keeping (-2 1) and hiding *,
 Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of minus_ptr_ptr_TCC7.1.

minus_ptr_ptr_TCC7.2:

<div style="display: flex; flex-direction: column; gap: 5px;"> <div style="display: flex; align-items: center;"> {-1} Cpp_Type?(typ') </div> <div style="display: flex; align-items: center;"> {-2} cv(extend[Cpp_Type_, (pointer?), bool, FALSE] (λ (p: (pointer?)): ¬ void?(typ(p)) ∧ ¬ function?(typ(p)))) (typ')) </div> <div style="display: flex; align-items: center;"> {-3} range_pointer(cv_base(typ'))(ptr_val1') </div> </div>	<div style="display: flex; flex-direction: column; gap: 5px;"> <div style="display: flex; align-items: center;"> {1} (pointer? ⊆ interpreted?) </div> <div style="display: flex; align-items: center;"> {2} pointer?(cv_base(typ')) </div> </div>
---	---

Keeping (1) and hiding *,
 Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of minus_ptr_ptr_TCC7.2.
 Q.E.D.

C.11.30 ArithmeticExpressions.minus_ptr_ptr_TCC8

Terse proof for minus_ptr_ptr_TCC8.

minus_ptr_ptr_TCC8:

<div style="display: flex; flex-direction: column; gap: 5px;"> <div style="display: flex; align-items: center;"> {1} ∀ (typ: Cpp_Subtype(cv(extend[Cpp_Type_, (pointer?), bool, FALSE] (λ (p: (pointer?): ¬ void?(typ(p)) ∧ ¬ function?(typ(p))))), ptr_val1: (range_pointer(cv_base(typ))))): Cpp_Type?(int) ∧ cv(non_bool_integral_enum?(int)) </div> </div>

Repeatedly Skolemizing and flattening,
 Keeping (1) and hiding *,
 Applying propositional simplification,
 we get 2 subgoals:
 minus_ptr_ptr_TCC8.1:

<div style="display: flex; flex-direction: column; gap: 5px;"> <div style="display: flex; align-items: center;"> {1} Cpp_Type?(int) </div> </div>

Expanding the definition of Cpp_Type?,
 Repeatedly Skolemizing and flattening,
 Expanding the definition of subterm,
 Replacing using formula -1,
 Installing automatic rewrites from: no_pointers_to_bitfield? no_pointers_to_references?
 no_cv_references? no_reference_to_reference? no_reference_to_bitfields? no_pointer_to_member_to_reference?
 no_pointer_to_member_to_cv_void? no_cv_void_parameter? no_array_of_references? no_array_of_bitfields?
 no_array_of_cv_void? no_array_of_function? no_array_of_abstract_class? no_pointer_or_ref_to_incomplete_array_parameter?
 no_array_return_type? no_function_return_type? bitfield_underlying_integral_or_enum_type? cv_array?
 enum_underlying_integral? enum_constants? const_volatile? const_stutter? volatile_stutter?
 no_cv_class? no_cv_union? no_cv_function? no_reference_to_void? no_cv_void?

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 This completes the proof of minus_ptr_ptr_TCC8.1.

minus_ptr_ptr_TCC8.2:

<div style="display: flex; flex-direction: column; gap: 5px;"> <div style="display: flex; align-items: center;"> {1} cv(non_bool_integral_enum?(int) </div> </div>
--

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `minus_ptr_ptr_TCC8.2`.
Q.E.D.

C.11.31 ArithmeticExpressions.minus_ptr_ptr_TCC9

Terse proof for `minus_ptr_ptr_TCC9`.

`minus_ptr_ptr_TCC9`:

$\{1\} \quad \forall (\text{typ}: \text{Cpp_Subtype}(\text{cv}(\text{extend}[\text{Cpp_Type_}, (\text{pointer?}), \text{bool}, \text{FALSE}] (\lambda (p: (\text{pointer?})): \neg \text{void?}(\text{typ}(p)) \wedge \neg \text{function?}(\text{typ}(p))))): \text{pointer?}(\text{cv_base}(\text{typ})))$
--

Repeatedly Skolemizing and flattening,
Rewriting using `cv_base_result`, matching in `*`,
we get 2 subgoals:

`minus_ptr_ptr_TCC9.1`:

$\{-1\} \quad \text{Cpp_Type?}(\text{typ}')$ $\{-2\} \quad \text{cv}(\text{extend}[\text{Cpp_Type_}, (\text{pointer?}), \text{bool}, \text{FALSE}] (\lambda (p: (\text{pointer?})): \neg \text{void?}(\text{typ}(p)) \wedge \neg \text{function?}(\text{typ}(p)))) (\text{typ}')$
$\{1\} \quad \text{cv}(\text{pointer?})(\text{typ}')$ $\{2\} \quad \text{pointer?}(\text{cv_base}(\text{typ}'))$

Keeping `(-2 1)` and hiding `*`,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `minus_ptr_ptr_TCC9.1`.
`minus_ptr_ptr_TCC9.2`:

$\{-1\} \quad \text{Cpp_Type?}(\text{typ}')$ $\{-2\} \quad \text{cv}(\text{extend}[\text{Cpp_Type_}, (\text{pointer?}), \text{bool}, \text{FALSE}] (\lambda (p: (\text{pointer?})): \neg \text{void?}(\text{typ}(p)) \wedge \neg \text{function?}(\text{typ}(p)))) (\text{typ}')$
$\{1\} \quad (\text{pointer?} \subseteq \text{interpreted?})$ $\{2\} \quad \text{pointer?}(\text{cv_base}(\text{typ}'))$

Keeping `(1)` and hiding `*`,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `minus_ptr_ptr_TCC9.2`.
Q.E.D.

C.11.32 ArithmeticExpressions.minus_ptr_ptr_TCC10

Terse proof for `minus_ptr_ptr_TCC10`.

`minus_ptr_ptr_TCC10`:

$\{1\} \quad \forall (\text{typ}: \text{Cpp_Subtype}(\text{cv}(\text{extend}[\text{Cpp_Type_}, (\text{pointer?}), \text{bool}, \text{FALSE}] (\lambda (p: (\text{pointer?})): \neg \text{void?}(\text{typ}(p)) \wedge \neg \text{function?}(\text{typ}(p))))): \text{Cpp_Type?}(\text{int}) \wedge \text{cv}(\text{non_bool_Integral_enum?})(\text{int}))$

Repeatedly Skolemizing and flattening,
 Keeping (1) and hiding *,
 Applying propositional simplification,
 we get 2 subgoals:
 minus_ptr_ptr_TCC10.1:

{1} Cpp_Type?(int)

Expanding the definition of Cpp_Type?,
 Expanding the definition of subterm,
 Repeatedly Skolemizing and flattening,
 Replacing using formula -1,

Installing automatic rewrites from: no_pointers_to_bitfield? no_pointers_to_references?
 no_cv_references? no_reference_to_reference? no_reference_to_bitfields? no_pointer_to_member_to_reference?
 no_pointer_to_member_to_cv_void? no_cv_void_parameter? no_array_of_references? no_array_of_bitfields?
 no_array_of_cv_void? no_array_of_function? no_array_of_abstract_class? no_pointer_or_ref_to_incomplete_array_parameter?
 no_array_return_type? no_function_return_type? bitfield_underlying_integral_or_enum_type? cv_array?
 enum_underlying_integral? enum_constants? const_volatile? const_stutter? volatile_stutter?
 no_cv_class? no_cv_union? no_cv_function? no_reference_to_void? no_cv_void?

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 This completes the proof of minus_ptr_ptr_TCC10.1.

minus_ptr_ptr_TCC10.2:

{1} cv(non_bool_integral_enum?)(int)

Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of minus_ptr_ptr_TCC10.2.
 Q.E.D.

C.11.33 ArithmeticExpressions.cmp_pointer_TCC1

Terse proof for cmp_pointer_TCC1.

cmp_pointer_TCC1:

{1} $\forall (p_typ: \text{Cpp_Subtype}(cv(\text{pointer?})), ptr_val1: (\text{range_pointer}(cv_base(p_typ)))):$
 pointer?(cv_base(p_typ))

Repeatedly Skolemizing and flattening,
 Rewriting using cv_base_result, matching in *,
 Keeping (1) and hiding *,
 Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of cmp_pointer_TCC1.
 Q.E.D.

C.11.34 ArithmeticExpressions.cmp_pointer_TCC2

Terse proof for cmp_pointer_TCC2.

cmp_pointer_TCC2:

{1} $\forall (p_typ: \text{Cpp_Subtype}(cv(\text{pointer?}))):$ pointer?(cv_base(p_typ))

Repeatedly Skolemizing and flattening,

Rewriting using `cv_base_result`, matching in `*`,
 Keeping (1) and hiding `*`,
 Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of `cmp_pointer_TCC2`.
 Q.E.D.

C.12 Proofs for `AssemblerStatements (statements.pvs)`

This theory contains no provable formal statements.

C.13 Proofs for `AssignmentExpressions (expressions.pvs)`

C.13.1 `AssignmentExpressions.assign_times_TCC1`

Terse proof for `assign_times_TCC1`.

`assign_times_TCC1`:

$$\frac{}{\{1\} \quad \forall (i_typ: \text{Cpp_Subtype}(v(\text{non_bool_integral_enum?}))) : \text{cv}(\text{non_bool_integral_enum?})(i_typ)}$$

Repeatedly Skolemizing and flattening,
 Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of `assign_times_TCC1`.
 Q.E.D.

C.13.2 `AssignmentExpressions.assign_plus_ptr_TCC1`

Terse proof for `assign_plus_ptr_TCC1`.

`assign_plus_ptr_TCC1`:

$$\frac{}{\{1\} \quad \forall (p_typ: \text{Cpp_Subtype}(v(\text{pointer?})), ci_typ: \text{Cpp_Subtype}(\text{cv}(\text{non_bool_integral_enum?}))) : \text{cv}(\text{pointer?})(p_typ)}$$

Repeatedly Skolemizing and flattening,
 Keeping (-2 1) and hiding `*`,
 Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of `assign_plus_ptr_TCC1`.
 Q.E.D.

C.13.3 `AssignmentExpressions.assign_plus_ptr_TCC2`

Terse proof for `assign_plus_ptr_TCC2`.

`assign_plus_ptr_TCC2`:

$$\frac{}{\{1\} \quad \forall (p_typ: \text{Cpp_Subtype}(v(\text{pointer?})), ci_typ: \text{Cpp_Subtype}(\text{cv}(\text{non_bool_integral_enum?}))) : \text{pointer?}(\text{cv_base}(p_typ))}$$

Repeatedly Skolemizing and flattening,
 Rewriting using `cv_base_result`, matching in `*`,

we get 2 subgoals:

`assign_plus_ptr_TCC2.1:`

{-1}	Cpp_Type?(p_typ')
{-2}	v(pointer?)(p_typ')
{-3}	Cpp_Type?(ci_typ')
{-4}	cv(non_bool_integral_enum?)(ci_typ')
{1}	cv(pointer?)(p_typ')
{2}	pointer?(cv_base(p_typ'))

Using lemma `assign_plus_ptr_TCC1`,

This completes the proof of `assign_plus_ptr_TCC2.1`.

`assign_plus_ptr_TCC2.2:`

{-1}	Cpp_Type?(p_typ')
{-2}	v(pointer?)(p_typ')
{-3}	Cpp_Type?(ci_typ')
{-4}	cv(non_bool_integral_enum?)(ci_typ')
{1}	(pointer? \subseteq interpreted?)
{2}	pointer?(cv_base(p_typ'))

Keeping (1) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `assign_plus_ptr_TCC2.2`.

Q.E.D.

C.14 Proofs for AssignmentExpressions2 (expressions.pvs)

This theory contains no provable formal statements.

C.15 Proofs for Bitlist_ops (bits.pvs)

C.15.1 Bitlist_ops.binary_not_TCC1

Terse proof for `binary_not_TCC1`.

`binary_not_TCC1:`

{1}	$(\forall (x: \text{nat}): x < \text{expt}(2, \text{bits_per_byte}) \equiv x < \text{max_byte}) \wedge$ $(\forall (x_1: \text{below}[\text{expt}(2, \text{bits_per_byte})]): \text{binary_not}[\text{bits_per_byte}](x_1) < \text{max_byte})$
-----	---

Expanding the definition of `max_byte`,

Repeatedly Skolemizing and flattening,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `binary_not_TCC1`.

Q.E.D.

C.15.2 Bitlist_ops.binary_and_TCC1

Terse proof for `binary_and_TCC1`.

binary_and_TCC1:

$$\{1\} \quad (\forall (x: \text{nat}): x < \text{expt}(2, \text{bits_per_byte}) \equiv x < \text{max_byte}) \wedge \\ (\forall (x_1: [\text{below}[\text{expt}(2, \text{bits_per_byte})], \text{below}[\text{expt}(2, \text{bits_per_byte})]]): \\ \text{binary_and}[\text{bits_per_byte}](x_1) < \text{max_byte})$$

Expanding the definition of max_byte,
 Repeatedly Skolemizing and flattening,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 This completes the proof of binary_and_TCC1.
 Q.E.D.

C.15.3 Bitlist_ops.binary_or_TCC1

Terse proof for binary_or_TCC1.

binary_or_TCC1:

$$\{1\} \quad (\forall (x: \text{nat}): x < \text{expt}(2, \text{bits_per_byte}) \equiv x < \text{max_byte}) \wedge \\ (\forall (x_1: [\text{below}[\text{expt}(2, \text{bits_per_byte})], \text{below}[\text{expt}(2, \text{bits_per_byte})]]): \\ \text{binary_or}[\text{bits_per_byte}](x_1) < \text{max_byte})$$

Expanding the definition of max_byte,
 Repeatedly Skolemizing and flattening,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 This completes the proof of binary_or_TCC1.
 Q.E.D.

C.15.4 Bitlist_ops.binary_xor_TCC1

Terse proof for binary_xor_TCC1.

binary_xor_TCC1:

$$\{1\} \quad (\forall (x: \text{nat}): x < \text{expt}(2, \text{bits_per_byte}) \equiv x < \text{max_byte}) \wedge \\ (\forall (x_1: [\text{below}[\text{expt}(2, \text{bits_per_byte})], \text{below}[\text{expt}(2, \text{bits_per_byte})]]): \\ \text{binary_xor}[\text{bits_per_byte}](x_1) < \text{max_byte})$$

Expanding the definition of max_byte,
 Repeatedly Skolemizing and flattening,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 This completes the proof of binary_xor_TCC1.
 Q.E.D.

C.15.5 Bitlist_ops.singleton_zero_mask

Terse proof for singleton_zero_mask.

singleton_zero_mask:

$$\{1\} \quad \forall (s: \text{nat}): \text{singleton}?(zero_mask?(s))$$

Inducting on s on formula 1,
 we get 2 subgoals:

singleton_zero_mask.1:

{1} singleton?(zero_mask?(0))

Expanding the definition of zero_mask?,
 Expanding the definition of length,
 Expanding the definition of singleton?,
 Instantiating quantified variables,
 we get 3 subgoals:

singleton_zero_mask.1.1:

{1} $\forall (y_1:$
 $(\lambda (l: \text{list}[\text{Byte}]):$
 CASES l OF null: 0, cons(x , y): length(y) + 1 ENDCASES = 0 \wedge
 every($\lambda (t: \text{Byte}): t = 0$)(l))):

 null = y_1

Repeatedly Skolemizing and flattening,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 This completes the proof of singleton_zero_mask.1.1.

singleton_zero_mask.1.2:

{1} every[Byte]($\lambda (t: \text{Byte}): t = 0$)(null[Byte])

Expanding the definition of every,
 which is trivially true.
 This completes the proof of singleton_zero_mask.1.2.

singleton_zero_mask.1.3:

{1} every[Byte]($\lambda (t: \text{Byte}): t = 0$)(null[Byte])
 {2} $\exists (x_1:$
 $(\lambda (l: \text{list}[\text{Byte}]):$
 CASES l OF null: 0, cons(x , y): length(y) + 1 ENDCASES = 0 \wedge
 every($\lambda (t: \text{Byte}): t = 0$)(l))):

$\forall (y_1:$
 $(\lambda (l: \text{list}[\text{Byte}]):$
 CASES l OF null: 0, cons(x , y): length(y) + 1 ENDCASES = 0 \wedge
 every($\lambda (t: \text{Byte}): t = 0$)(l))):

$x_1 = y_1$

Expanding the definition of every,
 which is trivially true.
 This completes the proof of singleton_zero_mask.1.3.

singleton_zero_mask.2:

{1} $\forall j: \text{singleton?}(\text{zero_mask?}(j)) \supset \text{singleton?}(\text{zero_mask?}(j + 1))$

Repeatedly Skolemizing and flattening,
 Expanding the definition of zero_mask?,
 Expanding the definition of singleton?,
 Repeatedly Skolemizing and flattening,
 Instantiating the top quantifier in 1 with the terms: (cons(0, x!1)),

C Proof scripts

we get 3 subgoals:

`singleton_zero_mask.2.1:`

<pre> {-1} every(λ (x: number): number_field_pred(x) ∧ real_pred(x) ∧ rational_pred(x) ∧ integer_pred(x) ∧ x ≥ 0 ∧ x < max_byte) (x') {-2} length(x') = j' {-3} every(λ (t: Byte): t = 0)(x') {-4} j' ≥ 0 {-5} ∀ (y: (λ (l: list[Byte]): length(l) = j' ∧ every(λ (t: Byte): t = 0)(l))): x' = y </pre>	<pre> {1} ∀ (y: (λ (l: list[Byte]): length(l) = 1 + j' ∧ every(λ (t: Byte): t = 0)(l))): cons(0, x') = y </pre>
--	---

Repeatedly Skolemizing and flattening,

Case splitting on `null?(y!)`,

we get 2 subgoals:

`singleton_zero_mask.2.1.1:`

<pre> {-1} null?(y') {-2} every(λ (x: number): number_field_pred(x) ∧ real_pred(x) ∧ rational_pred(x) ∧ integer_pred(x) ∧ x ≥ 0 ∧ x < max_byte) (y') {-3} length(y') = 1 + j' {-4} every(λ (t: Byte): t = 0)(y') {-5} every(λ (x: number): number_field_pred(x) ∧ real_pred(x) ∧ rational_pred(x) ∧ integer_pred(x) ∧ x ≥ 0 ∧ x < max_byte) (x') {-6} length(x') = j' {-7} every(λ (t: Byte): t = 0)(x') {-8} j' ≥ 0 {-9} ∀ (y: (λ (l: list[Byte]): length(l) = j' ∧ every(λ (t: Byte): t = 0)(l))): x' = y </pre>	<pre> {1} cons(0, x') = y' </pre>
--	-----------------------------------

Expanding the definition of length,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `singleton_zero_mask.2.1.1`.

singleton_zero_mask.2.1.2:

{-1}	every(λ (x : number): number_field_pred(x) \wedge real_pred(x) \wedge rational_pred(x) \wedge integer_pred(x) \wedge $x \geq 0 \wedge x < \text{max_byte}$)
{-2}	(y') length(y') = $1 + j'$
{-3}	every(λ (t : Byte): $t = 0$)(y')
{-4}	every(λ (x : number): number_field_pred(x) \wedge real_pred(x) \wedge rational_pred(x) \wedge integer_pred(x) \wedge $x \geq 0 \wedge x < \text{max_byte}$)
{-5}	(x') length(x') = j'
{-6}	every(λ (t : Byte): $t = 0$)(x')
{-7}	$j' \geq 0$
{-8}	\forall (y : (λ (l : list[Byte]): length(l) = $j' \wedge$ every(λ (t : Byte): $t = 0$)(l))) $x' = y$
{1}	null?(y')
{2}	cons(0, x') = y'

Expanding the definition of length,
 Simplifying, rewriting, and recording with decision procedures,
 Instantiating the top quantifier in -8 with the terms: (cdr($y!$ 1)),
 we get 2 subgoals:

singleton_zero_mask.2.1.2.1:

{-1}	every(λ (x : number): number_field_pred(x) \wedge real_pred(x) \wedge rational_pred(x) \wedge integer_pred(x) \wedge $x \geq 0 \wedge x < \text{max_byte}$)
{-2}	(y') length(cdr(y')) = j'
{-3}	every(λ (t : Byte): $t = 0$)(y')
{-4}	every(λ (x : number): number_field_pred(x) \wedge real_pred(x) \wedge rational_pred(x) \wedge integer_pred(x) \wedge $x \geq 0 \wedge x < \text{max_byte}$)
{-5}	(x') CASES x' OF null: 0, cons(x , y): length(y) + 1 ENDCASES = j'
{-6}	every(λ (t : Byte): $t = 0$)(x')
{-7}	$j' \geq 0$
{-8}	$x' = \text{cdr}(y')$
{1}	null?(y')
{2}	cons(0, x') = y'

Expanding the definition of every,
 Replacing using formula -8,
 Using lemma list_cons_eta,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 This completes the proof of singleton_zero_mask.2.1.2.1.

singleton_zero_mask.2.1.2.2:

<p>{-1} every($\lambda (x: \text{number})$): number_field_pred(x) \wedge real_pred(x) \wedge rational_pred(x) \wedge integer_pred(x) $\wedge x \geq 0 \wedge x < \text{max_byte}$</p> <p style="text-align: center;">(y')</p> <p>{-2} length(cdr(y')) = j'</p> <p>{-3} every($\lambda (t: \text{Byte})$): $t = 0$)(y')</p> <p>{-4} every($\lambda (x: \text{number})$): number_field_pred(x) \wedge real_pred(x) \wedge rational_pred(x) \wedge integer_pred(x) $\wedge x \geq 0 \wedge x < \text{max_byte}$</p> <p style="text-align: center;">(x')</p> <p>{-5} CASES x' OF null: 0, cons(x, y): length(y) + 1 ENDCASES = j'</p> <p>{-6} every($\lambda (t: \text{Byte})$): $t = 0$)(x')</p> <p>{-7} $j' \geq 0$</p>	<hr style="border: 0.5px solid black;"/> <p>{1} every[Byte]($\lambda (t: \text{Byte})$): $t = 0$)(cdr[Byte](y'))</p> <p>{2} null?(y')</p> <p>{3} cons(0, x') = y'</p>
---	---

Expanding the definition of every,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of singleton_zero_mask.2.1.2.2.

singleton_zero_mask.2.2:

<p>{-1} every($\lambda (x: \text{number})$): number_field_pred(x) \wedge real_pred(x) \wedge rational_pred(x) \wedge integer_pred(x) $\wedge x \geq 0 \wedge x < \text{max_byte}$</p> <p style="text-align: center;">(x')</p> <p>{-2} length(x') = j'</p> <p>{-3} every($\lambda (t: \text{Byte})$): $t = 0$)(x')</p> <p>{-4} $j' \geq 0$</p> <p>{-5} $\forall (y$: ($\lambda (l: \text{list}[\text{Byte}])$): length(l) = $j' \wedge$ every($\lambda (t: \text{Byte})$): $t = 0$)(l)):</p> <p style="text-align: center;">$x' = y$</p>	<hr style="border: 0.5px solid black;"/> <p>{1} length[Byte](cons[Byte](0, x')) = $1 + j' \wedge$ every[Byte]($\lambda (t: \text{Byte})$): $t = 0$)(cons[Byte](0, x'))</p>
--	--

Expanding the definition of length,

Expanding the definition of every,

which is trivially true.

This completes the proof of singleton_zero_mask.2.2.

singleton_zero_mask.2.3:

{-1} every(λ (x : number): number_field_pred(x) \wedge real_pred(x) \wedge rational_pred(x) \wedge integer_pred(x) \wedge $x \geq 0 \wedge x < \text{max_byte}$)	
{-2} length(x') = j'	
{-3} every(λ (t : Byte): $t = 0$)(x')	
{-4} $j' \geq 0$	
{-5} \forall (y : (λ (l : list[Byte]): length(l) = $j' \wedge$ every(λ (t : Byte): $t = 0$)(l))): $x' = y$	
{1} $0 < \text{max_byte}$	

Keeping (1) and hiding *,
 Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of singleton_zero_mask.2.3.
 Q.E.D.

C.15.6 Bitlist_ops.bit_set?_TCC1

Terse proof for bit_set?_TCC1.

bit_set?_TCC1:

{1} \forall (n : nat, l : list[Byte]): $n < \text{length}(l) \times \text{max_byte} \supset \text{max_byte} > 0$	
---	--

Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of bit_set?_TCC1.
 Q.E.D.

C.15.7 Bitlist_ops.bit_set?_TCC2

Terse proof for bit_set?_TCC2.

bit_set?_TCC2:

{1} \forall (n : nat, l : list[Byte]): $n < \text{length}(l) \times \text{max_byte} \supset \text{ndiv}(n, \text{max_byte}) < \text{length}[\text{Byte}](l)$	
--	--

Repeatedly Skolemizing and flattening,
 Using lemma ndiv_lt,
 Using lemma both_sides_div_pos_lt1,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 This completes the proof of bit_set?_TCC2.
 Q.E.D.

C.16 Proofs for Bitops (bits.pvs)

C.16.1 Bitops.bit_set_zero_mask

Terse proof for bit_set_zero_mask.

bit_set_zero_mask:

$$\{1\} \quad \forall (n: \text{nat}): \neg \text{bit_set?}(n, 0)$$

Expanding the definition of bit_set?,
 Repeatedly Skolemizing and flattening,
 Rewriting using cut_bit_zero, matching in *,
 This completes the proof of bit_set_zero_mask.
 Q.E.D.

C.16.2 Bitops.bit_set_ndiv_expt2

Terse proof for bit_set_ndiv_expt2.

bit_set_ndiv_expt2:

$$\{1\} \quad \forall (n, \text{val}, m: \text{nat}): \text{bit_set?}(n, \text{ndiv}(\text{val}, \text{expt}(2, m))) = \text{bit_set?}(n + m, \text{val})$$

Repeatedly Skolemizing and flattening,
 Expanding the definition of bit_set?,
 Expanding the definition of cut_bit,
 Using lemma ndiv_times_2,
 Rewriting using expt_plus_aux, matching in *,
 Simplifying, rewriting, and recording with decision procedures,
 This completes the proof of bit_set_ndiv_expt2.
 Q.E.D.

C.16.3 Bitops.bit_set_full_mask_TCC1

Terse proof for bit_set_full_mask_TCC1.

bit_set_full_mask_TCC1:

$$\{1\} \quad \forall (n: \text{nat}): n < p \supset \text{expt}(2, p) - 1 \geq 0$$

Repeatedly Skolemizing and flattening,
 Using lemma expt_ge1,
 Expanding the definition of ^,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 This completes the proof of bit_set_full_mask_TCC1.
 Q.E.D.

C.16.4 Bitops.bit_set_full_mask

Terse proof for bit_set_full_mask.

bit_set_full_mask:

$$\{1\} \quad \forall (n: \text{nat}): n < p \supset \text{bit_set?}(n, \text{expt}(2, p) - 1)$$

Repeatedly Skolemizing and flattening,
 Expanding the definition of bit_set?,
 Expanding the definition of cut_bit,
 Using lemma ndiv_plus_1,

Case splitting on $\text{divides}(\text{expt}(2, n!), \text{expt}(2, p))$,

we get 2 subgoals:

bit_set_full_mask.1:

$$\begin{array}{l}
 \{-1\} \quad \text{divides}(\text{expt}(2, n'), \text{expt}(2, p)) \\
 \{-2\} \quad \text{divides}(\text{expt}(2, n'), \text{expt}(2, p)) \supset \\
 \quad \text{ndiv}(\text{expt}(2, p) - 1, \text{expt}(2, n')) = \\
 \quad \text{ndiv}(\text{expt}(2, p), \text{expt}(2, n')) + \text{ndiv}(-1, \text{expt}(2, n')) \\
 \{-3\} \quad n' \geq 0 \\
 \{-4\} \quad n' < p \\
 \hline
 \{1\} \quad \text{rem}(2)(\text{ndiv}(\text{expt}(2, p) - 1, \text{expt}(2, n'))) = 1
 \end{array}$$

Simplifying, rewriting, and recording with decision procedures,

Replacing using formula -2,

Rewriting using `ndiv_expt_expt`, matching in `*`,

Using lemma `rem_sum1`,

Replacing using formula -1,

Expanding the definition of `expt`,

Rewriting using `rem_multiple1`, matching in `*`,

Simplifying, rewriting, and recording with decision procedures,

Using lemma `rem_def`,

Using lemma `ndiv_plus_2`,

Rewriting using `divides_reflexive`, matching in `*`,

Rewriting using `ndiv_self`, matching in `*`,

Rewriting using `ndiv_bigger`, matching in `*`,

we get 2 subgoals:

bit_set_full_mask.1.1:

$$\begin{array}{l}
 \{-1\} \quad 0 = \text{ndiv}(-1, \text{expt}(2, n')) + 1 \\
 \{-2\} \quad \text{rem}(2)(\text{ndiv}(-1, \text{expt}(2, n'))) = 1 \equiv \\
 \quad (\exists q: \text{ndiv}(-1, \text{expt}(2, n')) = 2 \times q + 1) \\
 \{-3\} \quad \text{rem}(2)(\text{rem}(2)(\text{expt}(2, p - n')) + \text{ndiv}(-1, \text{expt}(2, n'))) = \\
 \quad \text{rem}(2)(\text{ndiv}(-1, \text{expt}(2, n')) + \text{expt}(2, p - n')) \\
 \{-4\} \quad \text{divides}(\text{expt}(2, n'), \text{expt}(2, p)) \\
 \{-5\} \quad \text{ndiv}(\text{expt}(2, p) - 1, \text{expt}(2, n')) = \\
 \quad \text{ndiv}(-1, \text{expt}(2, n')) + \text{expt}(2, p - n') \\
 \{-6\} \quad n' \geq 0 \\
 \{-7\} \quad n' < p \\
 \hline
 \{1\} \quad \text{rem}(2)(\text{ndiv}(-1, \text{expt}(2, n'))) = 1
 \end{array}$$

Using lemma `both_sides_plus1`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Replacing using formula -2,

Instantiating the top quantifier in 2 with the terms: (-1),

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `bit_set_full_mask.1.1`.

bit_set_full_mask.1.2:

{-1}	$\text{ndiv}(\text{expt}(2, n') - 1, \text{expt}(2, n')) = 1 + \text{ndiv}(-1, \text{expt}(2, n'))$
{-2}	$\text{rem}(2)(\text{rem}(2)(\text{expt}(2, p - n') + \text{ndiv}(-1, \text{expt}(2, n')))) = \text{rem}(2)(\text{ndiv}(-1, \text{expt}(2, n')) + \text{expt}(2, p - n'))$
{-3}	$\text{divides}(\text{expt}(2, n'), \text{expt}(2, p))$
{-4}	$\text{ndiv}(\text{expt}(2, p) - 1, \text{expt}(2, n')) = \text{ndiv}(-1, \text{expt}(2, n')) + \text{expt}(2, p - n')$
{-5}	$n' \geq 0$
{-6}	$n' < p$
{1}	$\text{expt}(2, n') - 1 \geq 0$
{2}	$\exists q: \text{ndiv}(-1, \text{expt}(2, n')) = 1 + 2 \times q$
{3}	$\text{rem}(2)(\text{ndiv}(-1, \text{expt}(2, n'))) = 1$

Using lemma `expt_ge1`,

Expanding the definition of \wedge ,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `bit_set_full_mask.1.2`.

bit_set_full_mask.2:

{-1}	$\text{divides}(\text{expt}(2, n'), \text{expt}(2, p)) \supset \text{ndiv}(\text{expt}(2, p) + -1, \text{expt}(2, n')) = \text{ndiv}(\text{expt}(2, p), \text{expt}(2, n')) + \text{ndiv}(-1, \text{expt}(2, n'))$
{-2}	$n' \geq 0$
{-3}	$n' < p$
{1}	$\text{divides}(\text{expt}(2, n'), \text{expt}(2, p))$
{2}	$\text{rem}(2)(\text{ndiv}(\text{expt}(2, p) - 1, \text{expt}(2, n'))) = 1$

Hiding formulas: (-1 2),

Expanding the definition of `divides`,

Instantiating the top quantifier in 1 with the terms: $(\text{expt}(2, p - n!))$,

we get 2 subgoals:

bit_set_full_mask.2.1:

{-1}	$n' \geq 0$
{-2}	$n' < p$
{1}	$\text{expt}(2, p) = \text{expt}(2, n') \times \text{expt}(2, p - n')$

Using lemma `expt_plus_aux`,

we get 2 subgoals:

bit_set_full_mask.2.1.1:

{-1}	$\text{expt}(2, n' + p - n') = \text{expt}(2, n') \times \text{expt}(2, p - n')$
{-2}	$n' \geq 0$
{-3}	$n' < p$
{1}	$\text{expt}(2, p) = \text{expt}(2, n') \times \text{expt}(2, p - n')$

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `bit_set_full_mask.2.1.1`.

bit_set_full_mask.2.1.2:

{-1}	$n' \geq 0$
{-2}	$n' < p$
{1}	$p - n' \geq 0$
{2}	$\text{expt}(2, p) = \text{expt}(2, n') \times \text{expt}(2, p - n')$

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `bit_set_full_mask.2.1.2`.

`bit_set_full_mask.2.2`:

{-1}	$n' \geq 0$
{-2}	$n' < p$
{1}	$p - n' \geq 0$

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `bit_set_full_mask.2.2`.

Q.E.D.

C.16.5 Bitops.bits_equal

Terse proof for `bits_equal`.

`bits_equal`:

{1}	$\forall (v_1, v_2, p: \text{nat}):$ $(\forall (n: \text{nat}): n < p \supset \text{bit_set?}(n, v_1) = \text{bit_set?}(n, v_2)) \supset$ $\text{rem}(\text{expt}(2, p))(v_1) = \text{rem}(\text{expt}(2, p))(v_2)$
-----	---

Inducting on p on formula 1,

we get 2 subgoals:

`bits_equal.1`:

{1}	$\forall (v_1, v_2: \text{nat}):$ $(\forall (n: \text{nat}): n < 0 \supset \text{bit_set?}(n, v_1) = \text{bit_set?}(n, v_2)) \supset$ $\text{rem}(\text{expt}(2, 0))(v_1) = \text{rem}(\text{expt}(2, 0))(v_2)$
-----	--

Repeatedly Skolemizing and flattening,

Expanding the definition of `expt`,

Rewriting using `rem_def`, matching in `*`,

Instantiating the top quantifier in 1 with the terms: $(v1!1 - \text{rem}(1)(v2!1))$,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `bits_equal.1`.

`bits_equal.2`:

{1}	$\forall j:$ $(\forall (v_1, v_2: \text{nat}):$ $(\forall (n: \text{nat}): n < j \supset \text{bit_set?}(n, v_1) = \text{bit_set?}(n, v_2)) \supset$ $\text{rem}(\text{expt}(2, j))(v_1) = \text{rem}(\text{expt}(2, j))(v_2))$ \supset $(\forall (v_1, v_2: \text{nat}):$ $(\forall (n: \text{nat}): n < j + 1 \supset \text{bit_set?}(n, v_1) = \text{bit_set?}(n, v_2)) \supset$ $\text{rem}(\text{expt}(2, j + 1))(v_1) = \text{rem}(\text{expt}(2, j + 1))(v_2))$
-----	---

Repeatedly Skolemizing and flattening,

Instantiating the top quantifier in -4 with the terms: $(v1!1 v2!1)$,

Splitting conjunctions,

we get 2 subgoals:

`bits_equal.2.1:`

{-1}	$\text{rem}(\text{expt}(2, j'))(v'_1) = \text{rem}(\text{expt}(2, j'))(v'_2)$
{-2}	$v'_1 \geq 0$
{-3}	$v'_2 \geq 0$
{-4}	$j' \geq 0$
{-5}	$\forall (n: \text{nat}): n < j' + 1 \supset \text{bit_set?}(n, v'_1) = \text{bit_set?}(n, v'_2)$
{1}	$\text{rem}(\text{expt}(2, j' + 1))(v'_1) = \text{rem}(\text{expt}(2, j' + 1))(v'_2)$

Expanding the definition of `expt`,
 Expanding the definition of `bit_set?`,
 Expanding the definition of `cut_bit`,
 Using lemma `rem_prod_3`,
 Using lemma `rem_prod_3`,
 Replacing using formula -1,
 Replacing using formula -2,
 Replacing using formula -3,
 Instantiating the top quantifier in -7 with the terms: `(j!1)`,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 This completes the proof of `bits_equal.2.1`.

`bits_equal.2.2:`

{-1}	$v'_1 \geq 0$
{-2}	$v'_2 \geq 0$
{-3}	$j' \geq 0$
{-4}	$\forall (n: \text{nat}): n < j' + 1 \supset \text{bit_set?}(n, v'_1) = \text{bit_set?}(n, v'_2)$
{1}	$\forall (n: \text{nat}): n < j' \supset \text{bit_set?}(n, v'_1) = \text{bit_set?}(n, v'_2)$
{2}	$\text{rem}(\text{expt}(2, j' + 1))(v'_1) = \text{rem}(\text{expt}(2, j' + 1))(v'_2)$

Repeatedly Skolemizing and flattening,
 Instantiating the top quantifier in -6 with the terms: `(n!1)`,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 This completes the proof of `bits_equal.2.2`.
 Q.E.D.

C.16.6 Bitops.bits_equal2

Terse proof for `bits_equal2`.

`bits_equal2:`

{1}	$\forall (v_1, v_2: \text{below}[\text{expt}(2, p)]):$ $(\forall (n: \text{nat}): n < p \supset \text{bit_set?}(n, v_1) = \text{bit_set?}(n, v_2)) \equiv v_1 = v_2$
-----	---

Repeatedly Skolemizing and flattening,
 Applying propositional simplification,
 we get 2 subgoals:

`bits_equal2.1:`

{-1}	$\forall (n: \text{nat}): n < p \supset \text{bit_set?}(n, v'_1) = \text{bit_set?}(n, v'_2)$
{-2}	$v'_1 < \text{expt}(2, p)$
{-3}	$v'_2 < \text{expt}(2, p)$
{1}	$v'_1 = v'_2$

Using lemma `bits_equal`,
 Replacing using formula -2,

Rewriting using `rem_mod`, matching in `*`,

Rewriting using `rem_mod`, matching in `*`,

This completes the proof of `bits_equal2.1`.

`bits_equal2.2`:

$$\frac{\begin{array}{l} \{-1\} \quad v'_1 = v'_2 \\ \{-2\} \quad v'_1 < \text{expt}(2, p) \\ \{-3\} \quad v'_2 < \text{expt}(2, p) \end{array}}{\{1\} \quad \forall (n: \text{nat}): n < p \supset \text{bit_set?}(n, v'_1) = \text{bit_set?}(n, v'_2)}$$

Repeatedly Skolemizing and flattening,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `bits_equal2.2`.

Q.E.D.

C.16.7 Bitops.bits_equal3

Terse proof for `bits_equal3`.

`bits_equal3`:

$$\frac{}{\{1\} \quad \forall (v_1, v_2: \text{nat}): (\forall (n: \text{nat}): \text{bit_set?}(n, v_1) = \text{bit_set?}(n, v_2)) \supset v_1 = v_2}$$

Repeatedly Skolemizing and flattening,

Applying `bits_equal`

Case splitting on EXISTS (x: posnat): v1!1 < expt(2, x) AND v2!1 < expt(2, x),

we get 2 subgoals:

`bits_equal3.1`:

$$\frac{\begin{array}{l} \{-1\} \quad \exists (x: \text{posnat}): v'_1 < \text{expt}(2, x) \wedge v'_2 < \text{expt}(2, x) \\ \{-2\} \quad \forall (v_1, v_2, p: \text{nat}): \\ \quad (\forall (n: \text{nat}): n < p \supset \text{bit_set?}(n, v_1) = \text{bit_set?}(n, v_2)) \supset \\ \quad \text{rem}(\text{expt}(2, p))(v_1) = \text{rem}(\text{expt}(2, p))(v_2) \\ \{-3\} \quad v'_1 \geq 0 \\ \{-4\} \quad v'_2 \geq 0 \\ \{-5\} \quad \forall (n: \text{nat}): \text{bit_set?}(n, v'_1) = \text{bit_set?}(n, v'_2) \end{array}}{\{1\} \quad v'_1 = v'_2}$$

Instantiating the top quantifier in -2 with the terms: (v1!1 v2!1 choose({p: posnat | v1!1 < expt(2, p) AND v2!1 < expt(2, p)})),

we get 2 subgoals:

bits_equal3.1.1.1:

<p>{-1} $\exists (x: \text{posnat}): v'_1 < \text{expt}(2, x) \wedge v'_2 < \text{expt}(2, x)$</p> <p>{-2} $(\forall (n: \text{nat}):$ $n < \text{choose}(\{p: \text{posnat} \mid v'_1 < \text{expt}(2, p) \wedge v'_2 < \text{expt}(2, p)\}) \supset$ $\text{bit_set?}(n, v'_1) = \text{bit_set?}(n, v'_2))$</p> <p style="text-align: center;">\supset</p> <p>$\text{rem}(\text{expt}(2, \text{choose}(\{p: \text{posnat} \mid v'_1 < \text{expt}(2, p) \wedge v'_2 < \text{expt}(2, p)\})))$ (v'_1)</p> <p style="text-align: center;">$=$</p> <p>$\text{rem}(\text{expt}(2, \text{choose}(\{p: \text{posnat} \mid v'_1 < \text{expt}(2, p) \wedge v'_2 < \text{expt}(2, p)\})))$ (v'_2)</p> <p>{-3} $v'_1 \geq 0$</p> <p>{-4} $v'_2 \geq 0$</p> <p>{-5} $\forall (n: \text{nat}): \text{bit_set?}(n, v'_1) = \text{bit_set?}(n, v'_2)$</p>	<hr style="border: 0.5px solid black;"/> <p>{1} $v'_1 = v'_2$</p>
--	--

Splitting conjunctions,

we get 2 subgoals:

bits_equal3.1.1.1.1:

<p>{-1} $\text{rem}(\text{expt}(2, \text{choose}(\{p: \text{posnat} \mid v'_1 < \text{expt}(2, p) \wedge v'_2 < \text{expt}(2, p)\})))$ (v'_1)</p> <p style="text-align: center;">$=$</p> <p>$\text{rem}(\text{expt}(2, \text{choose}(\{p: \text{posnat} \mid v'_1 < \text{expt}(2, p) \wedge v'_2 < \text{expt}(2, p)\})))$ (v'_2)</p> <p>{-2} $\exists (x: \text{posnat}): v'_1 < \text{expt}(2, x) \wedge v'_2 < \text{expt}(2, x)$</p> <p>{-3} $v'_1 \geq 0$</p> <p>{-4} $v'_2 \geq 0$</p> <p>{-5} $\forall (n: \text{nat}): \text{bit_set?}(n, v'_1) = \text{bit_set?}(n, v'_2)$</p>	<hr style="border: 0.5px solid black;"/> <p>{1} $v'_1 = v'_2$</p>
--	--

Using lemma rem_mod,

we get 3 subgoals:

bits_equal3.1.1.1.1.1:

<p>{-1} $\text{rem}(\text{expt}(2, \text{choose}(\{p: \text{posnat} \mid v'_1 < \text{expt}(2, p) \wedge v'_2 < \text{expt}(2, p)\})))$ (v'_1)</p> <p style="text-align: center;">$= v'_1$</p> <p>{-2} $\text{rem}(\text{expt}(2, \text{choose}(\{p: \text{posnat} \mid v'_1 < \text{expt}(2, p) \wedge v'_2 < \text{expt}(2, p)\})))$ (v'_1)</p> <p style="text-align: center;">$=$</p> <p>$\text{rem}(\text{expt}(2, \text{choose}(\{p: \text{posnat} \mid v'_1 < \text{expt}(2, p) \wedge v'_2 < \text{expt}(2, p)\})))$ (v'_2)</p> <p>{-3} $\exists (x: \text{posnat}): v'_1 < \text{expt}(2, x) \wedge v'_2 < \text{expt}(2, x)$</p> <p>{-4} $v'_1 \geq 0$</p> <p>{-5} $v'_2 \geq 0$</p> <p>{-6} $\forall (n: \text{nat}): \text{bit_set?}(n, v'_1) = \text{bit_set?}(n, v'_2)$</p>	<hr style="border: 0.5px solid black;"/> <p>{1} $v'_1 = v'_2$</p>
---	--

Replacing using formula -1,

Hiding formulas: -1,

Using lemma rem_mod,

we get 3 subgoals:

bits_equal3.1.1.1.1.1:

$\{-1\} \quad \text{rem}(\text{expt}(2, \text{choose}(\{p: \text{posnat} \mid v'_1 < \text{expt}(2, p) \wedge v'_2 < \text{expt}(2, p)\})))$ $\quad \quad \quad (v'_2)$ $\quad \quad \quad = v'_2$	$\{-2\} \quad v'_1 =$ $\quad \quad \quad \text{rem}(\text{expt}(2, \text{choose}(\{p: \text{posnat} \mid v'_1 < \text{expt}(2, p) \wedge v'_2 < \text{expt}(2, p)\})))$ $\quad \quad \quad (v'_2)$
$\{-3\} \quad \exists (x: \text{posnat}): v'_1 < \text{expt}(2, x) \wedge v'_2 < \text{expt}(2, x)$	$\{-4\} \quad v'_1 \geq 0$
$\{-5\} \quad v'_2 \geq 0$	$\{-6\} \quad \forall (n: \text{nat}): \text{bit_set?}(n, v'_1) = \text{bit_set?}(n, v'_2)$
$\{1\} \quad v'_1 = v'_2$	

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of bits_equal3.1.1.1.1.1.

bits_equal3.1.1.1.1.2:

$\{-1\} \quad v'_1 =$ $\quad \quad \quad \text{rem}(\text{expt}(2, \text{choose}(\{p: \text{posnat} \mid v'_1 < \text{expt}(2, p) \wedge v'_2 < \text{expt}(2, p)\})))$ $\quad \quad \quad (v'_2)$	$\{-2\} \quad \exists (x: \text{posnat}): v'_1 < \text{expt}(2, x) \wedge v'_2 < \text{expt}(2, x)$
$\{-3\} \quad v'_1 \geq 0$	$\{-4\} \quad v'_2 \geq 0$
$\{-5\} \quad \forall (n: \text{nat}): \text{bit_set?}(n, v'_1) = \text{bit_set?}(n, v'_2)$	
$\{1\} \quad v'_2 <$ $\quad \quad \quad \text{expt}(2, \text{choose}[\text{posnat}](\{p: \text{posnat} \mid v'_1 < \text{expt}(2, p) \wedge v'_2 < \text{expt}(2, p)\}))$	$\{2\} \quad v'_1 = v'_2$

Expanding the definition of choose,

Using lemma epsilon_ax,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of bits_equal3.1.1.1.1.2.

bits_equal3.1.1.1.1.3:

$\{-1\} \quad v'_1 =$ $\quad \quad \quad \text{rem}(\text{expt}(2, \text{choose}(\{p: \text{posnat} \mid v'_1 < \text{expt}(2, p) \wedge v'_2 < \text{expt}(2, p)\})))$ $\quad \quad \quad (v'_2)$	$\{-2\} \quad \exists (x: \text{posnat}): v'_1 < \text{expt}(2, x) \wedge v'_2 < \text{expt}(2, x)$
$\{-3\} \quad v'_1 \geq 0$	$\{-4\} \quad v'_2 \geq 0$
$\{-5\} \quad \forall (n: \text{nat}): \text{bit_set?}(n, v'_1) = \text{bit_set?}(n, v'_2)$	
$\{1\} \quad \text{nonempty?}[\text{posnat}](\{p: \text{posnat} \mid v'_1 < \text{expt}(2, p) \wedge v'_2 < \text{expt}(2, p)\})$	$\{2\} \quad v'_1 = v'_2$

Expanding the definition of nonempty?,

Expanding the definition of empty?,

Rewriting using forall_not, matching in *,

Expanding the definition of member,

which is trivially true.

This completes the proof of bits_equal3.1.1.1.1.3.

bits_equal3.1.1.1.2:

<p>{-1} $\text{rem}(\text{expt}(2, \text{choose}(\{p: \text{posnat} \mid v'_1 < \text{expt}(2, p) \wedge v'_2 < \text{expt}(2, p)\}))$ (v'_1) $=$ $\text{rem}(\text{expt}(2, \text{choose}(\{p: \text{posnat} \mid v'_1 < \text{expt}(2, p) \wedge v'_2 < \text{expt}(2, p)\}))$ (v'_2)</p> <p>{-2} $\exists (x: \text{posnat}): v'_1 < \text{expt}(2, x) \wedge v'_2 < \text{expt}(2, x)$ {-3} $v'_1 \geq 0$ {-4} $v'_2 \geq 0$ {-5} $\forall (n: \text{nat}): \text{bit_set?}(n, v'_1) = \text{bit_set?}(n, v'_2)$</p>	<hr style="border: 0.5px solid black;"/> <p>{1} $v'_1 <$ $\text{expt}(2, \text{choose}[\text{posnat}](\{p: \text{posnat} \mid v'_1 < \text{expt}(2, p) \wedge v'_2 < \text{expt}(2, p)\}))$ {2} $v'_1 = v'_2$</p>
--	---

Expanding the definition of choose,

Using lemma epsilon_ax,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of **bits_equal3.1.1.1.2**.

bits_equal3.1.1.1.3:

<p>{-1} $\text{rem}(\text{expt}(2, \text{choose}(\{p: \text{posnat} \mid v'_1 < \text{expt}(2, p) \wedge v'_2 < \text{expt}(2, p)\}))$ (v'_1) $=$ $\text{rem}(\text{expt}(2, \text{choose}(\{p: \text{posnat} \mid v'_1 < \text{expt}(2, p) \wedge v'_2 < \text{expt}(2, p)\}))$ (v'_2)</p> <p>{-2} $\exists (x: \text{posnat}): v'_1 < \text{expt}(2, x) \wedge v'_2 < \text{expt}(2, x)$ {-3} $v'_1 \geq 0$ {-4} $v'_2 \geq 0$ {-5} $\forall (n: \text{nat}): \text{bit_set?}(n, v'_1) = \text{bit_set?}(n, v'_2)$</p>	<hr style="border: 0.5px solid black;"/> <p>{1} $\text{nonempty?}[\text{posnat}](\{p: \text{posnat} \mid v'_1 < \text{expt}(2, p) \wedge v'_2 < \text{expt}(2, p)\})$ {2} $v'_1 = v'_2$</p>
--	--

Expanding the definition of nonempty?,

Expanding the definition of empty?,

Rewriting using forall_not, matching in *,

Expanding the definition of member,

which is trivially true.

This completes the proof of **bits_equal3.1.1.1.3**.

bits_equal3.1.1.2:

<p>{-1} $\exists (x: \text{posnat}): v'_1 < \text{expt}(2, x) \wedge v'_2 < \text{expt}(2, x)$ {-2} $v'_1 \geq 0$ {-3} $v'_2 \geq 0$ {-4} $\forall (n: \text{nat}): \text{bit_set?}(n, v'_1) = \text{bit_set?}(n, v'_2)$</p>	<hr style="border: 0.5px solid black;"/> <p>{1} $\forall (n: \text{nat}):$ $n < \text{choose}(\{p: \text{posnat} \mid v'_1 < \text{expt}(2, p) \wedge v'_2 < \text{expt}(2, p)\}) \supset$ $\text{bit_set?}(n, v'_1) = \text{bit_set?}(n, v'_2)$ {2} $v'_1 = v'_2$</p>
---	--

Repeatedly Skolemizing and flattening,

Instantiating quantified variables,

This completes the proof of **bits_equal3.1.1.2**.

bits_equal3.1.2:

{-1}	$\exists (x: \text{posnat}): v'_1 < \text{expt}(2, x) \wedge v'_2 < \text{expt}(2, x)$
{-2}	$v'_1 \geq 0$
{-3}	$v'_2 \geq 0$
{-4}	$\forall (n: \text{nat}): \text{bit_set?}(n, v'_1) = \text{bit_set?}(n, v'_2)$
{1}	$\text{nonempty?}[\text{posnat}](\{p: \text{posnat} \mid v'_1 < \text{expt}(2, p) \wedge v'_2 < \text{expt}(2, p)\})$
{2}	$v'_1 = v'_2$

Expanding the definition of nonempty?,
 Expanding the definition of empty?,
 Expanding the definition of member,
 Rewriting using forall_not, matching in *,
 This completes the proof of bits_equal3.1.2.

bits_equal3.2:

{-1}	$\forall (v_1, v_2, p: \text{nat}):$ $(\forall (n: \text{nat}): n < p \supset \text{bit_set?}(n, v_1) = \text{bit_set?}(n, v_2)) \supset$ $\text{rem}(\text{expt}(2, p))(v_1) = \text{rem}(\text{expt}(2, p))(v_2)$
{-2}	$v'_1 \geq 0$
{-3}	$v'_2 \geq 0$
{-4}	$\forall (n: \text{nat}): \text{bit_set?}(n, v'_1) = \text{bit_set?}(n, v'_2)$
{1}	$\exists (x: \text{posnat}): v'_1 < \text{expt}(2, x) \wedge v'_2 < \text{expt}(2, x)$
{2}	$v'_1 = v'_2$

Using lemma large_expt,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 Copying formula number: -1
 Hiding formulas: (-3 -6),
 Case splitting on v1!1 /= 0 AND v2!1 /= 0,
 we get 2 subgoals:

bits_equal3.2.1:

{-1}	$v'_1 \neq 0 \wedge v'_2 \neq 0$
{-2}	$\forall \text{py}: \exists n: \text{py} < \text{expt}(2, n)$
{-3}	$\forall \text{py}: \exists n: \text{py} < \text{expt}(2, n)$
{-4}	$v'_1 \geq 0$
{-5}	$v'_2 \geq 0$
{1}	$\exists (x: \text{posnat}): v'_1 < \text{expt}(2, x) \wedge v'_2 < \text{expt}(2, x)$
{2}	$v'_1 = v'_2$

Applying disjunctive simplification to flatten sequent,
 Instantiating the top quantifier in -1 with the terms: (v1!1),
 we get 2 subgoals:

bits_equal3.2.1.1:

{-1}	$\exists n: v'_1 < \text{expt}(2, n)$
{-2}	$\forall \text{py}: \exists n: \text{py} < \text{expt}(2, n)$
{-3}	$v'_1 \geq 0$
{-4}	$v'_2 \geq 0$
{1}	$v'_1 = 0$
{2}	$v'_2 = 0$
{3}	$\exists (x: \text{posnat}): v'_1 < \text{expt}(2, x) \wedge v'_2 < \text{expt}(2, x)$
{4}	$v'_1 = v'_2$

Instantiating the top quantifier in -2 with the terms: (v2!1),

C Proof scripts

we get 2 subgoals:

bits_equal3.2.1.1.1:

{-1}	$\exists n: v'_1 < \text{expt}(2, n)$
{-2}	$\exists n: v'_2 < \text{expt}(2, n)$
{-3}	$v'_1 \geq 0$
{-4}	$v'_2 \geq 0$
{1}	$v'_1 = 0$
{2}	$v'_2 = 0$
{3}	$\exists (x: \text{posnat}): v'_1 < \text{expt}(2, x) \wedge v'_2 < \text{expt}(2, x)$
{4}	$v'_1 = v'_2$

Repeatedly Skolemizing and flattening,

Instantiating the top quantifier in 3 with the terms: $(n!1 + n!2 + 1)$,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 2 subgoals:

bits_equal3.2.1.1.1.1:

{-1}	$n'' \geq 0$
{-2}	$n' \geq 0$
{-3}	$v'_1 < \text{expt}(2, n')$
{-4}	$v'_2 < \text{expt}(2, n'')$
{-5}	$v'_1 \geq 0$
{-6}	$v'_2 \geq 0$
{1}	$v'_1 = 0$
{2}	$v'_2 = 0$
{3}	$v'_2 < \text{expt}(2, 1 + n' + n'')$
{4}	$v'_1 = v'_2$

Rewriting using `expt_plus_aux`, matching in `*`,

Using lemma `lt_times_lt_pos2`,

Using lemma `expt_ge1`,

Expanding the definition of `^`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `bits_equal3.2.1.1.1.1`.

bits_equal3.2.1.1.1.2:

{-1}	$n'' \geq 0$
{-2}	$n' \geq 0$
{-3}	$v'_1 < \text{expt}(2, n')$
{-4}	$v'_2 < \text{expt}(2, n'')$
{-5}	$v'_1 \geq 0$
{-6}	$v'_2 \geq 0$
{1}	$v'_1 = 0$
{2}	$v'_2 = 0$
{3}	$v'_1 < \text{expt}(2, 1 + n' + n'')$
{4}	$v'_1 = v'_2$

Rewriting using `expt_plus_aux`, matching in `*` where `m` gets `n!1`, `n` gets `1 + n!2`,

Using lemma `lt_times_lt_pos2`,

Using lemma `expt_ge1`,

Expanding the definition of `^`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `bits_equal3.2.1.1.1.2`.

bits_equal3.2.1.1.2:

{-1}	$\exists n: v'_1 < \text{expt}(2, n)$
{-2}	$v'_1 \geq 0$
{-3}	$v'_2 \geq 0$
{1}	$v'_2 > 0$
{2}	$v'_1 = 0$
{3}	$v'_2 = 0$
{4}	$\exists (x: \text{posnat}): v'_1 < \text{expt}(2, x) \wedge v'_2 < \text{expt}(2, x)$
{5}	$v'_1 = v'_2$

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of bits_equal3.2.1.1.2.

bits_equal3.2.1.2:

{-1}	$\forall py: \exists n: py < \text{expt}(2, n)$
{-2}	$v'_1 \geq 0$
{-3}	$v'_2 \geq 0$
{1}	$v'_1 > 0$
{2}	$v'_1 = 0$
{3}	$v'_2 = 0$
{4}	$\exists (x: \text{posnat}): v'_1 < \text{expt}(2, x) \wedge v'_2 < \text{expt}(2, x)$
{5}	$v'_1 = v'_2$

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of bits_equal3.2.1.2.

bits_equal3.2.2:

{-1}	$\forall py: \exists n: py < \text{expt}(2, n)$
{-2}	$\forall py: \exists n: py < \text{expt}(2, n)$
{-3}	$v'_1 \geq 0$
{-4}	$v'_2 \geq 0$
{1}	$v'_1 \neq 0 \wedge v'_2 \neq 0$
{2}	$\exists (x: \text{posnat}): v'_1 < \text{expt}(2, x) \wedge v'_2 < \text{expt}(2, x)$
{3}	$v'_1 = v'_2$

Expanding the definition of /=,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
we get 2 subgoals:

bits_equal3.2.2.1:

{-1}	$(v'_1 = 0)$
{-2}	$\forall py: \exists n: py < \text{expt}(2, n)$
{-3}	$v'_1 \geq 0$
{-4}	$v'_2 \geq 0$
{1}	$\exists (x: \text{posnat}): v'_1 < \text{expt}(2, x) \wedge v'_2 < \text{expt}(2, x)$
{2}	$v'_1 = v'_2$

Replacing using formula -1,
Instantiating the top quantifier in -2 with the terms: (v2!1),
Repeatedly Skolemizing and flattening,
Instantiating the top quantifier in 1 with the terms: (n!1 + 1),
Rewriting using expt_plus_aux, matching in *,
Using lemma lt_times_lt_pos2,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Expanding the definition of expt,

C Proof scripts

Expanding the definition of `expt`,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `bits_equal3.2.2.1`.

`bits_equal3.2.2.2`:

{-1}	$(v'_2 = 0)$
{-2}	$\forall py: \exists n: py < \text{expt}(2, n)$
{-3}	$v'_1 \geq 0$
{-4}	$v'_2 \geq 0$
{1}	$\exists (x: \text{posnat}): v'_1 < \text{expt}(2, x) \wedge v'_2 < \text{expt}(2, x)$
{2}	$v'_1 = v'_2$

Replacing using formula -1,
Instantiating the top quantifier in -2 with the terms: `(v!1)`,
Repeatedly Skolemizing and flattening,
Instantiating the top quantifier in 1 with the terms: `(n!1 + 1)`,
Rewriting using `expt_plus_aux`, matching in `*`,
Using lemma `lt_times_lt_pos2`,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Expanding the definition of `expt`,
Expanding the definition of `expt`,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `bits_equal3.2.2.2`.
Q.E.D.

C.16.8 Bitops.bits_equal4

Terse proof for `bits_equal4`.

`bits_equal4`:

{1}	$\forall (v_1, v_2: \text{nat}, m: \text{nat}):$ $(\forall (n: \text{nat}): n \geq m \supset \text{bit_set?}(n, v_1) = \text{bit_set?}(n, v_2)) \supset$ $\text{ndiv}(v_1, \text{expt}(2, m)) = \text{ndiv}(v_2, \text{expt}(2, m))$
-----	--

Repeatedly Skolemizing and flattening,
Using lemma `bits_equal3`,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Hiding formulas: 2,
Repeatedly Skolemizing and flattening,
Instantiating the top quantifier in -5 with the terms: `(n!1 + m!1)`,
Simplifying, rewriting, and recording with decision procedures,
Rewriting using `bit_set_ndiv_expt2`, matching in `*`,
Rewriting using `bit_set_ndiv_expt2`, matching in `*`,
This completes the proof of `bits_equal4`.
Q.E.D.

C.16.9 Bitops.clear_bit_TCC1

Terse proof for `clear_bit_TCC1`.

`clear_bit_TCC1`:

{1}	$\forall (n: \text{nat}, \text{val}: \text{nat}): \text{bit_set?}(n, \text{val}) \supset \text{val} - \text{expt}(2, n) \geq 0$
-----	--

Repeatedly Skolemizing and flattening,
 Expanding the definition of bit_set?,
 Expanding the definition of cut_bit,
 Using lemma rem_def_pos,
 Simplifying, rewriting, and recording with decision procedures,
 Adding type constraints for ndiv(val!1, expt(2, n!1)),
 Repeatedly Skolemizing and flattening,
 Replacing using formula -4,
 Replacing using formula -2,
 Simplifying, rewriting, and recording with decision procedures,
 Keeping (-3 1) and hiding *,
 Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of clear_bit_TCC1.
 Q.E.D.

C.16.10 Bitops.change_bit_single_bit_op

Terse proof for change_bit_single_bit_op.

change_bit_single_bit_op:

{1} $\forall (n, m, v: \text{nat}, c: [\text{nat} \rightarrow \text{bool}]):$
 $n \neq m \supset \text{bit_set?}(n, v) = \text{bit_set?}(n, \text{change_bit}(v, c)(m))$

Installing automatic rewrites from: change_bit set_bit clear_bit bit_set? cut_bit

Repeatedly Skolemizing and flattening,
 Expanding the definition of change_bit,
 Expanding the definition of clear_bit,
 Expanding the definition of set_bit,
 Simplifying, rewriting, and recording with decision procedures,
 Case splitting on $\text{rem}(2)(\text{ndiv}(v!1, \text{expt}(2, m!1))) = 1$,
 we get 2 subgoals:

change_bit_single_bit_op.1:

{-1} $\text{rem}(2)(\text{ndiv}(v', \text{expt}(2, m')))) = 1$
 {-2} $n' \geq 0$
 {-3} $m' \geq 0$
 {-4} $v' \geq 0$

{1} $n' = m'$
 {2} $\text{rem}(2)(\text{ndiv}(v', \text{expt}(2, n')))) = 1 =$
 $(\text{rem}(2)$
 $(\text{ndiv}(\text{IF } c'(m')$
 $\quad \text{THEN IF } \text{rem}(2)(\text{ndiv}(v', \text{expt}(2, m')))) = 1$
 $\quad \quad \text{THEN } v'$
 $\quad \quad \text{ELSE } \text{expt}(2, m') + v'$
 $\quad \quad \text{ENDIF}$
 $\quad \text{ELSE IF } \text{rem}(2)(\text{ndiv}(v', \text{expt}(2, m')))) = 1$
 $\quad \quad \text{THEN } v' - \text{expt}(2, m')$
 $\quad \quad \text{ELSE } v'$
 $\quad \quad \text{ENDIF}$
 $\quad \text{ENDIF},$
 $\text{expt}(2, n'))$
 $= 1)$

C Proof scripts

Case splitting on $c!1(m!1)$,

we get 2 subgoals:

`change_bit_single_bit_op.1.1.1:`

<pre> {-1} c'(m') {-2} rem(2)(ndiv(v', expt(2, m'))) = 1 {-3} n' ≥ 0 {-4} m' ≥ 0 {-5} v' ≥ 0 </pre>
<pre> {1} n' = m' {2} rem(2)(ndiv(v', expt(2, n'))) = 1 = (rem(2) (ndiv(IF c'(m') THEN IF rem(2)(ndiv(v', expt(2, m'))) = 1 THEN v' ELSE expt(2, m') + v' ENDIF ELSE IF rem(2)(ndiv(v', expt(2, m'))) = 1 THEN v' - expt(2, m') ELSE v' ENDIF ENDIF, expt(2, n'))) = 1) </pre>

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `change_bit_single_bit_op.1.1.1`.

`change_bit_single_bit_op.1.1.2:`

<pre> {-1} rem(2)(ndiv(v', expt(2, m'))) = 1 {-2} n' ≥ 0 {-3} m' ≥ 0 {-4} v' ≥ 0 </pre>
<pre> {1} c'(m') {2} n' = m' {3} rem(2)(ndiv(v', expt(2, n'))) = 1 = (rem(2) (ndiv(IF c'(m') THEN IF rem(2)(ndiv(v', expt(2, m'))) = 1 THEN v' ELSE expt(2, m') + v' ENDIF ELSE IF rem(2)(ndiv(v', expt(2, m'))) = 1 THEN v' - expt(2, m') ELSE v' ENDIF ENDIF, expt(2, n'))) = 1) </pre>

Simplifying, rewriting, and recording with decision procedures,

Case splitting on $v!1 - \text{expt}(2, m!1) \geq 0$,

we get 2 subgoals:

change_bit_single_bit_op.1.2.1:

{-1}	$v' - \text{expt}(2, m') \geq 0$
{-2}	$\text{rem}(2)(\text{ndiv}(v', \text{expt}(2, m'))) = 1$
{-3}	$n' \geq 0$
{-4}	$m' \geq 0$
{-5}	$v' \geq 0$
{1}	$c'(m')$
{2}	$n' = m'$
{3}	$\text{rem}(2)(\text{ndiv}(v', \text{expt}(2, n'))) = 1 =$ $(\text{rem}(2)(\text{ndiv}(v' - \text{expt}(2, m'), \text{expt}(2, n'))) = 1)$

Case splitting on $n! > m!$,

we get 2 subgoals:

change_bit_single_bit_op.1.2.1.1:

{-1}	$n' > m'$
{-2}	$v' - \text{expt}(2, m') \geq 0$
{-3}	$\text{rem}(2)(\text{ndiv}(v', \text{expt}(2, m'))) = 1$
{-4}	$n' \geq 0$
{-5}	$m' \geq 0$
{-6}	$v' \geq 0$
{1}	$c'(m')$
{2}	$n' = m'$
{3}	$\text{rem}(2)(\text{ndiv}(v', \text{expt}(2, n'))) = 1 =$ $(\text{rem}(2)(\text{ndiv}(v' - \text{expt}(2, m'), \text{expt}(2, n'))) = 1)$

Rewriting using `rem_def`, matching in `*`,

Repeatedly Skolemizing and flattening,

Using lemma `ndiv_times_2`,

we get 2 subgoals:

change_bit_single_bit_op.1.2.1.1.1:

{-1}	$\text{ndiv}(v', \text{expt}(2, m') \times 2 \times \text{expt}(2, n' - m' - 1)) =$ $\text{ndiv}(\text{ndiv}(v', \text{expt}(2, m')), 2 \times \text{expt}(2, n' - m' - 1))$
{-2}	<code>integer_pred</code> (q')
{-3}	$n' > m'$
{-4}	$v' - \text{expt}(2, m') \geq 0$
{-5}	$\text{ndiv}(v', \text{expt}(2, m')) = 1 + 2 \times q'$
{-6}	$n' \geq 0$
{-7}	$m' \geq 0$
{-8}	$v' \geq 0$
{1}	$c'(m')$
{2}	$n' = m'$
{3}	$\text{rem}(2)(\text{ndiv}(v', \text{expt}(2, n'))) = 1 =$ $(\text{rem}(2)(\text{ndiv}(v' - \text{expt}(2, m'), \text{expt}(2, n'))) = 1)$

Using lemma `ndiv_times_2`,

we get 2 subgoals:

change_bit_single_bit_op.1.2.1.1.1.1.1:

{-1}	$\text{ndiv}(v' - \text{expt}(2, m'), \text{expt}(2, m') \times 2 \times \text{expt}(2, n' - m' - 1))$
	$=$
	$\text{ndiv}(\text{ndiv}(v' - \text{expt}(2, m'), \text{expt}(2, m')),$
	$2 \times \text{expt}(2, n' - m' - 1))$
{-2}	$\text{ndiv}(v', \text{expt}(2, m') \times 2 \times \text{expt}(2, n' - m' - 1)) =$
	$\text{ndiv}(\text{ndiv}(v', \text{expt}(2, m')), 2 \times \text{expt}(2, n' - m' - 1))$
{-3}	$\text{integer_pred}(q')$
{-4}	$n' > m'$
{-5}	$v' - \text{expt}(2, m') \geq 0$
{-6}	$\text{ndiv}(v', \text{expt}(2, m')) = 1 + 2 \times q'$
{-7}	$n' \geq 0$
{-8}	$m' \geq 0$
{-9}	$v' \geq 0$
{1}	$c'(m')$
{2}	$n' = m'$
{3}	$\text{rem}(2)(\text{ndiv}(v', \text{expt}(2, n'))) = 1 =$
	$(\text{rem}(2)(\text{ndiv}(v' - \text{expt}(2, m'), \text{expt}(2, n'))) = 1)$

Case splitting on $\text{expt}(2, m!) * (2 * \text{expt}(2, n! - m! - 1)) = \text{expt}(2, n!)$,
we get 3 subgoals:

change_bit_single_bit_op.1.2.1.1.1.1.1.1:

{-1}	$\text{expt}(2, m') \times (2 \times \text{expt}(2, n' - m' - 1)) = \text{expt}(2, n')$
{-2}	$\text{ndiv}(v' - \text{expt}(2, m'), \text{expt}(2, m') \times 2 \times \text{expt}(2, n' - m' - 1))$
	$=$
	$\text{ndiv}(\text{ndiv}(v' - \text{expt}(2, m'), \text{expt}(2, m')),$
	$2 \times \text{expt}(2, n' - m' - 1))$
{-3}	$\text{ndiv}(v', \text{expt}(2, m') \times 2 \times \text{expt}(2, n' - m' - 1)) =$
	$\text{ndiv}(\text{ndiv}(v', \text{expt}(2, m')), 2 \times \text{expt}(2, n' - m' - 1))$
{-4}	$\text{integer_pred}(q')$
{-5}	$n' > m'$
{-6}	$v' - \text{expt}(2, m') \geq 0$
{-7}	$\text{ndiv}(v', \text{expt}(2, m')) = 1 + 2 \times q'$
{-8}	$n' \geq 0$
{-9}	$m' \geq 0$
{-10}	$v' \geq 0$
{1}	$c'(m')$
{2}	$n' = m'$
{3}	$\text{rem}(2)(\text{ndiv}(v', \text{expt}(2, n'))) = 1 =$
	$(\text{rem}(2)(\text{ndiv}(v' - \text{expt}(2, m'), \text{expt}(2, n'))) = 1)$

Replacing using formula -1,

Replacing using formula -2,

Replacing using formula -3,

Using lemma `ndiv_minus_2`,

Rewriting using `divides_reflexive`, matching in `*`,

Rewriting using `ndiv_self`, matching in `*`,

Replacing using formula -1,

Replacing using formula -8,

Simplifying, rewriting, and recording with decision procedures,

Rewriting using `ndiv_times_2`, matching in `*` where b gets 2, c gets $\text{expt}(2, -1 - m! + n!)$,

Rewriting using `ndiv_times_2`, matching in `*` where b gets 2, c gets $\text{expt}(2, -1 - m! + n!)$,

Using lemma `ndiv_plus_mod`,

Splitting conjunctions,

we get 2 subgoals:

`change_bit_single_bit_op.1.2.1.1.1.1.1.1:`

<pre> {-1} ndiv(2 × q' + 1, 2) = ndiv(2 × q', 2) {-2} ndiv(v' - expt(2, m'), expt(2, m')) = 2 × q' {-3} 2 × expt(2, -1 - m' + n') × expt(2, m') = expt(2, n') {-4} ndiv(v' - expt(2, m'), expt(2, n')) = ndiv(ndiv(2 × q', 2), expt(2, -1 - m' + n')) {-5} ndiv(v', expt(2, n')) = ndiv(ndiv(1 + 2 × q', 2), expt(2, -1 - m' + n')) {-6} integer_pred(q') {-7} n' > m' {-8} v' - expt(2, m') ≥ 0 {-9} ndiv(v', expt(2, m')) = 1 + 2 × q' {-10} n' ≥ 0 {-11} m' ≥ 0 {-12} v' ≥ 0 </pre>	<pre> {1} c'(m') {2} n' = m' {3} rem(2)(ndiv(ndiv(1 + 2 × q', 2), expt(2, -1 - m' + n'))) = 1 = (rem(2)(ndiv(ndiv(2 × q', 2), expt(2, -1 - m' + n'))) = 1) </pre>
--	--

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `change_bit_single_bit_op.1.2.1.1.1.1.1.1.`

`change_bit_single_bit_op.1.2.1.1.1.1.1.2:`

<pre> {-1} ndiv(v' - expt(2, m'), expt(2, m')) = 2 × q' {-2} 2 × expt(2, -1 - m' + n') × expt(2, m') = expt(2, n') {-3} ndiv(v' - expt(2, m'), expt(2, n')) = ndiv(ndiv(2 × q', 2), expt(2, -1 - m' + n')) {-4} ndiv(v', expt(2, n')) = ndiv(ndiv(1 + 2 × q', 2), expt(2, -1 - m' + n')) {-5} integer_pred(q') {-6} n' > m' {-7} v' - expt(2, m') ≥ 0 {-8} ndiv(v', expt(2, m')) = 1 + 2 × q' {-9} n' ≥ 0 {-10} m' ≥ 0 {-11} v' ≥ 0 </pre>	<pre> {1} divides(2, 2 × q') {2} c'(m') {3} n' = m' {4} rem(2)(ndiv(ndiv(1 + 2 × q', 2), expt(2, -1 - m' + n'))) = 1 = (rem(2)(ndiv(ndiv(2 × q', 2), expt(2, -1 - m' + n'))) = 1) </pre>
--	--

Expanding the definition of divides,

Instantiating quantified variables,

This completes the proof of `change_bit_single_bit_op.1.2.1.1.1.1.1.2.`

C Proof scripts

change_bit_single_bit_op.1.2.1.1.1.1.2:

<p>{-1} $\text{ndiv}(v' - \text{expt}(2, m'), \text{expt}(2, m') \times 2 \times \text{expt}(2, n' - m' - 1))$ $=$ $\text{ndiv}(\text{ndiv}(v' - \text{expt}(2, m'), \text{expt}(2, m')),$ $2 \times \text{expt}(2, n' - m' - 1))$</p> <p>{-2} $\text{ndiv}(v', \text{expt}(2, m') \times 2 \times \text{expt}(2, n' - m' - 1)) =$ $\text{ndiv}(\text{ndiv}(v', \text{expt}(2, m')), 2 \times \text{expt}(2, n' - m' - 1))$</p> <p>{-3} $\text{integer_pred}(q')$</p> <p>{-4} $n' > m'$</p> <p>{-5} $v' - \text{expt}(2, m') \geq 0$</p> <p>{-6} $\text{ndiv}(v', \text{expt}(2, m')) = 1 + 2 \times q'$</p> <p>{-7} $n' \geq 0$</p> <p>{-8} $m' \geq 0$</p> <p>{-9} $v' \geq 0$</p>	<p>{1} $\text{expt}(2, m') \times (2 \times \text{expt}(2, n' - m' - 1)) = \text{expt}(2, n')$</p> <p>{2} $c'(m')$</p> <p>{3} $n' = m'$</p> <p>{4} $\text{rem}(2)(\text{ndiv}(v', \text{expt}(2, n'))) = 1 =$ $(\text{rem}(2)(\text{ndiv}(v' - \text{expt}(2, m'), \text{expt}(2, n'))) = 1)$</p>
---	---

Keeping (-4 1) and hiding *,

Expanding the definition of expt,

Simplifying, rewriting, and recording with decision procedures,

Using lemma expt_plus_aux,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of change_bit_single_bit_op.1.2.1.1.1.1.2.

change_bit_single_bit_op.1.2.1.1.1.1.3:

<p>{-1} $\text{ndiv}(v' - \text{expt}(2, m'), \text{expt}(2, m') \times 2 \times \text{expt}(2, n' - m' - 1))$ $=$ $\text{ndiv}(\text{ndiv}(v' - \text{expt}(2, m'), \text{expt}(2, m')),$ $2 \times \text{expt}(2, n' - m' - 1))$</p> <p>{-2} $\text{ndiv}(v', \text{expt}(2, m') \times 2 \times \text{expt}(2, n' - m' - 1)) =$ $\text{ndiv}(\text{ndiv}(v', \text{expt}(2, m')), 2 \times \text{expt}(2, n' - m' - 1))$</p> <p>{-3} $\text{integer_pred}(q')$</p> <p>{-4} $n' > m'$</p> <p>{-5} $v' - \text{expt}(2, m') \geq 0$</p> <p>{-6} $\text{ndiv}(v', \text{expt}(2, m')) = 1 + 2 \times q'$</p> <p>{-7} $n' \geq 0$</p> <p>{-8} $m' \geq 0$</p> <p>{-9} $v' \geq 0$</p>	<p>{1} $-1 - m' + n' \geq 0$</p> <p>{2} $c'(m')$</p> <p>{3} $n' = m'$</p> <p>{4} $\text{rem}(2)(\text{ndiv}(v', \text{expt}(2, n'))) = 1 =$ $(\text{rem}(2)(\text{ndiv}(v' - \text{expt}(2, m'), \text{expt}(2, n'))) = 1)$</p>
---	---

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of change_bit_single_bit_op.1.2.1.1.1.1.3.

change_bit_single_bit_op.1.2.1.1.1.2:

{-1}	$\text{ndiv}(v', \text{expt}(2, m') \times 2 \times \text{expt}(2, n' - m' - 1)) =$
	$\text{ndiv}(\text{ndiv}(v', \text{expt}(2, m')), 2 \times \text{expt}(2, n' - m' - 1))$
{-2}	$\text{integer_pred}(q')$
{-3}	$n' > m'$
{-4}	$v' - \text{expt}(2, m') \geq 0$
{-5}	$\text{ndiv}(v', \text{expt}(2, m')) = 1 + 2 \times q'$
{-6}	$n' \geq 0$
{-7}	$m' \geq 0$
{-8}	$v' \geq 0$
{1}	$-1 - m' + n' \geq 0$
{2}	$c'(m')$
{3}	$n' = m'$
{4}	$\text{rem}(2)(\text{ndiv}(v', \text{expt}(2, n'))) = 1 =$
	$(\text{rem}(2)(\text{ndiv}(v' - \text{expt}(2, m'), \text{expt}(2, n'))) = 1)$

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of change_bit_single_bit_op.1.2.1.1.1.2.

change_bit_single_bit_op.1.2.1.1.2:

{-1}	$\text{integer_pred}(q')$
{-2}	$n' > m'$
{-3}	$v' - \text{expt}(2, m') \geq 0$
{-4}	$\text{ndiv}(v', \text{expt}(2, m')) = 1 + 2 \times q'$
{-5}	$n' \geq 0$
{-6}	$m' \geq 0$
{-7}	$v' \geq 0$
{1}	$-1 - m' + n' \geq 0$
{2}	$c'(m')$
{3}	$n' = m'$
{4}	$\text{rem}(2)(\text{ndiv}(v', \text{expt}(2, n'))) = 1 =$
	$(\text{rem}(2)(\text{ndiv}(v' - \text{expt}(2, m'), \text{expt}(2, n'))) = 1)$

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of change_bit_single_bit_op.1.2.1.1.2.

change_bit_single_bit_op.1.2.1.2:

{-1}	$v' - \text{expt}(2, m') \geq 0$
{-2}	$\text{rem}(2)(\text{ndiv}(v', \text{expt}(2, m'))) = 1$
{-3}	$n' \geq 0$
{-4}	$m' \geq 0$
{-5}	$v' \geq 0$
{1}	$n' > m'$
{2}	$c'(m')$
{3}	$n' = m'$
{4}	$\text{rem}(2)(\text{ndiv}(v', \text{expt}(2, n'))) = 1 =$
	$(\text{rem}(2)(\text{ndiv}(v' - \text{expt}(2, m'), \text{expt}(2, n'))) = 1)$

Using lemma ndiv_minus_2,

Splitting conjunctions,

we get 2 subgoals:

change_bit_single_bit_op.1.2.1.2.1:

{-1}	$\text{ndiv}(v' - \text{expt}(2, m'), \text{expt}(2, n')) =$ $\text{ndiv}(v', \text{expt}(2, n')) - \text{ndiv}(\text{expt}(2, m'), \text{expt}(2, n'))$
{-2}	$v' - \text{expt}(2, m') \geq 0$
{-3}	$\text{rem}(2)(\text{ndiv}(v', \text{expt}(2, m'))) = 1$
{-4}	$n' \geq 0$
{-5}	$m' \geq 0$
{-6}	$v' \geq 0$
{1}	$n' > m'$
{2}	$c'(m')$
{3}	$n' = m'$
{4}	$\text{rem}(2)(\text{ndiv}(v', \text{expt}(2, n'))) = 1 =$ $\text{rem}(2)(\text{ndiv}(v' - \text{expt}(2, m'), \text{expt}(2, n'))) = 1$

Replacing using formula -1,

Rewriting using `ndiv_expt_expt`, matching in `*`,

Simplifying, rewriting, and recording with decision procedures,

Using lemma `rem_diff2`,

Using lemma `rem_multiple1`,

Expanding the definition of `expt`,

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `change_bit_single_bit_op.1.2.1.2.1`.

change_bit_single_bit_op.1.2.1.2.2:

{-1}	$v' - \text{expt}(2, m') \geq 0$
{-2}	$\text{rem}(2)(\text{ndiv}(v', \text{expt}(2, m'))) = 1$
{-3}	$n' \geq 0$
{-4}	$m' \geq 0$
{-5}	$v' \geq 0$
{1}	$\text{divides}(\text{expt}(2, n'), \text{expt}(2, m'))$
{2}	$n' > m'$
{3}	$c'(m')$
{4}	$n' = m'$
{5}	$\text{rem}(2)(\text{ndiv}(v', \text{expt}(2, n'))) = 1 =$ $\text{rem}(2)(\text{ndiv}(v' - \text{expt}(2, m'), \text{expt}(2, n'))) = 1$

Expanding the definition of `divides`,

Instantiating the top quantifier in 1 with the terms: $(\text{expt}(2, m!1 - n!1))$,

we get 2 subgoals:

change_bit_single_bit_op.1.2.1.2.2.1:

{-1}	$v' - \text{expt}(2, m') \geq 0$
{-2}	$\text{rem}(2)(\text{ndiv}(v', \text{expt}(2, m'))) = 1$
{-3}	$n' \geq 0$
{-4}	$m' \geq 0$
{-5}	$v' \geq 0$
{1}	$\text{expt}(2, m') = \text{expt}(2, n') \times \text{expt}(2, m' - n')$
{2}	$n' > m'$
{3}	$c'(m')$
{4}	$n' = m'$
{5}	$\text{rem}(2)(\text{ndiv}(v', \text{expt}(2, n'))) = 1 =$ $\text{rem}(2)(\text{ndiv}(v' - \text{expt}(2, m'), \text{expt}(2, n'))) = 1$

Using lemma `expt_plus_aux`,

we get 2 subgoals:

`change_bit_single_bit_op.1.2.1.2.2.1.1:`

{-1}	$\text{expt}(2, n' + m' - n') = \text{expt}(2, n') \times \text{expt}(2, m' - n')$
{-2}	$v' - \text{expt}(2, m') \geq 0$
{-3}	$\text{rem}(2)(\text{ndiv}(v', \text{expt}(2, m'))) = 1$
{-4}	$n' \geq 0$
{-5}	$m' \geq 0$
{-6}	$v' \geq 0$
{1}	$\text{expt}(2, m') = \text{expt}(2, n') \times \text{expt}(2, m' - n')$
{2}	$n' > m'$
{3}	$c'(m')$
{4}	$n' = m'$
{5}	$\text{rem}(2)(\text{ndiv}(v', \text{expt}(2, n'))) = 1 =$ $(\text{rem}(2)(\text{ndiv}(v' - \text{expt}(2, m'), \text{expt}(2, n'))) = 1)$

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `change_bit_single_bit_op.1.2.1.2.2.1.1`.

`change_bit_single_bit_op.1.2.1.2.2.1.2:`

{-1}	$v' - \text{expt}(2, m') \geq 0$
{-2}	$\text{rem}(2)(\text{ndiv}(v', \text{expt}(2, m'))) = 1$
{-3}	$n' \geq 0$
{-4}	$m' \geq 0$
{-5}	$v' \geq 0$
{1}	$m' - n' \geq 0$
{2}	$\text{expt}(2, m') = \text{expt}(2, n') \times \text{expt}(2, m' - n')$
{3}	$n' > m'$
{4}	$c'(m')$
{5}	$n' = m'$
{6}	$\text{rem}(2)(\text{ndiv}(v', \text{expt}(2, n'))) = 1 =$ $(\text{rem}(2)(\text{ndiv}(v' - \text{expt}(2, m'), \text{expt}(2, n'))) = 1)$

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `change_bit_single_bit_op.1.2.1.2.2.1.2`.

`change_bit_single_bit_op.1.2.1.2.2.2:`

{-1}	$v' - \text{expt}(2, m') \geq 0$
{-2}	$\text{rem}(2)(\text{ndiv}(v', \text{expt}(2, m'))) = 1$
{-3}	$n' \geq 0$
{-4}	$m' \geq 0$
{-5}	$v' \geq 0$
{1}	$m' - n' \geq 0$
{2}	$n' > m'$
{3}	$c'(m')$
{4}	$n' = m'$
{5}	$\text{rem}(2)(\text{ndiv}(v', \text{expt}(2, n'))) = 1 =$ $(\text{rem}(2)(\text{ndiv}(v' - \text{expt}(2, m'), \text{expt}(2, n'))) = 1)$

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `change_bit_single_bit_op.1.2.1.2.2.2`.

C Proof scripts

`change_bit_single_bit_op.1.2.2:`

$$\begin{array}{l|l}
 \{-1\} & \text{rem}(2)(\text{ndiv}(v', \text{expt}(2, m'))) = 1 \\
 \{-2\} & n' \geq 0 \\
 \{-3\} & m' \geq 0 \\
 \{-4\} & v' \geq 0 \\
 \hline
 \{1\} & v' - \text{expt}(2, m') \geq 0 \\
 \{2\} & c'(m') \\
 \{3\} & n' = m' \\
 \{4\} & \text{rem}(2)(\text{ndiv}(v', \text{expt}(2, n'))) = 1 = \\
 & (\text{rem}(2)(\text{ndiv}(v' - \text{expt}(2, m'), \text{expt}(2, n'))) = 1)
 \end{array}$$

Hiding formulas: (-2 2 3 4),

Case splitting on $v' < \text{expt}(2, m')$,

we get 2 subgoals:

`change_bit_single_bit_op.1.2.2.1:`

$$\begin{array}{l|l}
 \{-1\} & v' < \text{expt}(2, m') \\
 \{-2\} & \text{rem}(2)(\text{ndiv}(v', \text{expt}(2, m'))) = 1 \\
 \{-3\} & m' \geq 0 \\
 \{-4\} & v' \geq 0 \\
 \hline
 \{1\} & v' - \text{expt}(2, m') \geq 0
 \end{array}$$

Rewriting using `ndiv_bigger`, matching in `*`,

Rewriting using `rem_zero`, matching in `*`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `change_bit_single_bit_op.1.2.2.1`.

`change_bit_single_bit_op.1.2.2.2:`

$$\begin{array}{l|l}
 \{-1\} & \text{rem}(2)(\text{ndiv}(v', \text{expt}(2, m'))) = 1 \\
 \{-2\} & m' \geq 0 \\
 \{-3\} & v' \geq 0 \\
 \hline
 \{1\} & v' < \text{expt}(2, m') \\
 \{2\} & v' - \text{expt}(2, m') \geq 0
 \end{array}$$

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `change_bit_single_bit_op.1.2.2.2`.

change_bit_single_bit_op.2:

<pre> {-1} n' ≥ 0 {-2} m' ≥ 0 {-3} v' ≥ 0 </pre>	<pre> {1} rem(2)(ndiv(v', expt(2, m'))) = 1 {2} n' = m' {3} rem(2)(ndiv(v', expt(2, n'))) = 1 = (rem(2) (ndiv(IF c'(m') THEN IF rem(2)(ndiv(v', expt(2, m'))) = 1 THEN v' ELSE expt(2, m') + v' ENDIF ELSE IF rem(2)(ndiv(v', expt(2, m'))) = 1 THEN v' - expt(2, m') ELSE v' ENDIF, expt(2, n')))) = 1) </pre>
---	--

Case splitting on $c!1(m!1)$,

we get 2 subgoals:

change_bit_single_bit_op.2.1:

<pre> {-1} c'(m') {-2} n' ≥ 0 {-3} m' ≥ 0 {-4} v' ≥ 0 </pre>	<pre> {1} rem(2)(ndiv(v', expt(2, m'))) = 1 {2} n' = m' {3} rem(2)(ndiv(v', expt(2, n'))) = 1 = (rem(2) (ndiv(IF c'(m') THEN IF rem(2)(ndiv(v', expt(2, m'))) = 1 THEN v' ELSE expt(2, m') + v' ENDIF ELSE IF rem(2)(ndiv(v', expt(2, m'))) = 1 THEN v' - expt(2, m') ELSE v' ENDIF, expt(2, n')))) = 1) </pre>
--	--

Simplifying, rewriting, and recording with decision procedures,

Case splitting on $n!1 > m!1$,

we get 2 subgoals:

C Proof scripts

change_bit_single_bit_op.2.1.1:

{-1}	$n' > m'$
{-2}	$c'(m')$
{-3}	$n' \geq 0$
{-4}	$m' \geq 0$
{-5}	$v' \geq 0$
{1}	$\text{rem}(2)(\text{ndiv}(v', \text{expt}(2, m'))) = 1$
{2}	$n' = m'$
{3}	$\text{rem}(2)(\text{ndiv}(v', \text{expt}(2, n'))) = 1 =$ $\text{rem}(2)(\text{ndiv}(\text{expt}(2, m') + v', \text{expt}(2, n'))) = 1$

Case splitting on $\text{rem}(2)(\text{ndiv}(v!1, \text{expt}(2, m!1))) = 0$,

we get 2 subgoals:

change_bit_single_bit_op.2.1.1.1:

{-1}	$\text{rem}(2)(\text{ndiv}(v', \text{expt}(2, m'))) = 0$
{-2}	$n' > m'$
{-3}	$c'(m')$
{-4}	$n' \geq 0$
{-5}	$m' \geq 0$
{-6}	$v' \geq 0$
{1}	$\text{rem}(2)(\text{ndiv}(v', \text{expt}(2, m'))) = 1$
{2}	$n' = m'$
{3}	$\text{rem}(2)(\text{ndiv}(v', \text{expt}(2, n'))) = 1 =$ $\text{rem}(2)(\text{ndiv}(\text{expt}(2, m') + v', \text{expt}(2, n'))) = 1$

Hiding formulas: 1,

Using lemma `ndiv_times_2`,

we get 2 subgoals:

change_bit_single_bit_op.2.1.1.1.1:

{-1}	$\text{ndiv}(v', \text{expt}(2, m') \times 2 \times \text{expt}(2, n' - m' - 1)) =$ $\text{ndiv}(\text{ndiv}(v', \text{expt}(2, m')), 2 \times \text{expt}(2, n' - m' - 1))$
{-2}	$\text{rem}(2)(\text{ndiv}(v', \text{expt}(2, m'))) = 0$
{-3}	$n' > m'$
{-4}	$c'(m')$
{-5}	$n' \geq 0$
{-6}	$m' \geq 0$
{-7}	$v' \geq 0$
{1}	$n' = m'$
{2}	$\text{rem}(2)(\text{ndiv}(v', \text{expt}(2, n'))) = 1 =$ $\text{rem}(2)(\text{ndiv}(\text{expt}(2, m') + v', \text{expt}(2, n'))) = 1$

Using lemma `ndiv_times_2`,

we get 2 subgoals:

change_bit_single_bit_op.2.1.1.1.1.1.1:

{-1}	$\text{ndiv}(v' + \text{expt}(2, m'), \text{expt}(2, m') \times 2 \times \text{expt}(2, n' - m' - 1))$
	$=$
	$\text{ndiv}(\text{ndiv}(v' + \text{expt}(2, m'), \text{expt}(2, m')),$
	$2 \times \text{expt}(2, n' - m' - 1))$
{-2}	$\text{ndiv}(v', \text{expt}(2, m') \times 2 \times \text{expt}(2, n' - m' - 1)) =$
	$\text{ndiv}(\text{ndiv}(v', \text{expt}(2, m')), 2 \times \text{expt}(2, n' - m' - 1)) =$
{-3}	$\text{rem}(2)(\text{ndiv}(v', \text{expt}(2, m'))) = 0$
{-4}	$n' > m'$
{-5}	$c'(m')$
{-6}	$n' \geq 0$
{-7}	$m' \geq 0$
{-8}	$v' \geq 0$
{1}	$n' = m'$
{2}	$\text{rem}(2)(\text{ndiv}(v', \text{expt}(2, n'))) = 1 =$ $(\text{rem}(2)(\text{ndiv}(\text{expt}(2, m') + v', \text{expt}(2, n')))) = 1)$

Case splitting on $\text{expt}(2, m!1) * (2 * \text{expt}(2, n!1 - m!1 - 1)) = \text{expt}(2, n!1)$,
we get 3 subgoals:

change_bit_single_bit_op.2.1.1.1.1.1.1.1:

{-1}	$\text{expt}(2, m') \times (2 \times \text{expt}(2, n' - m' - 1)) = \text{expt}(2, n')$
{-2}	$\text{ndiv}(v' + \text{expt}(2, m'), \text{expt}(2, m') \times 2 \times \text{expt}(2, n' - m' - 1))$
	$=$
	$\text{ndiv}(\text{ndiv}(v' + \text{expt}(2, m'), \text{expt}(2, m')),$
	$2 \times \text{expt}(2, n' - m' - 1))$
{-3}	$\text{ndiv}(v', \text{expt}(2, m') \times 2 \times \text{expt}(2, n' - m' - 1)) =$
	$\text{ndiv}(\text{ndiv}(v', \text{expt}(2, m')), 2 \times \text{expt}(2, n' - m' - 1)) =$
{-4}	$\text{rem}(2)(\text{ndiv}(v', \text{expt}(2, m'))) = 0$
{-5}	$n' > m'$
{-6}	$c'(m')$
{-7}	$n' \geq 0$
{-8}	$m' \geq 0$
{-9}	$v' \geq 0$
{1}	$n' = m'$
{2}	$\text{rem}(2)(\text{ndiv}(v', \text{expt}(2, n'))) = 1 =$ $(\text{rem}(2)(\text{ndiv}(\text{expt}(2, m') + v', \text{expt}(2, n')))) = 1)$

Replacing using formula -1,

Replacing using formula -2,

Replacing using formula -3,

Using lemma `ndiv_plus_2`,

Rewriting using `divides_reflexive`, matching in `*`,

Rewriting using `ndiv_self`, matching in `*`,

Replacing using formula -1,

Hiding formulas: (-1 -2 -3 -4),

Rewriting using `rem_def`, matching in `*`,

Repeatedly Skolemizing and flattening,

Replacing using formula -2,

Rewriting using `ndiv_times_2`, matching in `*` where b gets 2, c gets $\text{expt}(2, -1 - m!1 + n!1)$,

Rewriting using `ndiv_times_2`, matching in `*` where b gets 2, c gets $\text{expt}(2, -1 - m!1 + n!1)$,

Using lemma `ndiv_plus_mod`,

Splitting conjunctions,

we get 2 subgoals:

change_bit_single_bit_op.2.1.1.1.1.1.1.1:

{-1}	$\text{ndiv}(2 \times q' + 1, 2) = \text{ndiv}(2 \times q', 2)$
{-2}	$\text{integer_pred}(q')$
{-3}	$\text{ndiv}(v', \text{expt}(2, m')) = 2 \times q'$
{-4}	$n' > m'$
{-5}	$c'(m')$
{-6}	$n' \geq 0$
{-7}	$m' \geq 0$
{-8}	$v' \geq 0$
{1}	$n' = m'$
{2}	$\text{rem}(2)(\text{ndiv}(\text{ndiv}(2 \times q', 2), \text{expt}(2, -1 - m' + n')))) = 1 =$ $(\text{rem}(2)(\text{ndiv}(\text{ndiv}(1 + 2 \times q', 2), \text{expt}(2, -1 - m' + n'))))$ $= 1)$

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of change_bit_single_bit_op.2.1.1.1.1.1.1.1.

change_bit_single_bit_op.2.1.1.1.1.1.1.2:

{-1}	$\text{integer_pred}(q')$
{-2}	$\text{ndiv}(v', \text{expt}(2, m')) = 2 \times q'$
{-3}	$n' > m'$
{-4}	$c'(m')$
{-5}	$n' \geq 0$
{-6}	$m' \geq 0$
{-7}	$v' \geq 0$
{1}	$\text{divides}(2, 2 \times q')$
{2}	$n' = m'$
{3}	$\text{rem}(2)(\text{ndiv}(\text{ndiv}(2 \times q', 2), \text{expt}(2, -1 - m' + n')))) = 1 =$ $(\text{rem}(2)(\text{ndiv}(\text{ndiv}(1 + 2 \times q', 2), \text{expt}(2, -1 - m' + n'))))$ $= 1)$

Expanding the definition of divides,

Instantiating quantified variables,

This completes the proof of change_bit_single_bit_op.2.1.1.1.1.1.1.2.

change_bit_single_bit_op.2.1.1.1.1.1.2:

{-1}	$\text{ndiv}(v' + \text{expt}(2, m'), \text{expt}(2, m') \times 2 \times \text{expt}(2, n' - m' - 1))$ $=$ $\text{ndiv}(\text{ndiv}(v' + \text{expt}(2, m'), \text{expt}(2, m')),$ $2 \times \text{expt}(2, n' - m' - 1))$
{-2}	$\text{ndiv}(v', \text{expt}(2, m') \times 2 \times \text{expt}(2, n' - m' - 1)) =$ $\text{ndiv}(\text{ndiv}(v', \text{expt}(2, m')), 2 \times \text{expt}(2, n' - m' - 1))$
{-3}	$\text{rem}(2)(\text{ndiv}(v', \text{expt}(2, m'))) = 0$
{-4}	$n' > m'$
{-5}	$c'(m')$
{-6}	$n' \geq 0$
{-7}	$m' \geq 0$
{-8}	$v' \geq 0$
{1}	$\text{expt}(2, m') \times (2 \times \text{expt}(2, n' - m' - 1)) = \text{expt}(2, n')$
{2}	$n' = m'$
{3}	$\text{rem}(2)(\text{ndiv}(v', \text{expt}(2, n'))) = 1 =$ $(\text{rem}(2)(\text{ndiv}(\text{expt}(2, m') + v', \text{expt}(2, n')))) = 1)$

Keeping (-4 1) and hiding *,

Expanding the definition of `expt`,
 Simplifying, rewriting, and recording with decision procedures,
 Using lemma `expt_plus_aux`,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 This completes the proof of `change_bit_single_bit_op.2.1.1.1.1.1.2`.
`change_bit_single_bit_op.2.1.1.1.1.1.3`:

<pre> {-1} ndiv(v' + expt(2, m'), expt(2, m') × 2 × expt(2, n' - m' - 1)) = ndiv(ndiv(v' + expt(2, m'), expt(2, m')), 2 × expt(2, n' - m' - 1)) {-2} ndiv(v', expt(2, m') × 2 × expt(2, n' - m' - 1)) = ndiv(ndiv(v', expt(2, m')), 2 × expt(2, n' - m' - 1)) {-3} rem(2)(ndiv(v', expt(2, m'))) = 0 {-4} n' > m' {-5} c'(m') {-6} n' ≥ 0 {-7} m' ≥ 0 {-8} v' ≥ 0 </pre>	<pre> {1} -1 - m' + n' ≥ 0 {2} n' = m' {3} rem(2)(ndiv(v', expt(2, n'))) = 1 = (rem(2)(ndiv(expt(2, m') + v', expt(2, n'))) = 1) </pre>
---	--

Simplifying, rewriting, and recording with decision procedures,
 This completes the proof of `change_bit_single_bit_op.2.1.1.1.1.1.3`.
`change_bit_single_bit_op.2.1.1.1.1.2`:

<pre> {-1} ndiv(v', expt(2, m') × 2 × expt(2, n' - m' - 1)) = ndiv(ndiv(v', expt(2, m')), 2 × expt(2, n' - m' - 1)) {-2} rem(2)(ndiv(v', expt(2, m'))) = 0 {-3} n' > m' {-4} c'(m') {-5} n' ≥ 0 {-6} m' ≥ 0 {-7} v' ≥ 0 </pre>	<pre> {1} -1 - m' + n' ≥ 0 {2} n' = m' {3} rem(2)(ndiv(v', expt(2, n'))) = 1 = (rem(2)(ndiv(expt(2, m') + v', expt(2, n'))) = 1) </pre>
--	--

Simplifying, rewriting, and recording with decision procedures,
 This completes the proof of `change_bit_single_bit_op.2.1.1.1.1.2`.
`change_bit_single_bit_op.2.1.1.1.2`:

<pre> {-1} rem(2)(ndiv(v', expt(2, m'))) = 0 {-2} n' > m' {-3} c'(m') {-4} n' ≥ 0 {-5} m' ≥ 0 {-6} v' ≥ 0 </pre>	<pre> {1} -1 - m' + n' ≥ 0 {2} n' = m' {3} rem(2)(ndiv(v', expt(2, n'))) = 1 = (rem(2)(ndiv(expt(2, m') + v', expt(2, n'))) = 1) </pre>
---	--

C Proof scripts

Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `change_bit_single_bit_op.2.1.1.1.2`.
`change_bit_single_bit_op.2.1.1.2`:

<pre>{-1} n' > m' {-2} c'(m') {-3} n' ≥ 0 {-4} m' ≥ 0 {-5} v' ≥ 0</pre>	<hr style="border: 0.5px solid black;"/> <pre>{1} rem(2)(ndiv(v', expt(2, m'))) = 0 {2} rem(2)(ndiv(v', expt(2, m'))) = 1 {3} n' = m' {4} rem(2)(ndiv(v', expt(2, n'))) = 1 = (rem(2)(ndiv(expt(2, m') + v', expt(2, n'))) = 1)</pre>
---	---

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `change_bit_single_bit_op.2.1.1.2`.
`change_bit_single_bit_op.2.1.2`:

<pre>{-1} c'(m') {-2} n' ≥ 0 {-3} m' ≥ 0 {-4} v' ≥ 0</pre>	<hr style="border: 0.5px solid black;"/> <pre>{1} n' > m' {2} rem(2)(ndiv(v', expt(2, m'))) = 1 {3} n' = m' {4} rem(2)(ndiv(v', expt(2, n'))) = 1 = (rem(2)(ndiv(expt(2, m') + v', expt(2, n'))) = 1)</pre>
--	--

Using lemma `ndiv_plus_2`,
Splitting conjunctions,
we get 2 subgoals:

`change_bit_single_bit_op.2.1.2.1`:

<pre>{-1} ndiv(v' + expt(2, m'), expt(2, n')) = ndiv(v', expt(2, n')) + ndiv(expt(2, m'), expt(2, n')) {-2} c'(m') {-3} n' ≥ 0 {-4} m' ≥ 0 {-5} v' ≥ 0</pre>	<hr style="border: 0.5px solid black;"/> <pre>{1} n' > m' {2} rem(2)(ndiv(v', expt(2, m'))) = 1 {3} n' = m' {4} rem(2)(ndiv(v', expt(2, n'))) = 1 = (rem(2)(ndiv(expt(2, m') + v', expt(2, n'))) = 1)</pre>
---	--

Rewriting using `ndiv_expt_expt`, matching in *,
Simplifying, rewriting, and recording with decision procedures,
Using lemma `rem_sum1`,
Using lemma `rem_multiple1`,
Expanding the definition of `expt`,
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `change_bit_single_bit_op.2.1.2.1`.

change_bit_single_bit_op.2.1.2.2:

{-1}	$c'(m')$
{-2}	$n' \geq 0$
{-3}	$m' \geq 0$
{-4}	$v' \geq 0$
{1}	$\text{divides}(\text{expt}(2, n'), \text{expt}(2, m'))$
{2}	$n' > m'$
{3}	$\text{rem}(2)(\text{ndiv}(v', \text{expt}(2, m'))) = 1$
{4}	$n' = m'$
{5}	$\text{rem}(2)(\text{ndiv}(v', \text{expt}(2, n'))) = 1 =$ $(\text{rem}(2)(\text{ndiv}(\text{expt}(2, m') + v', \text{expt}(2, n'))) = 1)$

Expanding the definition of divides,

Instantiating the top quantifier in 1 with the terms: $(\text{expt}(2, m! - n!))$,

we get 2 subgoals:

change_bit_single_bit_op.2.1.2.2.1:

{-1}	$c'(m')$
{-2}	$n' \geq 0$
{-3}	$m' \geq 0$
{-4}	$v' \geq 0$
{1}	$\text{expt}(2, m') = \text{expt}(2, n') \times \text{expt}(2, m' - n')$
{2}	$n' > m'$
{3}	$\text{rem}(2)(\text{ndiv}(v', \text{expt}(2, m'))) = 1$
{4}	$n' = m'$
{5}	$\text{rem}(2)(\text{ndiv}(v', \text{expt}(2, n'))) = 1 =$ $(\text{rem}(2)(\text{ndiv}(\text{expt}(2, m') + v', \text{expt}(2, n'))) = 1)$

Using lemma `expt_plus_aux`,

we get 2 subgoals:

change_bit_single_bit_op.2.1.2.2.1.1:

{-1}	$\text{expt}(2, n' + m' - n') = \text{expt}(2, n') \times \text{expt}(2, m' - n')$
{-2}	$c'(m')$
{-3}	$n' \geq 0$
{-4}	$m' \geq 0$
{-5}	$v' \geq 0$
{1}	$\text{expt}(2, m') = \text{expt}(2, n') \times \text{expt}(2, m' - n')$
{2}	$n' > m'$
{3}	$\text{rem}(2)(\text{ndiv}(v', \text{expt}(2, m'))) = 1$
{4}	$n' = m'$
{5}	$\text{rem}(2)(\text{ndiv}(v', \text{expt}(2, n'))) = 1 =$ $(\text{rem}(2)(\text{ndiv}(\text{expt}(2, m') + v', \text{expt}(2, n'))) = 1)$

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `change_bit_single_bit_op.2.1.2.2.1.1`.

C Proof scripts

change_bit_single_bit_op.2.1.2.2.1.2:

{-1}	$c'(m')$
{-2}	$n' \geq 0$
{-3}	$m' \geq 0$
{-4}	$v' \geq 0$
{1}	$m' - n' \geq 0$
{2}	$\text{expt}(2, m') = \text{expt}(2, n') \times \text{expt}(2, m' - n')$
{3}	$n' > m'$
{4}	$\text{rem}(2)(\text{ndiv}(v', \text{expt}(2, m'))) = 1$
{5}	$n' = m'$
{6}	$\text{rem}(2)(\text{ndiv}(v', \text{expt}(2, n'))) = 1 =$ $(\text{rem}(2)(\text{ndiv}(\text{expt}(2, m') + v', \text{expt}(2, n')))) = 1)$

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of change_bit_single_bit_op.2.1.2.2.1.2.

change_bit_single_bit_op.2.1.2.2.2:

{-1}	$c'(m')$
{-2}	$n' \geq 0$
{-3}	$m' \geq 0$
{-4}	$v' \geq 0$
{1}	$m' - n' \geq 0$
{2}	$n' > m'$
{3}	$\text{rem}(2)(\text{ndiv}(v', \text{expt}(2, m'))) = 1$
{4}	$n' = m'$
{5}	$\text{rem}(2)(\text{ndiv}(v', \text{expt}(2, n'))) = 1 =$ $(\text{rem}(2)(\text{ndiv}(\text{expt}(2, m') + v', \text{expt}(2, n')))) = 1)$

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of change_bit_single_bit_op.2.1.2.2.2.

change_bit_single_bit_op.2.2:

{-1}	$n' \geq 0$
{-2}	$m' \geq 0$
{-3}	$v' \geq 0$
{1}	$c'(m')$
{2}	$\text{rem}(2)(\text{ndiv}(v', \text{expt}(2, m'))) = 1$
{3}	$n' = m'$
{4}	$\text{rem}(2)(\text{ndiv}(v', \text{expt}(2, n'))) = 1 =$ $(\text{rem}(2)$ $(\text{ndiv}(\text{IF } c'(m')$ $\quad \text{THEN IF } \text{rem}(2)(\text{ndiv}(v', \text{expt}(2, m'))) = 1$ $\quad \quad \text{THEN } v'$ $\quad \quad \text{ELSE } \text{expt}(2, m') + v'$ $\quad \quad \text{ENDIF}$ $\quad \text{ELSE IF } \text{rem}(2)(\text{ndiv}(v', \text{expt}(2, m'))) = 1$ $\quad \quad \text{THEN } v' - \text{expt}(2, m')$ $\quad \quad \text{ELSE } v'$ $\quad \quad \text{ENDIF}$ $\quad \text{ENDIF},$ $\quad \text{expt}(2, n'))$ $= 1)$

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `change_bit_single_bit_op.2.2`.
Q.E.D.

C.16.11 Bitops.change_bit_result

Terse proof for `change_bit_result`.

`change_bit_result`:

$$\frac{}{\{1\} \quad \forall (n, v: \text{nat}, c: [\text{nat} \rightarrow \text{bool}]): \text{bit_set?}(n, \text{change_bit}(v, c)(n)) = c(n)}$$

Repeatedly Skolemizing and flattening,
Trying repeated skolemization, instantiation, and if-lifting,
we get 2 subgoals:

`change_bit_result.1`:

$$\frac{\begin{array}{l} \{-1\} \quad n' \geq 0 \\ \{-2\} \quad v' \geq 0 \\ \{-3\} \quad \text{rem}(2)(\text{ndiv}(v', \text{expt}(2, n'))) = 1 \\ \{-4\} \quad \text{rem}(2)(\text{ndiv}(v' - \text{expt}(2, n'), \text{expt}(2, n'))) = 1 \end{array}}{\{1\} \quad c'(n')}$$

Using lemma `ndiv_minus_2`,
Rewriting using `divides_reflexive`, matching in *,
Rewriting using `ndiv_self`, matching in *,
Using lemma `rem_add_one`,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `change_bit_result.1`.
`change_bit_result.2`:

$$\frac{\begin{array}{l} \{-1\} \quad n' \geq 0 \\ \{-2\} \quad v' \geq 0 \\ \{-3\} \quad c'(n') \end{array}}{\begin{array}{l} \{1\} \quad \text{rem}(2)(\text{ndiv}(v', \text{expt}(2, n'))) = 1 \\ \{2\} \quad \text{rem}(2)(\text{ndiv}(\text{expt}(2, n') + v', \text{expt}(2, n'))) = 1 \end{array}}$$

Using lemma `ndiv_plus_1`,
Rewriting using `divides_reflexive`, matching in *,
Rewriting using `ndiv_self`, matching in *,
Using lemma `rem_add_one`,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `change_bit_result.2`.
Q.E.D.

C.16.12 Bitops.change_bits_TCC1

Terse proof for `change_bits_TCC1`.

`change_bits_TCC1`:

$$\frac{}{\{1\} \quad \forall (n: \text{nat}): \neg n = 0 \supset n - 1 \geq 0}$$

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `change_bits_TCC1`.
Q.E.D.

C.16.13 Bitops.change_bits_TCC2

Terse proof for `change_bits_TCC2`.

`change_bits_TCC2`:

$$\frac{}{\{1\} \quad \forall (n: \text{nat}): \neg n = 0 \supset n - 1 < n}$$

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `change_bits_TCC2`.

Q.E.D.

C.16.14 Bitops.change_bits_result

Terse proof for `change_bits_result`.

`change_bits_result`:

$$\frac{}{\{1\} \quad \forall (m, n: \text{nat}, \text{val}: \text{nat}, c: [\text{nat} \rightarrow \text{bool}]): \\ m \leq n \supset \text{bit_set?}(m, \text{change_bits}(\text{val}, c)(n)) = c(m)}$$

Inducting on n on formula 1,

we get 2 subgoals:

`change_bits_result.1`:

$$\frac{}{\{1\} \quad \forall (m, \text{val}: \text{nat}, c: [\text{nat} \rightarrow \text{bool}]): \\ m \leq 0 \supset \text{bit_set?}(m, \text{change_bits}(\text{val}, c)(0)) = c(m)}$$

Repeatedly Skolemizing and flattening,

Case splitting on $m!1 = 0$,

we get 2 subgoals:

`change_bits_result.1.1`:

$$\frac{\begin{array}{l} \{-1\} \quad m' = 0 \\ \{-2\} \quad m' \geq 0 \\ \{-3\} \quad \text{val}' \geq 0 \\ \{-4\} \quad m' \leq 0 \end{array}}{\{1\} \quad \text{bit_set?}(m', \text{change_bits}(\text{val}', c')(0)) = c'(m')}$$

Replacing using formula -1,

Expanding the definition of `change_bits`,

Using lemma `change_bit_result`,

This completes the proof of `change_bits_result.1.1`.

`change_bits_result.1.2`:

$$\frac{\begin{array}{l} \{-1\} \quad m' \geq 0 \\ \{-2\} \quad \text{val}' \geq 0 \\ \{-3\} \quad m' \leq 0 \end{array}}{\begin{array}{l} \{1\} \quad m' = 0 \\ \{2\} \quad \text{bit_set?}(m', \text{change_bits}(\text{val}', c')(0)) = c'(m') \end{array}}$$

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `change_bits_result.1.2`.

change_bits_result.2:

$\{1\} \quad \forall j:$ $\quad (\forall (m, \text{val}: \text{nat}, c: [\text{nat} \rightarrow \text{bool}]):$ $\quad \quad m \leq j \supset \text{bit_set?}(m, \text{change_bits}(\text{val}, c)(j)) = c(m))$ $\quad \supset$ $\quad (\forall (m, \text{val}: \text{nat}, c: [\text{nat} \rightarrow \text{bool}]):$ $\quad \quad m \leq j + 1 \supset \text{bit_set?}(m, \text{change_bits}(\text{val}, c)(j + 1)) = c(m))$
--

Repeatedly Skolemizing and flattening,

Expanding the definition of change_bits,

Case splitting on $m! = j! + 1$,

we get 2 subgoals:

change_bits_result.2.1:

$\{-1\} \quad m' = j' + 1$ $\{-2\} \quad m' \geq 0$ $\{-3\} \quad \text{val}' \geq 0$ $\{-4\} \quad j' \geq 0$ $\{-5\} \quad \forall (m, \text{val}: \text{nat}, c: [\text{nat} \rightarrow \text{bool}]):$ $\quad m \leq j' \supset \text{bit_set?}(m, \text{change_bits}(\text{val}, c)(j')) = c(m)$ $\{-6\} \quad m' \leq j' + 1$
$\{1\} \quad \text{bit_set?}(m', \text{change_bit}(\text{change_bits}(\text{val}', c')(j'), c')(1 + j')) = c'(m')$

Using lemma change_bit_result,

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of change_bits_result.2.1.

change_bits_result.2.2:

$\{-1\} \quad m' \geq 0$ $\{-2\} \quad \text{val}' \geq 0$ $\{-3\} \quad j' \geq 0$ $\{-4\} \quad \forall (m, \text{val}: \text{nat}, c: [\text{nat} \rightarrow \text{bool}]):$ $\quad m \leq j' \supset \text{bit_set?}(m, \text{change_bits}(\text{val}, c)(j')) = c(m)$ $\{-5\} \quad m' \leq j' + 1$
$\{1\} \quad m' = j' + 1$ $\{2\} \quad \text{bit_set?}(m', \text{change_bit}(\text{change_bits}(\text{val}', c')(j'), c')(1 + j')) = c'(m')$

Using lemma change_bit_single_bit_op,

Simplifying, rewriting, and recording with decision procedures,

Replacing using formula -1,

Instantiating quantified variables,

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of change_bits_result.2.2.

Q.E.D.

C.16.15 Bitops.change_bits_unchanged

Terse proof for change_bits_unchanged.

change_bits_unchanged:

$\{1\} \quad \forall (m, n: \text{nat}, \text{val}: \text{nat}, c: [\text{nat} \rightarrow \text{bool}]):$ $\quad m > n \supset \text{bit_set?}(m, \text{change_bits}(\text{val}, c)(n)) = \text{bit_set?}(m, \text{val})$

C Proof scripts

Inducting on n on formula 1,
we get 2 subgoals:
change_bits_unchanged.1:

$$\{1\} \quad \forall (m, \text{val}: \text{nat}, c: [\text{nat} \rightarrow \text{bool}]):$$
$$m > 0 \supset \text{bit_set?}(m, \text{change_bits}(\text{val}, c)(0)) = \text{bit_set?}(m, \text{val})$$

Repeatedly Skolemizing and flattening,
Expanding the definition of change_bits,
Using lemma change_bit_single_bit_op,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of change_bits_unchanged.1.
change_bits_unchanged.2:

$$\{1\} \quad \forall j:$$
$$(\forall (m, \text{val}: \text{nat}, c: [\text{nat} \rightarrow \text{bool}]):$$
$$m > j \supset \text{bit_set?}(m, \text{change_bits}(\text{val}, c)(j)) = \text{bit_set?}(m, \text{val}))$$
$$\supset$$
$$(\forall (m, \text{val}: \text{nat}, c: [\text{nat} \rightarrow \text{bool}]):$$
$$m > j + 1 \supset$$
$$\text{bit_set?}(m, \text{change_bits}(\text{val}, c)(j + 1)) = \text{bit_set?}(m, \text{val}))$$

Repeatedly Skolemizing and flattening,
Expanding the definition of change_bits,
Using lemma change_bit_single_bit_op,
Simplifying, rewriting, and recording with decision procedures,
Replacing using formula -1,
Instantiating quantified variables,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of change_bits_unchanged.2.
Q.E.D.

C.16.16 Bitops.change_bits_below_TCC1

Terse proof for change_bits_below_TCC1.
change_bits_below_TCC1:

$$\{1\} \quad \forall (v: \text{below}[\text{expt}(2, p)]): p - 1 \geq 0$$

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of change_bits_below_TCC1.
Q.E.D.

C.16.17 Bitops.change_bits_below

Terse proof for change_bits_below.
change_bits_below:

$$\{1\} \quad \forall (v: \text{below}[\text{expt}(2, p)], c: [\text{nat} \rightarrow \text{bool}]):$$
$$\text{change_bits}(v, c)(p - 1) < \text{expt}(2, p)$$

Repeatedly Skolemizing and flattening,

Using lemma number_split,
we get 2 subgoals:
change_bits_below.1:

$$\left| \begin{array}{l} \{-1\} \quad \text{change_bits}(v', c')(p-1) = \\ \quad \text{expt}(2, p) \times \text{ndiv}(\text{change_bits}(v', c')(p-1), \text{expt}(2, p)) + \text{rem}(\text{expt}(2, p))(\text{change_bits}(v', c')(p-1)) \\ \{-2\} \quad v' < \text{expt}(2, p) \\ \hline \{1\} \quad \text{change_bits}(v', c')(p-1) < \text{expt}(2, p) \end{array} \right.$$

Replacing using formula -1,
Hiding formulas: -1,
Using lemma bit_split_less,
we get 2 subgoals:

change_bits_below.1.1:

$$\left| \begin{array}{l} \{-1\} \quad \text{rem}(\text{expt}(2, p))(\text{change_bits}(v', c')(p-1)) < \text{expt}(2, p) \supset \\ \quad (\text{ndiv}(\text{change_bits}(v', c')(p-1), \text{expt}(2, p)) \times \text{expt}(2, p) + \text{rem}(\text{expt}(2, p))(\text{change_bits}(v', c')(p-1))) \\ \quad < \text{expt}(2, p+0) \\ \quad \equiv \text{ndiv}(\text{change_bits}(v', c')(p-1), \text{expt}(2, p)) < \text{expt}(2, 0) \\ \{-2\} \quad v' < \text{expt}(2, p) \\ \hline \{1\} \quad \text{expt}(2, p) \times \text{ndiv}(\text{change_bits}(v', c')(p-1), \text{expt}(2, p)) + \text{rem}(\text{expt}(2, p))(\text{change_bits}(v', c')(p-1)) \\ \quad < \text{expt}(2, p) \end{array} \right.$$

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Hiding formulas: (1 3),
Applying change_bits_unchanged
Instantiating the top quantifier in -1 with the terms: ($_ p - 1 v!1 c!1$),
Using lemma bits_equal4,
Splitting conjunctions,
we get 2 subgoals:

change_bits_below.1.1.1:

$$\left| \begin{array}{l} \{-1\} \quad \text{ndiv}(\text{change_bits}(v', c')(p-1), \text{expt}(2, p)) = \text{ndiv}(v', \text{expt}(2, p)) \\ \{-2\} \quad \forall (m: \text{nat}): \\ \quad m > p-1 \supset \\ \quad \quad \text{bit_set?}(m, \text{change_bits}(v', c')(p-1)) = \text{bit_set?}(m, v') \\ \{-3\} \quad v' < \text{expt}(2, p) \\ \hline \{1\} \quad \text{ndiv}(\text{change_bits}(v', c')(p-1), \text{expt}(2, p)) < \text{expt}(2, 0) \end{array} \right.$$

Replacing using formula -1,
Hiding formulas: (-1 -2),
Rewriting using ndiv_bigger, matching in *,
Expanding the definition of expt,
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of change_bits_below.1.1.1.

change_bits_below.1.1.2:

$$\left| \begin{array}{l} \{-1\} \quad \forall (m: \text{nat}): \\ \quad m > p-1 \supset \\ \quad \quad \text{bit_set?}(m, \text{change_bits}(v', c')(p-1)) = \text{bit_set?}(m, v') \\ \{-2\} \quad v' < \text{expt}(2, p) \\ \hline \{1\} \quad \forall (n: \text{nat}): \\ \quad n \geq p \supset \text{bit_set?}(n, \text{change_bits}(v', c')(p-1)) = \text{bit_set?}(n, v') \\ \{2\} \quad \text{ndiv}(\text{change_bits}(v', c')(p-1), \text{expt}(2, p)) < \text{expt}(2, 0) \end{array} \right.$$

Repeatedly Skolemizing and flattening,

C Proof scripts

Instantiating quantified variables,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `change_bits_below.1.1.2`.
`change_bits_below.1.2`:

$$\begin{array}{|l} \{-1\} \quad v' < \text{expt}(2, p) \\ \{1\} \quad p - 1 \geq 0 \\ \{2\} \quad \text{expt}(2, p) \times \text{ndiv}(\text{change_bits}(v', c')(p - 1), \text{expt}(2, p)) + \text{rem}(\text{expt}(2, p))(\text{change_bits}(v', c')(p - 1)) < \text{expt}(2, p) \end{array}$$

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `change_bits_below.1.2`.
`change_bits_below.2`:

$$\begin{array}{|l} \{-1\} \quad v' < \text{expt}(2, p) \\ \{1\} \quad p - 1 \geq 0 \\ \{2\} \quad \text{change_bits}(v', c')(p - 1) < \text{expt}(2, p) \end{array}$$

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `change_bits_below.2`.
Q.E.D.

C.16.18 Bitops.binary_not_TCC1

Terse proof for `binary_not_TCC1`.
`binary_not_TCC1`:

$$\begin{array}{|l} \{1\} \quad \forall (\text{val}: \text{below}[\text{expt}(2, p)]): \\ \quad \text{change_bits}(\text{val}, \lambda (n: \text{nat}): \neg \text{bit_set?}(n, \text{val}))(p - 1) < \text{expt}(2, p) \end{array}$$

Repeatedly Skolemizing and flattening,
Rewriting using `change_bits_below`, matching in *,
This completes the proof of `binary_not_TCC1`.
Q.E.D.

C.16.19 Bitops.binary_not_not

Terse proof for `binary_not_not`.
`binary_not_not`:

$$\begin{array}{|l} \{1\} \quad \forall (\text{val}: \text{below}[\text{expt}(2, p)]): \text{binary_not}(\text{binary_not}(\text{val})) = \text{val} \end{array}$$

Repeatedly Skolemizing and flattening,
Using lemma `bits_equal2`,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Hiding formulas: 2,
Repeatedly Skolemizing and flattening,
Expanding the definition of `binary_not`,
Using lemma `change_bits_result`,
Simplifying, rewriting, and recording with decision procedures,
Replacing using formula -1,
Hiding formulas: -1,
Using lemma `change_bits_result`,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `binary_not_not`.
Q.E.D.

C.16.20 Bitops.binary_and_TCC1

Terse proof for `binary_and_TCC1`.

`binary_and_TCC1`:

$$\frac{\{1\} \quad \forall (v_1, v_2: \text{below}[\text{expt}(2, p)]): \quad \text{change_bits}(v_1, \lambda (n: \text{nat}): \text{bit_set?}(n, v_1) \wedge \text{bit_set?}(n, v_2))(p-1) < \text{expt}(2, p)}{}{}$$

Repeatedly Skolemizing and flattening,
Rewriting using `change_bits_below`, matching in *,
This completes the proof of `binary_and_TCC1`.
Q.E.D.

C.16.21 Bitops.binary_and_not

Terse proof for `binary_and_not`.

`binary_and_not`:

$$\frac{\{1\} \quad \forall (\text{val}: \text{below}[\text{expt}(2, p)]): \text{binary_and}(\text{val}, \text{binary_not}(\text{val})) = 0}{}{}$$

Repeatedly Skolemizing and flattening,
Using lemma `bits_equal2`,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Hiding formulas: 2,
Repeatedly Skolemizing and flattening,
Expanding the definition of `binary_and`,
Using lemma `change_bits_result`,
Simplifying, rewriting, and recording with decision procedures,
Replacing using formula -1,
Hiding formulas: -1,
Expanding the definition of `binary_not`,
Using lemma `change_bits_result`,
Simplifying, rewriting, and recording with decision procedures,
Replacing using formula -1,
Expanding the definition of `bit_set?`,
Using lemma `cut_bit_zero`,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `binary_and_not`.
Q.E.D.

C.16.22 Bitops.binary_and_full_TCC1

Terse proof for `binary_and_full_TCC1`.

`binary_and_full_TCC1`:

$$\frac{\{1\} \quad \forall (\text{val}: \text{below}[\text{expt}(2, p)]): \quad \text{expt}(2, p) - 1 \geq 0 \wedge \text{expt}(2, p) - 1 < \text{expt}(2, p)}{}{}$$

Repeatedly Skolemizing and flattening,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 Using lemma `expt_ge1`,
 Expanding the definition of `^`,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 This completes the proof of `binary_and_full_TCC1`.
 Q.E.D.

C.16.23 `Bitops.binary_and_full`

Terse proof for `binary_and_full`.

`binary_and_full`:

$$\frac{}{\{1\} \quad \forall (\text{val}: \text{below}[\text{expt}(2, p)]): \text{binary_and}(\text{val}, \text{expt}(2, p) - 1) = \text{val}}$$

Repeatedly Skolemizing and flattening,
 Using lemma `bits_equal2`,
 we get 2 subgoals:

`binary_and_full.1`:

$$\frac{\begin{array}{l} \{-1\} \quad (\forall (n: \text{nat}): \\ \quad \quad n < p \supset \\ \quad \quad \quad \text{bit_set?}(n, \text{binary_and}(\text{val}', \text{expt}(2, p) - 1)) = \text{bit_set?}(n, \text{val}')) \\ \quad \quad \equiv \text{binary_and}(\text{val}', \text{expt}(2, p) - 1) = \text{val}' \\ \{-2\} \quad \text{val}' < \text{expt}(2, p) \end{array}}{\{1\} \quad \text{binary_and}(\text{val}', \text{expt}(2, p) - 1) = \text{val}'}$$

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 Repeatedly Skolemizing and flattening,
 Hiding formulas: 2,
 Expanding the definition of `binary_and`,
 Using lemma `change_bits_result`,
 we get 2 subgoals:

`binary_and_full.1.1`:

$$\frac{\begin{array}{l} \{-1\} \quad n' \leq p - 1 \supset \\ \quad \quad \text{bit_set?}(n', \\ \quad \quad \quad \text{change_bits}(\text{val}', \\ \quad \quad \quad \quad \lambda (n: \text{nat}): \\ \quad \quad \quad \quad \quad \text{bit_set?}(n, \text{val}') \wedge \text{bit_set?}(n, \text{expt}(2, p) - 1)) \\ \quad \quad \quad \quad (p - 1)) \\ \quad \quad = (\text{bit_set?}(n', \text{val}') \wedge \text{bit_set?}(n', \text{expt}(2, p) - 1)) \\ \{-2\} \quad n' \geq 0 \\ \{-3\} \quad n' < p \\ \{-4\} \quad \text{val}' < \text{expt}(2, p) \end{array}}{\{1\} \quad \text{bit_set?}(n', \\ \quad \quad \text{change_bits}(\text{val}', \\ \quad \quad \quad \lambda (n: \text{nat}): \\ \quad \quad \quad \quad \text{bit_set?}(n, \text{val}') \wedge \text{bit_set?}(n, \text{expt}(2, p) - 1)) \\ \quad \quad \quad (p - 1)) \\ \quad \quad = \text{bit_set?}(n', \text{val}')}$$

Simplifying, rewriting, and recording with decision procedures,
 Replacing using formula -1,

Hiding formulas: -1,
 Rewriting using bit_set_full_mask, matching in *,
 This completes the proof of binary_and_full.1.1.
 binary_and_full.1.2:

$\begin{array}{l} \{-1\} \quad n' \geq 0 \\ \{-2\} \quad n' < p \\ \{-3\} \quad \text{val}' < \text{expt}(2, p) \end{array}$	<hr style="border: 0.5px solid black;"/> $\begin{array}{l} \{1\} \quad \forall (n: \text{nat}): \text{bit_set?}(n, \text{val}') \supset \text{expt}(2, p) - 1 \geq 0 \\ \{2\} \quad \text{bit_set?}(n', \\ \qquad \qquad \qquad \text{change_bits}(\text{val}', \\ \qquad \qquad \qquad \lambda (n: \text{nat}): \\ \qquad \qquad \qquad \text{bit_set?}(n, \text{val}') \wedge \text{bit_set?}(n, \text{expt}(2, p) - 1)) \\ \qquad \qquad \qquad (p - 1)) \\ \qquad \qquad \qquad = \text{bit_set?}(n', \text{val}') \end{array}$
--	---

Repeatedly Skolemizing and flattening,
 Using lemma expt_ge1,
 Expanding the definition of ^,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 This completes the proof of binary_and_full.1.2.
 binary_and_full.2:

$\begin{array}{l} \{-1\} \quad \text{val}' < \text{expt}(2, p) \\ \{1\} \quad \text{expt}(2, p) - 1 \geq 0 \\ \{2\} \quad \text{binary_and}(\text{val}', \text{expt}(2, p) - 1) = \text{val}' \end{array}$	<hr style="border: 0.5px solid black;"/>
---	--

Expanding the definition of binary_and,
 Using lemma expt_ge1,
 Expanding the definition of ^,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 This completes the proof of binary_and_full.2.
 Q.E.D.

C.16.24 Bitops.binary_and_zero_TCC1

Terse proof for binary_and_zero_TCC1.
 binary_and_zero_TCC1:

$\{1\} \quad \forall (\text{val}: \text{below}[\text{expt}(2, p)]): 0 < \text{expt}(2, p)$	<hr style="border: 0.5px solid black;"/>
--	--

Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of binary_and_zero_TCC1.
 Q.E.D.

C.16.25 Bitops.binary_and_zero

Terse proof for binary_and_zero.
 binary_and_zero:

$\{1\} \quad \forall (\text{val}: \text{below}[\text{expt}(2, p)]): \text{binary_and}(\text{val}, 0) = 0$	<hr style="border: 0.5px solid black;"/>
--	--

Repeatedly Skolemizing and flattening,

Using lemma `bits_equal2`,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 Repeatedly Skolemizing and flattening,
 Hiding formulas: 2,
 Expanding the definition of `binary_and`,
 Rewriting using `change_bits_result`, matching in *,
 Rewriting using `bit_set_zero_mask`, matching in *,
 This completes the proof of `binary_and_zero`.
 Q.E.D.

C.16.26 Bitops.binary_or_TCC1

Terse proof for `binary_or_TCC1`.

`binary_or_TCC1`:

$$\{1\} \quad \forall (v_1, v_2: \text{below}[\text{expt}(2, p)]): \\ \text{change_bits}(v_1, \lambda (n: \text{nat}): \text{bit_set?}(n, v_1) \vee \text{bit_set?}(n, v_2))(p-1) < \\ \text{expt}(2, p)$$

Repeatedly Skolemizing and flattening,
 Rewriting using `change_bits_below`, matching in *,
 This completes the proof of `binary_or_TCC1`.
 Q.E.D.

C.16.27 Bitops.binary_or_not

Terse proof for `binary_or_not`.

`binary_or_not`:

$$\{1\} \quad \forall (\text{val}: \text{below}[\text{expt}(2, p)]): \text{binary_or}(\text{val}, \text{binary_not}(\text{val})) = \text{expt}(2, p) - 1$$

Repeatedly Skolemizing and flattening,
 Using lemma `bits_equal2`,
 we get 2 subgoals:
`binary_or_not.1`:

$$\begin{array}{l} \{-1\} \quad (\forall (n: \text{nat}): \\ \quad \quad n < p \supset \\ \quad \quad \text{bit_set?}(n, \text{binary_or}(\text{val}', \text{binary_not}(\text{val}')) = \\ \quad \quad \text{bit_set?}(n, \text{expt}(2, p) - 1)) \\ \quad \quad \equiv \text{binary_or}(\text{val}', \text{binary_not}(\text{val}')) = \text{expt}(2, p) - 1 \\ \{-2\} \quad \text{val}' < \text{expt}(2, p) \\ \{1\} \quad \text{binary_or}(\text{val}', \text{binary_not}(\text{val}')) = \text{expt}(2, p) - 1 \end{array}$$

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 Hiding formulas: 2,
 Repeatedly Skolemizing and flattening,
 Expanding the definition of `binary_or`,
 Rewriting using `change_bits_result`, matching in *,
 Expanding the definition of `binary_not`,
 Rewriting using `change_bits_result`, matching in *,
 Rewriting using `bit_set_full_mask`, matching in *,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `binary_or_not.1`.

`binary_or_not.2`:

$$\begin{array}{|l} \{-1\} \quad \text{val}' < \text{expt}(2, p) \\ \{1\} \quad \text{expt}(2, p) - 1 \geq 0 \\ \{2\} \quad \text{binary_or}(\text{val}', \text{binary_not}(\text{val}')) = \text{expt}(2, p) - 1 \end{array}$$

Using lemma `expt_ge1`,

Expanding the definition of \wedge ,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `binary_or_not.2`.

Q.E.D.

C.16.28 Bitops.binary_xor_TCC1

Terse proof for `binary_xor_TCC1`.

`binary_xor_TCC1`:

$$\begin{array}{|l} \{1\} \quad \forall (v_1, v_2: \text{below}[\text{expt}(2, p)]): \\ \quad \text{change_bits}(v_1, \lambda (n: \text{nat}): \text{bit_set?}(n, v_1) \neq \text{bit_set?}(n, v_2))(p-1) < \\ \quad \text{expt}(2, p) \end{array}$$

Repeatedly Skolemizing and flattening,

Rewriting using `change_bits_below`, matching in $*$,

This completes the proof of `binary_xor_TCC1`.

Q.E.D.

C.16.29 Bitops.binary_xor_self

Terse proof for `binary_xor_self`.

`binary_xor_self`:

$$\begin{array}{|l} \{1\} \quad \forall (\text{val}: \text{below}[\text{expt}(2, p)]): \text{binary_xor}(\text{val}, \text{val}) = 0 \end{array}$$

Expanding the definition of `binary_xor`,

Repeatedly Skolemizing and flattening,

Using lemma `bits_equal2`,

we get 3 subgoals:

`binary_xor_self.1`:

$$\begin{array}{|l} \{-1\} \quad (\forall (n_1: \text{nat}): \\ \quad n_1 < p \supset \\ \quad \text{bit_set?}(n_1, \text{change_bits}(\text{val}', \lambda (n: \text{nat}): \text{FALSE}))(p-1) = \\ \quad \text{bit_set?}(n_1, 0)) \\ \quad \equiv \text{change_bits}(\text{val}', \lambda (n: \text{nat}): \text{FALSE})(p-1) = 0 \\ \{-2\} \quad \text{val}' < \text{expt}(2, p) \\ \{1\} \quad \text{change_bits}(\text{val}', \lambda (n: \text{nat}): \text{FALSE})(p-1) = 0 \end{array}$$

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Repeatedly Skolemizing and flattening,

Rewriting using `bit_set_zero_mask`, matching in $*$,

Rewriting using `change_bits_result`, matching in $*$,

This completes the proof of `binary_xor_self.1`.

binary_xor_self.2:

{-1}	$val' < \text{expt}(2, p)$
{1}	$\text{change_bits}(val', \lambda (n: \text{nat}): \text{FALSE})(p-1) < \text{expt}(2, p)$
{2}	$\text{change_bits}(val', \lambda (n: \text{nat}): \text{FALSE})(p-1) = 0$

Using lemma change_bits_below,

This completes the proof of binary_xor_self.2.

binary_xor_self.3:

{-1}	$val' < \text{expt}(2, p)$
{1}	$p - 1 \geq 0$
{2}	$\text{change_bits}(val', \lambda (n: \text{nat}): \text{FALSE})(p-1) = 0$

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of binary_xor_self.3.

Q.E.D.

C.16.30 Bitops.binary_or_full

Terse proof for binary_or_full.

binary_or_full:

{1}	$\forall (val: \text{below}[\text{expt}(2, p)]):$ $\text{binary_or}(val, \text{expt}(2, p) - 1) = \text{expt}(2, p) - 1$
-----	--

Repeatedly Skolemizing and flattening,

Using lemma bits_equal2,

we get 2 subgoals:

binary_or_full.1:

{-1}	$(\forall (n: \text{nat}):$ $n < p \supset$ $\text{bit_set?}(n, \text{binary_or}(val', \text{expt}(2, p) - 1)) =$ $\text{bit_set?}(n, \text{expt}(2, p) - 1))$ $\equiv \text{binary_or}(val', \text{expt}(2, p) - 1) = \text{expt}(2, p) - 1$
{-2}	$val' < \text{expt}(2, p)$
{1}	$\text{binary_or}(val', \text{expt}(2, p) - 1) = \text{expt}(2, p) - 1$

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Repeatedly Skolemizing and flattening,

Expanding the definition of binary_or,

Hiding formulas: 2,

Rewriting using change_bits_result, matching in *,

we get 2 subgoals:

binary_or_full.1.1:

{-1}	$n' \geq 0$
{-2}	$n' < p$
{-3}	$val' < \text{expt}(2, p)$
{1}	$(\text{bit_set?}(n', val') \vee \text{bit_set?}(n', \text{expt}(2, p) - 1)) =$ $\text{bit_set?}(n', \text{expt}(2, p) - 1)$

Rewriting using bit_set_full_mask, matching in *,

This completes the proof of binary_or_full.1.1.

binary_or_full.1.2:

$\{-1\}$ $n' \geq 0$ $\{-2\}$ $n' < p$ $\{-3\}$ $val' < \text{expt}(2, p)$	$\{1\}$ $\forall (n: \text{nat}): \neg \text{bit_set?}(n, val') \supset \text{expt}(2, p) - 1 \geq 0$ $\{2\}$ $\text{bit_set?}(n',$ <div style="margin-left: 100px;"> $\text{change_bits}(val',$ <div style="margin-left: 20px;"> $\lambda (n: \text{nat}):$ $\text{bit_set?}(n, val') \vee \text{bit_set?}(n, \text{expt}(2, p) - 1)$ $(p - 1)$ </div> </div> $) = \text{bit_set?}(n', \text{expt}(2, p) - 1)$
--	--

Repeatedly Skolemizing and flattening,

Using lemma `expt_ge1`,

Expanding the definition of \wedge ,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `binary_or_full.1.2`.

binary_or_full.2:

$\{-1\}$ $val' < \text{expt}(2, p)$	$\{1\}$ $\text{expt}(2, p) - 1 \geq 0$ $\{2\}$ $\text{binary_or}(val', \text{expt}(2, p) - 1) = \text{expt}(2, p) - 1$
-------------------------------------	--

Expanding the definition of `binary_or`,

Using lemma `expt_ge1`,

Expanding the definition of \wedge ,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `binary_or_full.2`.

Q.E.D.

C.16.31 Bits.binary_or_zero

Terse proof for `binary_or_zero`.

binary_or_zero:

$\{1\}$ $\forall (val: \text{below}[\text{expt}(2, p)]): \text{binary_or}(val, 0) = val$

Repeatedly Skolemizing and flattening,

Using lemma `bits_equal2`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Repeatedly Skolemizing and flattening,

Expanding the definition of `binary_or`,

Rewriting using `change_bits_result`, matching in $*$,

Rewriting using `bit_set_zero_mask`, matching in $*$,

This completes the proof of `binary_or_zero`.

Q.E.D.

C.17 Proofs for Bits (bits.pvs)

C.17.1 Bits.bool_to_nat_below

Terse proof for `bool_to_nat_below`.

C Proof scripts

`bool_to_nat_below`:

$$\{1\} \quad \forall (b: \text{bool}): \text{bool_to_nat}(b) < 2$$

Repeatedly Skolemizing and flattening,
Expanding the definition of `bool_to_nat`,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `bool_to_nat_below`.
Q.E.D.

C.17.2 Bits.nat_bool_iso

Terse proof for `nat_bool_iso`.

`nat_bool_iso`:

$$\{1\} \quad \forall (b: \text{bool}): \text{nat_to_bool}(\text{bool_to_nat}(b)) = b$$

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `nat_bool_iso`.
Q.E.D.

C.17.3 Bits.cut_bit_zero

Terse proof for `cut_bit_zero`.

`cut_bit_zero`:

$$\{1\} \quad \forall (i: \text{nat}): \text{cut_bit}(0, i) = \text{FALSE}$$

Repeatedly Skolemizing and flattening,
Expanding the definition of `cut_bit`,
Rewriting using `ndiv_0`, matching in `*`,
Rewriting using `rem_zero`, matching in `*`,
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `cut_bit_zero`.
Q.E.D.

C.17.4 Bits.cut_bits_below

Terse proof for `cut_bits_below`.

`cut_bits_below`:

$$\{1\} \quad \forall (i, k, n: \text{nat}): \text{cut_bits}(n, i, k) < \text{expt}(2, k)$$

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `cut_bits_below`.
Q.E.D.

C.17.5 Bits.cut_bits_aligned

Terse proof for `cut_bits_aligned`.

cut_bits_aligned:

$$\frac{}{\{1\} \quad \forall (n, i, k: \text{nat}): i \leq k \wedge \text{cut_bits}(n, 0, k) = 0 \supset \text{aligned?}(i)(n)}$$

Repeatedly Skolemizing and flattening,
 Expanding the definition of cut_bits,
 Expanding the definition of expt,
 Rewriting using ndiv_1, matching in *,
 Expanding the definition of aligned?,
 Rewriting using rem_def2, matching in *,
 Applying divides_expt_gt
 Instantiating the top quantifier in -1 with the terms: n', k', i' ,
 Simplifying, rewriting, and recording with decision procedures,
 This completes the proof of cut_bits_aligned.
 Q.E.D.

C.17.6 Bits.aligned_cut_bits

Terse proof for aligned_cut_bits.

aligned_cut_bits:

$$\frac{}{\{1\} \quad \forall (n, i, j: \text{nat}, k: \text{nat}): i + j \leq k \wedge \text{aligned?}(k)(n) \supset \text{cut_bits}(n, i, j) = 0}$$

Repeatedly Skolemizing and flattening,
 Expanding the definition of aligned?,
 Expanding the definition of cut_bits,
 Rewriting using rem_ndiv, matching in *,
 Rewriting using expt_plus_aux, matching in *,
 Applying rem_def2
 Instantiating the top quantifier in -1 with the terms: $\text{expt}(2, i' + j'), n', 0$,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 Using lemma divides_expt_gt,
 Simplifying, rewriting, and recording with decision procedures,
 This completes the proof of aligned_cut_bits.
 Q.E.D.

C.17.7 Bits.aligned_add_cut_bits

Terse proof for aligned_add_cut_bits.

aligned_add_cut_bits:

$$\frac{}{\{1\} \quad \forall (n, m, i, j, k: \text{nat}): i + j \leq k \wedge \text{aligned?}(k)(n) \supset \text{cut_bits}(n + m, i, j) = \text{cut_bits}(m, i, j)}$$

Repeatedly Skolemizing and flattening,
 Applying aligned_cut_bits
 Instantiating the top quantifier in -1 with the terms: $(n!1 i!1 j!1 k!1)$,
 Simplifying, rewriting, and recording with decision procedures,
 Expanding the definition of cut_bits,
 Using lemma ndiv_plus_2,

C Proof scripts

Splitting conjunctions,

we get 2 subgoals:

`aligned_add_cut_bits.1`:

{-1}	$\text{ndiv}(m' + n', \text{expt}(2, i')) =$
	$\text{ndiv}(m', \text{expt}(2, i')) + \text{ndiv}(n', \text{expt}(2, i'))$
{-2}	$\text{rem}(\text{expt}(2, j'))(\text{ndiv}(n', \text{expt}(2, i'))) = 0$
{-3}	$n' \geq 0$
{-4}	$m' \geq 0$
{-5}	$i' \geq 0$
{-6}	$j' \geq 0$
{-7}	$k' \geq 0$
{-8}	$i' + j' \leq k'$
{-9}	$\text{aligned?}(k')(n')$
{1}	
	$\text{rem}(\text{expt}(2, j'))(\text{ndiv}(m' + n', \text{expt}(2, i'))) =$
	$\text{rem}(\text{expt}(2, j'))(\text{ndiv}(m', \text{expt}(2, i')))$

Replacing using formula -1,

Rewriting using `rem_sum2`, matching in `*`,

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `aligned_add_cut_bits.1`.

`aligned_add_cut_bits.2`:

{-1}	$\text{rem}(\text{expt}(2, j'))(\text{ndiv}(n', \text{expt}(2, i'))) = 0$
{-2}	$n' \geq 0$
{-3}	$m' \geq 0$
{-4}	$i' \geq 0$
{-5}	$j' \geq 0$
{-6}	$k' \geq 0$
{-7}	$i' + j' \leq k'$
{-8}	$\text{aligned?}(k')(n')$
{1}	
	$\text{divides}(\text{expt}(2, i'), n')$
{2}	$\text{rem}(\text{expt}(2, j'))(\text{ndiv}(m' + n', \text{expt}(2, i'))) =$
	$\text{rem}(\text{expt}(2, j'))(\text{ndiv}(m', \text{expt}(2, i')))$

Hiding formulas: 2,

Expanding the definition of `aligned?`,

Rewriting using `divides_transitive`, matching in `*` where `n` gets `expt(2, i!1)`, `m` gets `expt(2, k!1)`, `l` gets `n!1`,

Hiding formulas: 2,

Expanding the definition of `divides`,

Instantiating the top quantifier in 1 with the terms: `expt(2, k' - i')`,

Keeping 1 and hiding `*`,

Rewriting using `expt_plus_aux`, matching in `*`,

This completes the proof of `aligned_add_cut_bits.2`.

Q.E.D.

C.17.8 Bits.cut_bit_bits

Terse proof for `cut_bit_bits`.

`cut_bit_bits`:

{1}	$\forall (i, n: \text{nat}): \text{cut_bit}(n, i) = \text{IF } \text{cut_bits}(n, i, 1) = 1 \text{ THEN TRUE ELSE FALSE EN-}$
	DIF

Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of `cut_bit_bits`.
 Q.E.D.

C.17.9 Bits.cut_bits_zero

Terse proof for `cut_bits_zero`.

`cut_bits_zero`:

$$\frac{}{\{1\} \quad \forall (i, k: \text{nat}): \text{cut_bits}(0, i, k) = 0}$$

Repeatedly Skolemizing and flattening,
 Expanding the definition of `cut_bits`,
 Installing automatic rewrites from: `ndiv_0 rem_zero`
 Simplifying, rewriting, and recording with decision procedures,
 This completes the proof of `cut_bits_zero`.
 Q.E.D.

C.17.10 Bits.zero_cut_bits

Terse proof for `zero_cut_bits`.

`zero_cut_bits`:

$$\frac{}{\{1\} \quad \forall (n, i, k: \text{nat}): k = 0 \supset \text{cut_bits}(n, i, k) = 0}$$

Repeatedly Skolemizing and flattening,
 Replacing using formula -4,
 Expanding the definition of `cut_bits`,
 Rewriting using `expt_x0_aux`, matching in *,
 Simplifying, rewriting, and recording with decision procedures,
 This completes the proof of `zero_cut_bits`.
 Q.E.D.

C.17.11 Bits.cut_bits_cut_bits_TCC1

Terse proof for `cut_bits_cut_bits_TCC1`.

`cut_bits_cut_bits_TCC1`:

$$\frac{}{\{1\} \quad \forall (k_1, i_2, k_2: \text{nat}): i_2 + k_2 > k_1 \wedge \neg i_2 \geq k_1 \supset k_1 - i_2 \geq 0}$$

Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of `cut_bits_cut_bits_TCC1`.
 Q.E.D.

C.17.12 Bits.cut_bits_cut_bits

Terse proof for `cut_bits_cut_bits`.

C Proof scripts

cut_bits_cut_bits:

$\{1\}$	$\forall (n, i_1, k_1, i_2, k_2: \text{nat}):$ $\text{cut_bits}(\text{cut_bits}(n, i_1, k_1), i_2, k_2) =$ IF $i_2 \geq k_1$ THEN 0 ELSIF $i_2 + k_2 > k_1$ THEN $\text{cut_bits}(n, i_1 + i_2, k_1 - i_2)$ ELSE $\text{cut_bits}(n, i_1 + i_2, k_2)$ ENDIF
---------	---

Repeatedly Skolemizing and flattening,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 3 subgoals:

cut_bits_cut_bits.1:

$\{-1\}$	$n' \geq 0$
$\{-2\}$	$i'_1 \geq 0$
$\{-3\}$	$k'_1 \geq 0$
$\{-4\}$	$i'_2 \geq 0$
$\{-5\}$	$k'_2 \geq 0$
$\{-6\}$	$i'_2 \geq k'_1$
$\{1\}$	$\text{cut_bits}(\text{cut_bits}(n', i'_1, k'_1), i'_2, k'_2) = 0$

Expanding the definition of cut_bits,

Applying ndiv_plus_mod

Installing automatic rewrites from: divides_zero ndiv_0 rem_zero

Instantiating the top quantifier in -1 with the terms: 0, $\text{expt}(2, i'_2)$, $\text{rem}(\text{expt}(2, k'_1))(\text{ndiv}(n', \text{expt}(2, i'_1)))$,

we get 2 subgoals:

cut_bits_cut_bits.1.1:

$\{-1\}$	$\text{divides}(\text{expt}(2, i'_2), 0) \supset$ $\text{ndiv}(0 + \text{rem}(\text{expt}(2, k'_1))(\text{ndiv}(n', \text{expt}(2, i'_1))), \text{expt}(2, i'_2)) =$ $\text{ndiv}(0, \text{expt}(2, i'_2))$
$\{-2\}$	$n' \geq 0$
$\{-3\}$	$i'_1 \geq 0$
$\{-4\}$	$k'_1 \geq 0$
$\{-5\}$	$i'_2 \geq 0$
$\{-6\}$	$k'_2 \geq 0$
$\{-7\}$	$i'_2 \geq k'_1$
$\{1\}$	$\text{rem}(\text{expt}(2, k'_2))$ $(\text{ndiv}(\text{rem}(\text{expt}(2, k'_1))(\text{ndiv}(n', \text{expt}(2, i'_1))), \text{expt}(2, i'_2)))$ $= 0$

Simplifying, rewriting, and recording with decision procedures,

Replacing using formula -1,

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of cut_bits_cut_bits.1.1.

cut_bits_cut_bits.1.2:

{-1}	$n' \geq 0$
{-2}	$i'_1 \geq 0$
{-3}	$k'_1 \geq 0$
{-4}	$i'_2 \geq 0$
{-5}	$k'_2 \geq 0$
{-6}	$i'_2 \geq k'_1$
{1}	
	$\text{rem}(\text{expt}(2, k'_1))(\text{ndiv}(n', \text{expt}(2, i'_1))) < \text{expt}(2, i'_2)$
{2}	$\text{rem}(\text{expt}(2, k'_2))$ $(\text{ndiv}(\text{rem}(\text{expt}(2, k'_1))(\text{ndiv}(n', \text{expt}(2, i'_1))), \text{expt}(2, i'_2)))$ $= 0$

Applying both_sides_expt_gt1_le

Instantiating the top quantifier in -1 with the terms: 2, k'_1 , i'_2 ,

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of cut_bits_cut_bits.1.2.

cut_bits_cut_bits.2:

{-1}	$n' \geq 0$
{-2}	$i'_1 \geq 0$
{-3}	$k'_1 \geq 0$
{-4}	$i'_2 \geq 0$
{-5}	$k'_2 \geq 0$
{-6}	$i'_2 + k'_2 > k'_1$
{1}	
	$i'_2 \geq k'_1$
{2}	$\text{cut_bits}(\text{cut_bits}(n', i'_1, k'_1), i'_2, k'_2) =$ $\text{cut_bits}(n', i'_1 + i'_2, k'_1 - i'_2)$

Installing automatic rewrites from: cut_bits rem_ndiv expt_plus

Simplifying, rewriting, and recording with decision procedures,

Rewriting using ndiv_times_2, matching in *,

Applying rem_mod

Instantiating the top quantifier in -1 with the terms: $\text{expt}(2, i'_1 + i'_2 + k'_2)$, $\text{rem}(\text{expt}(2, i'_1 + k'_1))(n')$,

we get 2 subgoals:

cut_bits_cut_bits.2.1:

{-1}	$\text{rem}(\text{expt}(2, i'_1 + i'_2 + k'_2))(\text{rem}(\text{expt}(2, i'_1 + k'_1))(n')) =$ $\text{rem}(\text{expt}(2, i'_1 + k'_1))(n')$
{-2}	$n' \geq 0$
{-3}	$i'_1 \geq 0$
{-4}	$k'_1 \geq 0$
{-5}	$i'_2 \geq 0$
{-6}	$k'_2 \geq 0$
{-7}	$i'_2 + k'_2 > k'_1$
{1}	
	$i'_2 \geq k'_1$
{2}	$\text{ndiv}(\text{rem}(\text{expt}(2, i'_1 + i'_2 + k'_2))$ $(\text{rem}(\text{expt}(2, i'_1 + k'_1))(n')),$ $\text{expt}(2, i'_1 + i'_2))$ $= \text{ndiv}(\text{rem}(\text{expt}(2, i'_1 + k'_1))(n'), \text{expt}(2, i'_1 + i'_2))$

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of cut_bits_cut_bits.2.1.

cut_bits_cut_bits.2.2:

{-1}	$n' \geq 0$
{-2}	$i'_1 \geq 0$
{-3}	$k'_1 \geq 0$
{-4}	$i'_2 \geq 0$
{-5}	$k'_2 \geq 0$
{-6}	$i'_2 + k'_2 > k'_1$
{1}	$\text{rem}(\text{expt}(2, i'_1 + k'_1))(n') < \text{expt}(2, i'_1 + i'_2 + k'_2)$
{2}	$i'_2 \geq k'_1$
{3}	$\text{ndiv}(\text{rem}(\text{expt}(2, i'_1 + i'_2 + k'_2))$ $\quad (\text{rem}(\text{expt}(2, i'_1 + k'_1))(n')),$ $\quad \text{expt}(2, i'_1 + i'_2))$ $\quad = \text{ndiv}(\text{rem}(\text{expt}(2, i'_1 + k'_1))(n'), \text{expt}(2, i'_1 + i'_2))$

Hiding formulas: 3,

Applying both_sides_expt_gt1_le

Instantiating the top quantifier in -1 with the terms: 2, $i'_1 + k'_1$, $i'_1 + i'_2 + k'_2$,

Simplifying, rewriting, and recording with decision procedures,

Adding type constraints for $\text{rem}(\text{expt}(2, i1!1 + k1!1))(n!1)$,

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of cut_bits_cut_bits.2.2.

cut_bits_cut_bits.3:

{-1}	$n' \geq 0$
{-2}	$i'_1 \geq 0$
{-3}	$k'_1 \geq 0$
{-4}	$i'_2 \geq 0$
{-5}	$k'_2 \geq 0$
{1}	$i'_2 \geq k'_1$
{2}	$i'_2 + k'_2 > k'_1$
{3}	$\text{cut_bits}(\text{cut_bits}(n', i'_1, k'_1), i'_2, k'_2) = \text{cut_bits}(n', i'_1 + i'_2, k'_2)$

Installing automatic rewrites from: cut_bits rem_ndiv expt_plus

Simplifying, rewriting, and recording with decision procedures,

Rewriting using ndiv_times_2, matching in *,

Rewriting using rem_rem, matching in *,

Hiding formulas: 4,

Rewriting using expt_divides, matching in *,

This completes the proof of cut_bits_cut_bits.3.

Q.E.D.

C.17.13 Bits.cut_bit_cut_bits

Terse proof for cut_bit_cut_bits.

cut_bit_cut_bits:

{1}	$\forall (n, i_1, k_1, i_2: \text{nat}):$ $\text{cut_bit}(\text{cut_bits}(n, i_1, k_1), i_2) =$ $\text{IF } i_2 \geq k_1 \text{ THEN FALSE ELSE } \text{cut_bit}(n, i_1 + i_2) \text{ ENDIF}$
-----	--

Repeatedly Skolemizing and flattening,

Rewriting using cut_bit_bits, matching in *,

Rewriting using cut_bit_bits, matching in *,

Rewriting using `cut_bits_cut_bits`, matching in `*`,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 This completes the proof of `cut_bit_cut_bits`.
 Q.E.D.

C.17.14 Bits.shift_bits_left_less_TCC1

Terse proof for `shift_bits_left_less_TCC1`.
`shift_bits_left_less_TCC1`:

$$\frac{}{\{1\} \quad \forall (i, k: \text{nat}): k \geq i \supset k - i \geq 0}$$

Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of `shift_bits_left_less_TCC1`.
 Q.E.D.

C.17.15 Bits.shift_bits_left_less

Terse proof for `shift_bits_left_less`.
`shift_bits_left_less`:

$$\frac{}{\{1\} \quad \forall (n, i, k: \text{nat}): k \geq i \wedge n < \text{expt}(2, k - i) \supset \text{shift_bits_left}(n, i) < \text{expt}(2, k)}$$

Repeatedly Skolemizing and flattening,
 Expanding the definition of `shift_bits_left`,
 Applying `both_sides_times_pos_lt1`
 Instantiating the top quantifier in -1 with the terms: `expt(2, i')`, `n'`, `expt(2, k' - i')`,
 we get 2 subgoals:
`shift_bits_left_less.1`:

$$\frac{\begin{array}{l} \{-1\} \quad n' \times \text{expt}(2, i') < \text{expt}(2, k' - i') \times \text{expt}(2, i') \equiv \\ \quad n' < \text{expt}(2, k' - i') \\ \{-2\} \quad n' \geq 0 \\ \{-3\} \quad i' \geq 0 \\ \{-4\} \quad k' \geq 0 \\ \{-5\} \quad k' \geq i' \\ \{-6\} \quad n' < \text{expt}(2, k' - i') \end{array}}{\{1\} \quad n' \times \text{expt}(2, i') < \text{expt}(2, k')}$$

Rewriting using `expt_plus`, matching in `*`,
 Simplifying, rewriting, and recording with decision procedures,
 This completes the proof of `shift_bits_left_less.1`.
`shift_bits_left_less.2`:

$$\frac{\begin{array}{l} \{-1\} \quad n' \geq 0 \\ \{-2\} \quad i' \geq 0 \\ \{-3\} \quad k' \geq 0 \\ \{-4\} \quad k' \geq i' \\ \{-5\} \quad n' < \text{expt}(2, k' - i') \end{array}}{\begin{array}{l} \{1\} \quad k' - i' \geq 0 \\ \{2\} \quad n' \times \text{expt}(2, i') < \text{expt}(2, k') \end{array}}$$

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `shift_bits_left_less.2`.
Q.E.D.

C.17.16 Bits.aligned_shift_bits_ok

Terse proof for `aligned_shift_bits_ok`.

`aligned_shift_bits_ok`:

{1} $\forall (n, i, j: \text{nat}): i \leq j \supset \text{aligned?}(i)(\text{shift_bits_left}(n, j))$

Repeatedly Skolemizing and flattening,
Expanding the definition of `aligned?`,
Expanding the definition of `shift_bits_left`,
Rewriting using `divides_prod2`, matching in *,
Rewriting using `expt_divides`, matching in *,
This completes the proof of `aligned_shift_bits_ok`.
Q.E.D.

C.17.17 Bits.aligned_shift_bits_reduce_TCC1

Terse proof for `aligned_shift_bits_reduce_TCC1`.

`aligned_shift_bits_reduce_TCC1`:

{1} $\forall (i, j: \text{nat}): i > j \supset i - j \geq 0$

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `aligned_shift_bits_reduce_TCC1`.
Q.E.D.

C.17.18 Bits.aligned_shift_bits_reduce

Terse proof for `aligned_shift_bits_reduce`.

`aligned_shift_bits_reduce`:

{1} $\forall (n, i, j: \text{nat}):$
 $i > j \wedge \text{aligned?}(i - j)(n) \supset \text{aligned?}(i)(\text{shift_bits_left}(n, j))$

Repeatedly Skolemizing and flattening,
Expanding the definition of `aligned?`,
Expanding the definition of `shift_bits_left`,
Applying `expt_plus`
Instantiating the top quantifier in -1 with the terms: $2, i' - j', j'$,
we get 2 subgoals:

`aligned_shift_bits_reduce.1`:

{-1} $\text{expt}(2, i' - j') \times \text{expt}(2, j') = \text{expt}(2, i' - j' + j')$
{-2} $n' \geq 0$
{-3} $i' \geq 0$
{-4} $j' \geq 0$
{-5} $i' > j'$
{-6} $\text{divides}(\text{expt}(2, i' - j'), n')$
{1} $\text{divides}(\text{expt}(2, i'), n' \times \text{expt}(2, j'))$

Simplifying, rewriting, and recording with decision procedures,
 Replacing using formula -1,
 Using lemma divides_prod_elim2,
 Simplifying, rewriting, and recording with decision procedures,
 This completes the proof of aligned_shift_bits_reduce.1.
 aligned_shift_bits_reduce.2:

{-1}	$n' \geq 0$
{-2}	$i' \geq 0$
{-3}	$j' \geq 0$
{-4}	$i' > j'$
{-5}	$\text{divides}(\text{expt}(2, i' - j'), n')$
{1}	$i' - j' \geq 0$
{2}	$\text{divides}(\text{expt}(2, i'), n' \times \text{expt}(2, j'))$

Simplifying, rewriting, and recording with decision procedures,
 This completes the proof of aligned_shift_bits_reduce.2.
 Q.E.D.

C.17.19 Bits.overwrite_bits_TCC1

Terse proof for overwrite_bits_TCC1.

overwrite_bits_TCC1:

{1}	$\forall (n, m, i, k: \text{nat}):$ $n - \text{rem}(\text{expt}(2, i + k))(n) + \text{shift_bits_left}(\text{cut_bits}(m, i, k), i) + \text{rem}(\text{expt}(2, i))(n)$ ≥ 0
-----	---

Repeatedly Skolemizing and flattening,
 Using lemma rem_def_pos,
 Simplifying, rewriting, and recording with decision procedures,
 Repeatedly Skolemizing and flattening,
 Simplifying, rewriting, and recording with decision procedures,
 This completes the proof of overwrite_bits_TCC1.
 Q.E.D.

C.17.20 Bits.overwrite_bits_less

Terse proof for overwrite_bits_less.

overwrite_bits_less:

{1}	$\forall (n, m, i, k, j: \text{nat}):$ $n < \text{expt}(2, j) \wedge i + k \leq j \supset \text{overwrite_bits}(n, m, i, k) < \text{expt}(2, j)$
-----	--

Repeatedly Skolemizing and flattening,
 Expanding the definition of overwrite_bits,
 Using lemma rem_def_pos,
 Simplifying, rewriting, and recording with decision procedures,
 Repeatedly Skolemizing and flattening,
 Applying bit_split_less
 Instantiating the top quantifier in -1 with the terms: $q', \text{rem}(\text{expt}(2, i'))(n') + \text{shift_bits_left}(\text{cut_bits}(m', i', k'), i')$,
 $j' - i' - k', i' + k'$,
 Simplifying, rewriting, and recording with decision procedures,

C Proof scripts

Splitting conjunctions,

we get 2 subgoals:

`overwrite_bits_less.1:`

<pre>{-1} q' ≥ 0 {-2} n' = rem(expt(2, i' + k'))(n') + expt(2, i' + k') × q' {-3} n' ≥ 0 {-4} m' ≥ 0 {-5} i' ≥ 0 {-6} k' ≥ 0 {-7} j' ≥ 0 {-8} n' < expt(2, j') {-9} i' + k' ≤ j'</pre>	<hr style="border: 0.5px solid black;"/> <pre>{1} q' < expt(2, -1 × i' - k' + j') {2} rem(expt(2, i'))(n') + shift_bits_left(cut_bits(m', i', k'), i') - rem(expt(2, i' + k'))(n') + n' < expt(2, j')</pre>
--	---

Hiding formulas: 2,

Applying `both_sides_times_pos_ge2`

Instantiating the top quantifier in -1 with the terms: `expt(2, i' + k')`, `q'`, `expt(2, -1 × i' - k' + j')`,

Rewriting using `expt_plus`, matching in `*`,

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `overwrite_bits_less.1`.

`overwrite_bits_less.2:`

<pre>{-1} q' ≥ 0 {-2} n' = rem(expt(2, i' + k'))(n') + expt(2, i' + k') × q' {-3} n' ≥ 0 {-4} m' ≥ 0 {-5} i' ≥ 0 {-6} k' ≥ 0 {-7} j' ≥ 0 {-8} n' < expt(2, j') {-9} i' + k' ≤ j'</pre>	<hr style="border: 0.5px solid black;"/> <pre>{1} rem(expt(2, i'))(n') + shift_bits_left(cut_bits(m', i', k'), i') < expt(2, i' + k') {2} rem(expt(2, i'))(n') + shift_bits_left(cut_bits(m', i', k'), i') - rem(expt(2, i' + k'))(n') + n' < expt(2, j')</pre>
--	---

Hiding formulas: 2,

Expanding the definition of `shift_bits_left`,

Applying `bit_split_less`

Instantiating the top quantifier in -1 with the terms: `cut_bits(m', i', k')`, `rem(expt(2, i'))(n')`, `k'`, `i'`,

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `overwrite_bits_less.2`.

Q.E.D.

C.17.21 Bits.overwrite_bit_less

Terse proof for `overwrite_bit_less`.

`overwrite_bit_less:`

<pre>{1} ∀ (n, m, i, k: nat):</pre>	<hr style="border: 0.5px solid black;"/> <pre> n < expt(2, k) ∧ i < k ⊃ overwrite_bit(n, m, i) < expt(2, k)</pre>
--------------------------------------	---

Repeatedly Skolemizing and flattening,
 Expanding the definition of `overwrite_bit`,
 Rewriting using `overwrite_bits_less`, matching in *,
 This completes the proof of `overwrite_bit_less`.
 Q.E.D.

C.17.22 Bits.overwrite_bool_bit_less

Terse proof for `overwrite_bool_bit_less`.

`overwrite_bool_bit_less`:

$$\frac{\{1\} \quad \forall (n: \text{nat}, b: \text{bool}, i, k: \text{nat}):}{n < \text{expt}(2, k) \wedge i < k \supset \text{overwrite_bool_bit}(n, b, i) < \text{expt}(2, k)}$$

Repeatedly Skolemizing and flattening,
 Expanding the definition of `overwrite_bool_bit`,
 Rewriting using `overwrite_bit_less`, matching in *,
 This completes the proof of `overwrite_bool_bit_less`.
 Q.E.D.

C.17.23 Bits.overwrite_shift_bits_less

Terse proof for `overwrite_shift_bits_less`.

`overwrite_shift_bits_less`:

$$\frac{\{1\} \quad \forall (n, m, i, k, j: \text{nat}):}{n < \text{expt}(2, j) \wedge i + k \leq j \supset \text{overwrite_shift_bits}(n, m, i, k) < \text{expt}(2, j)}$$

Repeatedly Skolemizing and flattening,
 Expanding the definition of `overwrite_shift_bits`,
 Rewriting using `overwrite_bits_less`, matching in *,
 This completes the proof of `overwrite_shift_bits_less`.
 Q.E.D.

C.17.24 Bits.aligned_overwrite_shift_bits_more

Terse proof for `aligned_overwrite_shift_bits_more`.

`aligned_overwrite_shift_bits_more`:

$$\frac{\{1\} \quad \forall (a, n, m, i: \text{nat}, k: \text{posnat}):}{a \leq i \wedge \text{aligned?}(a)(n) \supset \text{aligned?}(a)(\text{overwrite_shift_bits}(n, m, i, k))}$$

Repeatedly Skolemizing and flattening,
 Expanding the definition of `overwrite_shift_bits`,
 Expanding the definition of `overwrite_bits`,
 Applying `aligned_plus_n`
 Instantiating the top quantifier in -1 with the terms: a' , $n' - \text{rem}(\text{expt}(2, i' + k'))(n')$,
 $\text{rem}(\text{expt}(2, i'))(n') + \text{shift_bits_left}(\text{cut_bits}(\text{shift_bits_left}(m', i'), i', k'), i')$,
 we get 3 subgoals:

C Proof scripts

`aligned_overwrite_shift_bits_more.1:`

{-1}	$\text{aligned?}(a')$	
{-2}	$a' \geq 0$	
{-3}	$n' \geq 0$	
{-4}	$m' \geq 0$	
{-5}	$i' \geq 0$	
{-6}	$k' > 0$	
{-7}	$a' \leq i'$	
{-8}	$\text{aligned?}(a')(n')$	
{1}	$\text{aligned?}(a')$	$(\text{rem}(\text{expt}(2, i' + k'))(n') + \text{rem}(\text{expt}(2, i'))(n') + \text{shift_bits_left}(\text{cut_bits}(\text{shift_bits_left}(m', i'), i', k'), i') - \text{rem}(\text{expt}(2, i'))(n') + \text{shift_bits_left}(\text{cut_bits}(\text{shift_bits_left}(m', i'), i', k'), i') - \text{rem}(\text{expt}(2, i'))(n'))$

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `aligned_overwrite_shift_bits_more.1`.

`aligned_overwrite_shift_bits_more.2:`

{-1}	$a' \geq 0$	
{-2}	$n' \geq 0$	
{-3}	$m' \geq 0$	
{-4}	$i' \geq 0$	
{-5}	$k' > 0$	
{-6}	$a' \leq i'$	
{-7}	$\text{aligned?}(a')(n')$	
{1}	$\text{aligned?}(a')$	$(\text{rem}(\text{expt}(2, i'))(n') + \text{shift_bits_left}(\text{cut_bits}(\text{shift_bits_left}(m', i'), i', k'), i'))$
{2}	$\text{aligned?}(a')$	$(\text{rem}(\text{expt}(2, i'))(n') + \text{shift_bits_left}(\text{cut_bits}(\text{shift_bits_left}(m', i'), i', k'), i') - \text{rem}(\text{expt}(2, i'))(n'))$

Hiding formulas: 2,

Rewriting using `aligned_plus_n`, matching in `*`,

we get 2 subgoals:

`aligned_overwrite_shift_bits_more.2.1:`

{-1}	$a' \geq 0$	
{-2}	$n' \geq 0$	
{-3}	$m' \geq 0$	
{-4}	$i' \geq 0$	
{-5}	$k' > 0$	
{-6}	$a' \leq i'$	
{-7}	$\text{aligned?}(a')(n')$	
{1}	$\text{aligned?}(a')(\text{shift_bits_left}(\text{cut_bits}(\text{shift_bits_left}(m', i'), i', k'), i'))$	
{2}	$\text{aligned?}(a')$	$(\text{rem}(\text{expt}(2, i'))(n') + \text{shift_bits_left}(\text{cut_bits}(\text{shift_bits_left}(m', i'), i', k'), i'))$

Hiding formulas: 2,

Rewriting using `aligned_shift_bits_ok`, matching in `*`,

This completes the proof of `aligned_overwrite_shift_bits_more.2.1`.

aligned_overwrite_shift_bits_more.2.2:

{-1}	$a' \geq 0$
{-2}	$n' \geq 0$
{-3}	$m' \geq 0$
{-4}	$i' \geq 0$
{-5}	$k' > 0$
{-6}	$a' \leq i'$
{-7}	$\text{aligned?}(a')(n')$
{1}	$\text{aligned?}(a')(\text{rem}(\text{expt}(2, i'))(n'))$
{2}	$\text{aligned?}(a')(\text{rem}(\text{expt}(2, i'))(n') + \text{shift_bits_left}(\text{cut_bits}(\text{shift_bits_left}(m', i'), i', k'), i'))$

Rewriting using aligned_rem, matching in *,

This completes the proof of aligned_overwrite_shift_bits_more.2.2.

aligned_overwrite_shift_bits_more.3:

{-1}	$a' \geq 0$
{-2}	$n' \geq 0$
{-3}	$m' \geq 0$
{-4}	$i' \geq 0$
{-5}	$k' > 0$
{-6}	$a' \leq i'$
{-7}	$\text{aligned?}(a')(n')$
{1}	$n' - \text{rem}(\text{expt}(2, i' + k'))(n') \geq 0 \wedge \text{aligned?}(a')(n' - \text{rem}(\text{expt}(2, i' + k'))(n'))$
{2}	$\text{aligned?}(a')(\text{rem}(\text{expt}(2, i'))(n') + \text{shift_bits_left}(\text{cut_bits}(\text{shift_bits_left}(m', i'), i', k'), i') - \text{rem}(\text{expt}(2, i' + k'))(n'))$

Hiding formulas: 2,

Applying propositional simplification,

we get 2 subgoals:

aligned_overwrite_shift_bits_more.3.1:

{-1}	$a' \geq 0$
{-2}	$n' \geq 0$
{-3}	$m' \geq 0$
{-4}	$i' \geq 0$
{-5}	$k' > 0$
{-6}	$a' \leq i'$
{-7}	$\text{aligned?}(a')(n')$
{1}	$n' - \text{rem}(\text{expt}(2, i' + k'))(n') \geq 0$

Using lemma rem_def_pos,

Simplifying, rewriting, and recording with decision procedures,

Repeatedly Skolemizing and flattening,

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of aligned_overwrite_shift_bits_more.3.1.

C Proof scripts

`aligned_overwrite_shift_bits_more.3.2:`

{-1}	$a' \geq 0$
{-2}	$n' \geq 0$
{-3}	$m' \geq 0$
{-4}	$i' \geq 0$
{-5}	$k' > 0$
{-6}	$a' \leq i'$
{-7}	$\text{aligned?}(a')(n')$
{1}	$\text{aligned?}(a')(n' - \text{rem}(\text{expt}(2, i' + k'))(n'))$

Applying `aligned_minus`

Instantiating the top quantifier in -1 with the terms: $a', a', n', \text{rem}(\text{expt}(2, i' + k'))(n')$, we get 2 subgoals:

`aligned_overwrite_shift_bits_more.3.2.1:`

{-1}	$n' \geq \text{rem}(\text{expt}(2, i' + k'))(n') \supset$ $\text{aligned?}(\min(a', a'))(n' - \text{rem}(\text{expt}(2, i' + k'))(n'))$
{-2}	$a' \geq 0$
{-3}	$n' \geq 0$
{-4}	$m' \geq 0$
{-5}	$i' \geq 0$
{-6}	$k' > 0$
{-7}	$a' \leq i'$
{-8}	$\text{aligned?}(a')(n')$
{1}	$\text{aligned?}(a')(n' - \text{rem}(\text{expt}(2, i' + k'))(n'))$

Expanding the definition of `min`,

Simplifying, rewriting, and recording with decision procedures,

Using lemma `rem_def_pos`,

Simplifying, rewriting, and recording with decision procedures,

Repeatedly Skolemizing and flattening,

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `aligned_overwrite_shift_bits_more.3.2.1`.

`aligned_overwrite_shift_bits_more.3.2.2:`

{-1}	$a' \geq 0$
{-2}	$n' \geq 0$
{-3}	$m' \geq 0$
{-4}	$i' \geq 0$
{-5}	$k' > 0$
{-6}	$a' \leq i'$
{-7}	$\text{aligned?}(a')(n')$
{1}	$\text{aligned?}(a')(\text{rem}(\text{expt}(2, i' + k'))(n'))$
{2}	$\text{aligned?}(a')(n' - \text{rem}(\text{expt}(2, i' + k'))(n'))$

Rewriting using `aligned_rem`, matching in `*`,

This completes the proof of `aligned_overwrite_shift_bits_more.3.2.2`.

Q.E.D.

C.17.25 `Bits.aligned_overwrite_shift_bits_less_TCC1`

Terse proof for `aligned_overwrite_shift_bits_less_TCC1`.

aligned_overwrite_shift_bits_less_TCC1:

$$\frac{}{\{1\} \quad \forall (a, n, i: \text{nat}): \text{aligned?}(a)(n) \wedge i < a \supset a - i \geq 0}$$

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of aligned_overwrite_shift_bits_less_TCC1.

Q.E.D.

C.17.26 Bits.aligned_overwrite_shift_bits_less

Terse proof for aligned_overwrite_shift_bits_less.

aligned_overwrite_shift_bits_less:

$$\frac{}{\{1\} \quad \forall (a, n, m, i: \text{nat}, k: \text{posnat}): \\ i < a \wedge \text{aligned?}(a)(n) \wedge \text{aligned?}(a-i)(m) \supset \\ \text{aligned?}(a)(\text{overwrite_shift_bits}(n, m, i, k))}$$

Installing automatic rewrites from: aligned_shift_bits_reduce ndiv_times_divident_2 ndiv_self aligned_rem divides_reflexive

Repeatedly Skolemizing and flattening,

Expanding the definition of overwrite_shift_bits,

Expanding the definition of overwrite_bits,

Applying aligned_plus_n

Instantiating the top quantifier in -1 with the terms: a' , $n' - \text{rem}(\text{expt}(2, i' + k'))(n')$, $\text{rem}(\text{expt}(2, i'))(n') + \text{shift_bits_left}(\text{cut_bits}(\text{shift_bits_left}(m', i'), i', k'), i')$,

we get 3 subgoals:

aligned_overwrite_shift_bits_less.1:

$$\frac{\begin{array}{l} \{-1\} \quad \text{aligned?}(a') \\ \quad \quad \quad (n' - \text{rem}(\text{expt}(2, i' + k'))(n') + \text{rem}(\text{expt}(2, i'))(n') + \text{shift_bits_left}(\text{cut_bits}(\text{shift_bits_left}(m', i'), \\ \{-2\} \quad a' \geq 0 \\ \{-3\} \quad n' \geq 0 \\ \{-4\} \quad m' \geq 0 \\ \{-5\} \quad i' \geq 0 \\ \{-6\} \quad k' > 0 \\ \{-7\} \quad i' < a' \\ \{-8\} \quad \text{aligned?}(a')(n') \\ \{-9\} \quad \text{aligned?}(a' - i')(m') \end{array}}{\{1\} \quad \text{aligned?}(a') \\ \quad \quad \quad (\text{rem}(\text{expt}(2, i'))(n') + \text{shift_bits_left}(\text{cut_bits}(\text{shift_bits_left}(m', i'), i', k'), i') - \text{rem}(\text{expt}(2, i' + k'))(n'))}$$

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of aligned_overwrite_shift_bits_less.1.

C Proof scripts

`aligned_overwrite_shift_bits_less.2:`

{-1}	$a' \geq 0$
{-2}	$n' \geq 0$
{-3}	$m' \geq 0$
{-4}	$i' \geq 0$
{-5}	$k' > 0$
{-6}	$i' < a'$
{-7}	$\text{aligned?}(a')(n')$
{-8}	$\text{aligned?}(a' - i')(m')$
{1}	$\text{aligned?}(a')$ $(\text{rem}(\text{expt}(2, i'))(n') + \text{shift_bits_left}(\text{cut_bits}(\text{shift_bits_left}(m', i'), i', k'), i'))$
{2}	$\text{aligned?}(a')$ $(\text{rem}(\text{expt}(2, i'))(n') + \text{shift_bits_left}(\text{cut_bits}(\text{shift_bits_left}(m', i'), i', k'), i') - \text{rem}(\text{expt}(2, i')(m'), i'))$

Hiding formulas: 2,

Rewriting using `aligned_plus_n`, matching in *,

we get 2 subgoals:

`aligned_overwrite_shift_bits_less.2.1:`

{-1}	$a' \geq 0$
{-2}	$n' \geq 0$
{-3}	$m' \geq 0$
{-4}	$i' \geq 0$
{-5}	$k' > 0$
{-6}	$i' < a'$
{-7}	$\text{aligned?}(a')(n')$
{-8}	$\text{aligned?}(a' - i')(m')$
{1}	$\text{aligned?}(a')(\text{shift_bits_left}(\text{cut_bits}(\text{shift_bits_left}(m', i'), i', k'), i'))$
{2}	$\text{aligned?}(a')$ $(\text{rem}(\text{expt}(2, i'))(n') + \text{shift_bits_left}(\text{cut_bits}(\text{shift_bits_left}(m', i'), i', k'), i'))$

Hiding formulas: 2,

Rewriting using `aligned_shift_bits_reduce`, matching in *,

Hiding formulas: 2,

Expanding the definition of `cut_bits`,

Expanding the definition of `shift_bits_left`,

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `aligned_overwrite_shift_bits_less.2.1`.

`aligned_overwrite_shift_bits_less.2.2:`

{-1}	$a' \geq 0$
{-2}	$n' \geq 0$
{-3}	$m' \geq 0$
{-4}	$i' \geq 0$
{-5}	$k' > 0$
{-6}	$i' < a'$
{-7}	$\text{aligned?}(a')(n')$
{-8}	$\text{aligned?}(a' - i')(m')$
{1}	$\text{aligned?}(a')(\text{rem}(\text{expt}(2, i'))(n'))$
{2}	$\text{aligned?}(a')$ $(\text{rem}(\text{expt}(2, i'))(n') + \text{shift_bits_left}(\text{cut_bits}(\text{shift_bits_left}(m', i'), i', k'), i'))$

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `aligned_overwrite_shift_bits_less.2.2`.

aligned_overwrite_shift_bits_less.3:

{-1}	$a' \geq 0$
{-2}	$n' \geq 0$
{-3}	$m' \geq 0$
{-4}	$i' \geq 0$
{-5}	$k' > 0$
{-6}	$i' < a'$
{-7}	$\text{aligned?}(a')(n')$
{-8}	$\text{aligned?}(a' - i')(m')$
{1}	$n' - \text{rem}(\text{expt}(2, i' + k'))(n') \geq 0 \wedge$ $\text{aligned?}(a')(n' - \text{rem}(\text{expt}(2, i' + k'))(n'))$
{2}	$\text{aligned?}(a')(\text{rem}(\text{expt}(2, i'))(n') + \text{shift_bits_left}(\text{cut_bits}(\text{shift_bits_left}(m', i'), i', k'), i') - \text{rem}(\text{expt}(2, i' + k'))(n'))$

Hiding formulas: 2,

Applying propositional simplification,

we get 2 subgoals:

aligned_overwrite_shift_bits_less.3.1:

{-1}	$a' \geq 0$
{-2}	$n' \geq 0$
{-3}	$m' \geq 0$
{-4}	$i' \geq 0$
{-5}	$k' > 0$
{-6}	$i' < a'$
{-7}	$\text{aligned?}(a')(n')$
{-8}	$\text{aligned?}(a' - i')(m')$
{1}	$n' - \text{rem}(\text{expt}(2, i' + k'))(n') \geq 0$

Using lemma rem_def_pos,

Simplifying, rewriting, and recording with decision procedures,

Repeatedly Skolemizing and flattening,

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of aligned_overwrite_shift_bits_less.3.1.

aligned_overwrite_shift_bits_less.3.2:

{-1}	$a' \geq 0$
{-2}	$n' \geq 0$
{-3}	$m' \geq 0$
{-4}	$i' \geq 0$
{-5}	$k' > 0$
{-6}	$i' < a'$
{-7}	$\text{aligned?}(a')(n')$
{-8}	$\text{aligned?}(a' - i')(m')$
{1}	$\text{aligned?}(a')(n' - \text{rem}(\text{expt}(2, i' + k'))(n'))$

Applying aligned_minus

Instantiating the top quantifier in -1 with the terms: $a', a', n', \text{rem}(\text{expt}(2, i' + k'))(n')$,

we get 2 subgoals:

`aligned_overwrite_shift_bits_less.3.2.1:`

{-1}	$n' \geq \text{rem}(\text{expt}(2, i' + k'))(n') \supset$ $\text{aligned}?(\min(a', a'))(n' - \text{rem}(\text{expt}(2, i' + k'))(n'))$
{-2}	$a' \geq 0$
{-3}	$n' \geq 0$
{-4}	$m' \geq 0$
{-5}	$i' \geq 0$
{-6}	$k' > 0$
{-7}	$i' < a'$
{-8}	$\text{aligned}?(a')(n')$
{-9}	$\text{aligned}?(a' - i')(m')$
{1}	$\text{aligned}?(a')(n' - \text{rem}(\text{expt}(2, i' + k'))(n'))$

Expanding the definition of min,

Simplifying, rewriting, and recording with decision procedures,

Using lemma `rem_def_pos`,

Simplifying, rewriting, and recording with decision procedures,

Repeatedly Skolemizing and flattening,

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `aligned_overwrite_shift_bits_less.3.2.1`.

`aligned_overwrite_shift_bits_less.3.2.2:`

{-1}	$a' \geq 0$
{-2}	$n' \geq 0$
{-3}	$m' \geq 0$
{-4}	$i' \geq 0$
{-5}	$k' > 0$
{-6}	$i' < a'$
{-7}	$\text{aligned}?(a')(n')$
{-8}	$\text{aligned}?(a' - i')(m')$
{1}	$\text{aligned}?(a')(\text{rem}(\text{expt}(2, i' + k'))(n'))$
{2}	$\text{aligned}?(a')(n' - \text{rem}(\text{expt}(2, i' + k'))(n'))$

Simplifying, rewriting, and recording with decision procedures,

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `aligned_overwrite_shift_bits_less.3.2.2`.

Q.E.D.

C.17.27 `Bits.cut_bits_shift_bits_left_TCC1`

Terse proof for `cut_bits_shift_bits_left_TCC1`.

`cut_bits_shift_bits_left_TCC1:`

{1}	$\forall (si, ci, ck: \text{nat}): ci < si \wedge \neg ci + ck \leq si \supset si - ci \geq 0$
-----	--

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `cut_bits_shift_bits_left_TCC1`.

Q.E.D.

C.17.28 `Bits.cut_bits_shift_bits_left_TCC2`

Terse proof for `cut_bits_shift_bits_left_TCC2`.

cut_bits_shift_bits_left_TCC2:

$$\{1\} \quad \forall (si, ci, ck: \text{nat}): \neg ci < si \wedge \neg ci + ck \leq si \supset ci - si \geq 0$$

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of cut_bits_shift_bits_left_TCC2.
Q.E.D.

C.17.29 Bits.cut_bits_shift_bits_left

Terse proof for cut_bits_shift_bits_left.

cut_bits_shift_bits_left:

$$\{1\} \quad \forall (n, si, ci, ck: \text{nat}):$$

$$\text{cut_bits}(\text{shift_bits_left}(n, si), ci, ck) =$$

$$\text{IF } ci + ck \leq si$$

$$\text{THEN } 0$$

$$\text{ELSIF } ci < si \text{ THEN } \text{cut_bits}(\text{shift_bits_left}(n, si - ci), 0, ck)$$

$$\text{ELSE } \text{cut_bits}(n, ci - si, ck)$$

$$\text{ENDIF}$$

Repeatedly Skolemizing and flattening,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
we get 3 subgoals:

cut_bits_shift_bits_left.1:

$$\{-1\} \quad n' \geq 0$$

$$\{-2\} \quad si' \geq 0$$

$$\{-3\} \quad ci' \geq 0$$

$$\{-4\} \quad ck' \geq 0$$

$$\{-5\} \quad ci' + ck' \leq si'$$

$$\{1\} \quad \text{cut_bits}(\text{shift_bits_left}(n', si'), ci', ck') = 0$$

Expanding the definition of cut_bits,
Rewriting using rem_def2, matching in *,
Expanding the definition of shift_bits_left,
Installing automatic rewrites from: ndiv_times_divident_2 expt_divides ndiv_expt_expt
Simplifying, rewriting, and recording with decision procedures,
Rewriting using divides_prod2, matching in *,
This completes the proof of cut_bits_shift_bits_left.1.

cut_bits_shift_bits_left.2:

$$\{-1\} \quad n' \geq 0$$

$$\{-2\} \quad si' \geq 0$$

$$\{-3\} \quad ci' \geq 0$$

$$\{-4\} \quad ck' \geq 0$$

$$\{-5\} \quad ci' < si'$$

$$\{1\} \quad ci' + ck' \leq si'$$

$$\{2\} \quad \text{cut_bits}(\text{shift_bits_left}(n', si'), ci', ck') =$$

$$\text{cut_bits}(\text{shift_bits_left}(n', si' - ci'), 0, ck')$$

Expanding the definition of shift_bits_left,
Expanding the definition of cut_bits,

C Proof scripts

Installing automatic rewrites from: `ndiv_times_divident_2` `expt_divides` `ndiv_expt_expt` `expt_x0_aux`
`ndiv_1`

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `cut_bits_shift_bits_left.2`.

`cut_bits_shift_bits_left.3`:

{-1}	$n' \geq 0$
{-2}	$si' \geq 0$
{-3}	$ci' \geq 0$
{-4}	$ck' \geq 0$
<hr/>	
{1}	$ci' + ck' \leq si'$
{2}	$ci' < si'$
{3}	$\text{cut_bits}(\text{shift_bits_left}(n', si'), ci', ck') = \text{cut_bits}(n', ci' - si', ck')$

Expanding the definition of `shift_bits_left`,

Expanding the definition of `cut_bits`,

Case splitting on `ndiv(n! * expt(2, si!1), expt(2, ci!1)) = ndiv(n!1, expt(2, ci!1 - si!1))`,

we get 2 subgoals:

`cut_bits_shift_bits_left.3.1`:

{-1}	$\text{ndiv}(n' \times \text{expt}(2, si'), \text{expt}(2, ci')) =$ $\text{ndiv}(n', \text{expt}(2, ci' - si'))$
{-2}	$n' \geq 0$
{-3}	$si' \geq 0$
{-4}	$ci' \geq 0$
{-5}	$ck' \geq 0$
<hr/>	
{1}	$ci' + ck' \leq si'$
{2}	$ci' < si'$
{3}	$\text{rem}(\text{expt}(2, ck'))(\text{ndiv}(n' \times \text{expt}(2, si'), \text{expt}(2, ci'))) =$ $\text{rem}(\text{expt}(2, ck'))(\text{ndiv}(n', \text{expt}(2, ci' - si')))$

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `cut_bits_shift_bits_left.3.1`.

`cut_bits_shift_bits_left.3.2`:

{-1}	$n' \geq 0$
{-2}	$si' \geq 0$
{-3}	$ci' \geq 0$
{-4}	$ck' \geq 0$
<hr/>	
{1}	$\text{ndiv}(n' \times \text{expt}(2, si'), \text{expt}(2, ci')) =$ $\text{ndiv}(n', \text{expt}(2, ci' - si'))$
{2}	$ci' + ck' \leq si'$
{3}	$ci' < si'$
{4}	$\text{rem}(\text{expt}(2, ck'))(\text{ndiv}(n' \times \text{expt}(2, si'), \text{expt}(2, ci'))) =$ $\text{rem}(\text{expt}(2, ck'))(\text{ndiv}(n', \text{expt}(2, ci' - si')))$

Installing automatic rewrites from: `expt_plus`

Rewriting using `ndiv_reduce`, matching in `*`,

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `cut_bits_shift_bits_left.3.2`.

Q.E.D.

C.17.30 `Bits.cut_bits_shift_bits_left_outside`

Terse proof for `cut_bits_shift_bits_left_outside`.

cut_bits_shift_bits_left_outside:

$$\frac{}{\{1\} \quad \forall (n, si, ci, ck: \text{nat}): \quad ci + ck \leq si \supset \text{cut_bits}(\text{shift_bits_left}(n, si), ci, ck) = 0}$$

Repeatedly Skolemizing and flattening,
 Rewriting using cut_bits_shift_bits_left, matching in *,
 Simplifying, rewriting, and recording with decision procedures,
 This completes the proof of cut_bits_shift_bits_left_outside.
 Q.E.D.

C.17.31 Bits.cut_bits_shift_bits_left_overlap_TCC1

Terse proof for cut_bits_shift_bits_left_overlap_TCC1.

cut_bits_shift_bits_left_overlap_TCC1:

$$\frac{}{\{1\} \quad \forall (si, ci, ck: \text{nat}): \quad ci + ck > si \wedge ci < si \wedge ci > 0 \supset si - ci \geq 0}$$

Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of cut_bits_shift_bits_left_overlap_TCC1.
 Q.E.D.

C.17.32 Bits.cut_bits_shift_bits_left_overlap

Terse proof for cut_bits_shift_bits_left_overlap.

cut_bits_shift_bits_left_overlap:

$$\frac{}{\{1\} \quad \forall (n, si, ci, ck: \text{nat}): \quad \begin{aligned} &ci + ck > si \wedge ci < si \wedge ci > 0 \supset \\ &\text{cut_bits}(\text{shift_bits_left}(n, si), ci, ck) = \\ &\text{cut_bits}(\text{shift_bits_left}(n, si - ci), 0, ck) \end{aligned}}$$

Repeatedly Skolemizing and flattening,
 Rewriting using cut_bits_shift_bits_left, matching in *,
 Simplifying, rewriting, and recording with decision procedures,
 This completes the proof of cut_bits_shift_bits_left_overlap.
 Q.E.D.

C.17.33 Bits.cut_bits_shift_bits_left_in

Terse proof for cut_bits_shift_bits_left_in.

cut_bits_shift_bits_left_in:

$$\frac{}{\{1\} \quad \forall (n, si, ci, ck: \text{nat}): \quad ci \geq si \supset \text{cut_bits}(\text{shift_bits_left}(n, si), ci, ck) = \text{cut_bits}(n, ci - si, ck)}$$

Repeatedly Skolemizing and flattening,
 Rewriting using cut_bits_shift_bits_left, matching in *,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 Case splitting on $ck!1 = 0$,
 we get 2 subgoals:

C Proof scripts

cut_bits_shift_bits_left_in.1:

{-1}	$ck' = 0$
{-2}	$n' \geq 0$
{-3}	$si' \geq 0$
{-4}	$ci' \geq 0$
{-5}	$ck' \geq 0$
{-6}	$ci' \geq si'$
{-7}	$ci' + ck' \leq si'$
{1}	$0 = \text{cut_bits}(n', ci' - si', ck')$

Expanding the definition of cut_bits,

Replacing using formula -1,

Rewriting using expt_x0_aux, matching in *,

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of cut_bits_shift_bits_left_in.1.

cut_bits_shift_bits_left_in.2:

{-1}	$n' \geq 0$
{-2}	$si' \geq 0$
{-3}	$ci' \geq 0$
{-4}	$ck' \geq 0$
{-5}	$ci' \geq si'$
{-6}	$ci' + ck' \leq si'$
{1}	$ck' = 0$
{2}	$0 = \text{cut_bits}(n', ci' - si', ck')$

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of cut_bits_shift_bits_left_in.2.

Q.E.D.

C.17.34 Bits.overwrite_shift_bits_zero

Terse proof for overwrite_shift_bits_zero.

overwrite_shift_bits_zero:

{1}	$\forall (z, m, i, k: \text{nat}):$ $z = 0 \supset$ $\text{overwrite_shift_bits}(z, m, i, k) = \text{shift_bits_left}(\text{cut_bits}(m, 0, k), i)$
-----	--

Repeatedly Skolemizing and flattening,

Replacing using formula -5,

Expanding the definition of overwrite_shift_bits,

Expanding the definition of overwrite_bits,

Rewriting using rem_zero, matching in *,

Rewriting using rem_zero, matching in *,

Simplifying, rewriting, and recording with decision procedures,

Rewriting using cut_bits_shift_bits_left, matching in *,

Case splitting on $k!1 = 0$,

we get 2 subgoals:

overwrite_shift_bits_zero.1:

{-1}	$k' = 0$
{-2}	$0 \geq 0$
{-3}	$m' \geq 0$
{-4}	$i' \geq 0$
{-5}	$k' \geq 0$
{-6}	$z' = 0$
{1} $\text{shift_bits_left}(\text{IF } i' + k' \leq i' \text{ THEN } 0 \text{ ELSE } \text{cut_bits}(m', 0, k') \text{ ENDIF}, i') =$ $\text{shift_bits_left}(\text{cut_bits}(m', 0, k'), i')$	

Replacing using formula -1,
Simplifying, rewriting, and recording with decision procedures,
Rewriting using zero_cut_bits, matching in *,
This completes the proof of overwrite_shift_bits_zero.1.

overwrite_shift_bits_zero.2:

{-1}	$0 \geq 0$
{-2}	$m' \geq 0$
{-3}	$i' \geq 0$
{-4}	$k' \geq 0$
{-5}	$z' = 0$
{1} $k' = 0$ {2} $\text{shift_bits_left}(\text{IF } i' + k' \leq i' \text{ THEN } 0 \text{ ELSE } \text{cut_bits}(m', 0, k') \text{ ENDIF}, i') =$ $\text{shift_bits_left}(\text{cut_bits}(m', 0, k'), i')$	

Simplifying, rewriting, and recording with decision procedures,
This completes the proof of overwrite_shift_bits_zero.2.
Q.E.D.

C.17.35 Bits.zero_overwrite_shift_bits

Terse proof for zero_overwrite_shift_bits.

zero_overwrite_shift_bits:

{1}	$\forall (n, m, i, k: \text{nat}): k = 0 \supset \text{overwrite_shift_bits}(n, m, i, k) = n$
-----	---

Repeatedly Skolemizing and flattening,
Replacing using formula -5,
Hiding formulas: -4, -5,
Expanding the definition of overwrite_shift_bits,
Expanding the definition of overwrite_bits,
Rewriting using zero_cut_bits, matching in *,
Expanding the definition of shift_bits_left,
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of zero_overwrite_shift_bits.
Q.E.D.

C.17.36 Bits.bit_split_TCC1

Terse proof for bit_split_TCC1.

bit_split_TCC1:

{1}	$\forall (n, i, k: \text{nat}): k \leq i \wedge n < \text{expt}(2, i) \supset i - k \geq 0$
-----	---

Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of `bit_split_TCC1`.
 Q.E.D.

C.17.37 Bits.bit_split

Terse proof for `bit_split`.

`bit_split`:

$\{1\} \quad \forall (n, i, k: \text{nat}):$ $k \leq i \wedge n < \text{expt}(2, i) \supset$ $n =$ $\text{overwrite_shift_bits}(\text{cut_bits}(n, 0, k), \text{cut_bits}(n, k, i - k), k,$ $i - k)$	
--	--

Repeatedly Skolemizing and flattening,
 Case splitting on `k!1 = i!1`,
 we get 2 subgoals:

`bit_split.1`:

$\{-1\} \quad k' = i'$ $\{-2\} \quad n' \geq 0$ $\{-3\} \quad i' \geq 0$ $\{-4\} \quad k' \geq 0$ $\{-5\} \quad k' \leq i'$ $\{-6\} \quad n' < \text{expt}(2, i')$	
$\{1\} \quad n' =$ $\text{overwrite_shift_bits}(\text{cut_bits}(n', 0, k'), \text{cut_bits}(n', k', i' - k'), k',$ $i' - k')$	

Simplifying, rewriting, and recording with decision procedures,
 Applying `zero_cut_bits`
 Instantiating the top quantifier in -1 with the terms: `n', k', i' - k'`,
 Simplifying, rewriting, and recording with decision procedures,
 Replacing using formula -1,
 Hiding formulas: -1,
 Rewriting using `zero_overwrite_shift_bits`, matching in *,
 Expanding the definition of `cut_bits`,
 Rewriting using `rem_mod`, matching in *,
 Rewriting using `expt_x0_aux`, matching in *,
 Rewriting using `ndiv_1`, matching in *,
 This completes the proof of `bit_split.1`.

`bit_split.2`:

$\{-1\} \quad n' \geq 0$ $\{-2\} \quad i' \geq 0$ $\{-3\} \quad k' \geq 0$ $\{-4\} \quad k' \leq i'$ $\{-5\} \quad n' < \text{expt}(2, i')$	
$\{1\} \quad k' = i'$ $\{2\} \quad n' =$ $\text{overwrite_shift_bits}(\text{cut_bits}(n', 0, k'), \text{cut_bits}(n', k', i' - k'), k',$ $i' - k')$	

Expanding the definition of `overwrite_shift_bits`,
 Expanding the definition of `overwrite_bits`,
 Rewriting using `cut_bits_shift_bits_left`, matching in *,
 Simplifying, rewriting, and recording with decision procedures,
 Rewriting using `cut_bits_cut_bits`, matching in *,
 Rewriting using `rem_mod`, matching in *,
 Rewriting using `rem_mod`, matching in *,
 we get 2 subgoals:

`bit_split.2.1`:

{-1}	$n' \geq 0$	
{-2}	$i' \geq 0$	
{-3}	$k' \geq 0$	
{-4}	$k' \leq i'$	
{-5}	$n' < \text{expt}(2, i')$	
{1}	$k' = i'$	
{2}	$n' =$	$\text{cut_bits}(n', 0, k') + \text{shift_bits_left}(\text{cut_bits}(n', k', i' - k'), k') + \text{cut_bits}(n', 0, k') - \text{cut_bits}(n', 0, k')$

Simplifying, rewriting, and recording with decision procedures,
 Expanding the definition of `shift_bits_left`,
 Expanding the definition of `cut_bits`,
 Rewriting using `expt_x0_aux`, matching in *,
 Rewriting using `ndiv_1`, matching in *,
 Rewriting using `rem_ndiv`, matching in *,
 Rewriting using `expt_plus`, matching in *,
 Applying `rem_mod`
 Instantiating the top quantifier in -1 with the terms: `expt(2, i')`, `n'`,
 Replacing using formula -1,
 Hiding formulas: -1,
 Using lemma `number_split`,
 Simplifying, rewriting, and recording with decision procedures,
 This completes the proof of `bit_split.2.1`.

`bit_split.2.2`:

{-1}	$n' \geq 0$	
{-2}	$i' \geq 0$	
{-3}	$k' \geq 0$	
{-4}	$k' \leq i'$	
{-5}	$n' < \text{expt}(2, i')$	
{1}	$\text{cut_bits}(n', 0, k') < \text{expt}(2, i')$	
{2}	$k' = i'$	
{3}	$n' =$	$\text{shift_bits_left}(\text{cut_bits}(n', k', i' - k'), k') + 2 \times \text{cut_bits}(n', 0, k') - \text{rem}(\text{expt}(2, i'))(\text{cut_bits}(n', 0, k'))$

Hiding formulas: 3,
 Using lemma `both_sides_expt_gt1_le`,
 Simplifying, rewriting, and recording with decision procedures,
 This completes the proof of `bit_split.2.2`.
 Q.E.D.

C.17.38 `Bits.shift_bits_left_cut_bits`

Terse proof for `shift_bits_left_cut_bits`.

C Proof scripts

`shift_bits_left_cut_bits:`

$\{1\} \quad \forall (n, i, k, j: \text{nat}):$ $i = j \wedge \text{aligned?}(i)(n) \wedge n < \text{expt}(2, i + k) \supset$ $\text{shift_bits_left}(\text{cut_bits}(n, i, k), j) = n$
--

Repeatedly Skolemizing and flattening,

Case splitting on $k!1 = 0$,

we get 2 subgoals:

`shift_bits_left_cut_bits.1:`

$\{-1\} \quad k' = 0$ $\{-2\} \quad n' \geq 0$ $\{-3\} \quad i' \geq 0$ $\{-4\} \quad k' \geq 0$ $\{-5\} \quad j' \geq 0$ $\{-6\} \quad i' = j'$ $\{-7\} \quad \text{aligned?}(i')(n')$ $\{-8\} \quad n' < \text{expt}(2, i' + k')$
$\{1\} \quad \text{shift_bits_left}(\text{cut_bits}(n', i', k'), j') = n'$

Replacing using formula -1,

Rewriting using `zero_cut_bits`, matching in `*`,

Expanding the definition of `shift_bits_left`,

Rewriting using `zero_times1`, matching in `*`,

Simplifying, rewriting, and recording with decision procedures,

Expanding the definition of `aligned?`,

Applying `rem_mod`

Instantiating the top quantifier in -1 with the terms: `expt(2, i')`, `n'`,

Applying `rem_def2`

Instantiating the top quantifier in -1 with the terms: `expt(2, i')`, `n'`, 0,

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `shift_bits_left_cut_bits.1`.

`shift_bits_left_cut_bits.2:`

$\{-1\} \quad n' \geq 0$ $\{-2\} \quad i' \geq 0$ $\{-3\} \quad k' \geq 0$ $\{-4\} \quad j' \geq 0$ $\{-5\} \quad i' = j'$ $\{-6\} \quad \text{aligned?}(i')(n')$ $\{-7\} \quad n' < \text{expt}(2, i' + k')$
$\{1\} \quad k' = 0$ $\{2\} \quad \text{shift_bits_left}(\text{cut_bits}(n', i', k'), j') = n'$

Using lemma `bit_split`,

Simplifying, rewriting, and recording with decision procedures,

Using lemma `aligned_cut_bits`,

Simplifying, rewriting, and recording with decision procedures,

Replacing using formula -1,

Rewriting using `overwrite_shift_bits_zero`, matching in `*`,

Rewriting using `cut_bits_cut_bits`, matching in `*`,

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `shift_bits_left_cut_bits.2`.

Q.E.D.

C.17.39 Bits.cut_bits_0_to_below

Terse proof for cut_bits_0_to_below.

cut_bits_0_to_below:

$$\frac{}{\{1\} \quad \forall (n, k: \text{nat}): n < \text{expt}(2, k) \supset \text{cut_bits}(n, 0, k) = n}$$

Repeatedly Skolemizing and flattening,

Applying shift_bits_left_cut_bits

Instantiating the top quantifier in -1 with the terms: n' , 0, k' , 0,

Rewriting using zero_aligned, matching in *

Simplifying, rewriting, and recording with decision procedures,

Expanding the definition of shift_bits_left,

Rewriting using expt_x0_aux, matching in *

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of cut_bits_0_to_below.

Q.E.D.

C.17.40 Bits.cut_bits_overwrite_bits_disjoined

Terse proof for cut_bits_overwrite_bits_disjoined.

cut_bits_overwrite_bits_disjoined:

$$\frac{}{\{1\} \quad \forall (n, m, oi, ok: \text{nat}, ci, ck: \text{nat}): \\ oi + ok \leq ci \vee ci + ck \leq oi \supset \\ \text{cut_bits}(\text{overwrite_bits}(n, m, oi, ok), ci, ck) = \text{cut_bits}(n, ci, ck)}$$

Repeatedly Skolemizing and flattening,

Expanding the definition of overwrite_bits,

Expanding the definition of cut_bits,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 2 subgoals:

cut_bits_overwrite_bits_disjoined.1:

$$\frac{\begin{array}{l} \{-1\} \quad n' \geq 0 \\ \{-2\} \quad m' \geq 0 \\ \{-3\} \quad oi' \geq 0 \\ \{-4\} \quad ok' \geq 0 \\ \{-5\} \quad ci' \geq 0 \\ \{-6\} \quad ck' \geq 0 \\ \{-7\} \quad oi' + ok' \leq ci' \end{array}}{\{1\} \quad \text{rem}(\text{expt}(2, ck')) \\ \quad (\text{ndiv}(\text{rem}(\text{expt}(2, oi'))(n') + \text{shift_bits_left}(\text{rem}(\text{expt}(2, ok'))(\text{ndiv}(m', \text{expt}(2, oi'))), oi') - \text{rem}(\text{expt}(2, \\ \quad \quad \quad \text{expt}(2, ci')))) \\ \quad = \text{rem}(\text{expt}(2, ck'))(\text{ndiv}(n', \text{expt}(2, ci')))}$$

Case splitting on $\text{ndiv}(\text{rem}(\text{expt}(2, oi!1))(n!1) + \text{shift_bits_left}(\text{rem}(\text{expt}(2, ok!1))(\text{ndiv}(m!1, \text{expt}(2, oi!1))), oi!1) - \text{rem}(\text{expt}(2, oi!1 + ok!1))(n!1) + n!1, \text{expt}(2, ci!1)) = \text{ndiv}(\text{rem}(\text{expt}(2, oi!1))(n!1) + \text{shift_bits_left}(\text{rem}(\text{expt}(2, ok!1))(\text{ndiv}(m!1, \text{expt}(2, oi!1))), oi!1) - \text{rem}(\text{expt}(2, oi!1 + ok!1))(n!1) + n!1, \text{expt}(2, oi!1 + ok!1) * \text{expt}(2, ci!1 - oi!1 - ok!1))$,

we get 2 subgoals:

cut_bits_overwrite_bits_disjoined.1.1.1:

$$\begin{array}{l}
 \{-1\} \quad \text{ndiv}(\text{rem}(\text{expt}(2, \text{oi}'))(n') + \text{shift_bits_left}(\text{rem}(\text{expt}(2, \text{ok}'))(\text{ndiv}(m', \text{expt}(2, \text{oi}'))), \text{oi}') - \text{rem}(\text{expt}(2, \text{ci}')) \\
 \quad = \\
 \quad \text{ndiv}(\text{rem}(\text{expt}(2, \text{oi}'))(n') + \text{shift_bits_left}(\text{rem}(\text{expt}(2, \text{ok}'))(\text{ndiv}(m', \text{expt}(2, \text{oi}'))), \text{oi}') - \text{rem}(\text{expt}(2, \text{oi}' + \text{ok}')) \times \text{expt}(2, \text{ci}' - \text{oi}' - \text{ok}')) \\
 \{-2\} \quad n' \geq 0 \\
 \{-3\} \quad m' \geq 0 \\
 \{-4\} \quad \text{oi}' \geq 0 \\
 \{-5\} \quad \text{ok}' \geq 0 \\
 \{-6\} \quad \text{ci}' \geq 0 \\
 \{-7\} \quad \text{ck}' \geq 0 \\
 \{-8\} \quad \text{oi}' + \text{ok}' \leq \text{ci}' \\
 \hline
 \{1\} \quad \text{rem}(\text{expt}(2, \text{ck}')) \\
 \quad (\text{ndiv}(\text{rem}(\text{expt}(2, \text{oi}'))(n') + \text{shift_bits_left}(\text{rem}(\text{expt}(2, \text{ok}'))(\text{ndiv}(m', \text{expt}(2, \text{oi}'))), \text{oi}') - \text{rem}(\text{expt}(2, \text{ci}')) \\
 \quad = \text{rem}(\text{expt}(2, \text{ck}'))(\text{ndiv}(n', \text{expt}(2, \text{ci}'))))
 \end{array}$$

Replacing using formula -1,

Hiding formulas: -1,

Rewriting using ndiv_times_2, matching in *,

Applying rem_def2

Instantiating the top quantifier in -1 with the terms: $\text{expt}(2, \text{oi}' + \text{ok}')$, n' , $\text{rem}(\text{expt}(2, \text{oi}' + \text{ok}'))(n')$,

Simplifying, rewriting, and recording with decision procedures,

Applying ndiv_plus_mod_2

Instantiating the top quantifier in -1 with the terms: $n' - \text{rem}(\text{expt}(2, \text{oi}' + \text{ok}'))(n')$, $\text{expt}(2, \text{oi}' + \text{ok}')$, $\text{rem}(\text{expt}(2, \text{oi}'))(n') + \text{shift_bits_left}(\text{rem}(\text{expt}(2, \text{ok}'))(\text{ndiv}(m', \text{expt}(2, \text{oi}'))), \text{oi}')$, $\text{rem}(\text{expt}(2, \text{oi}' + \text{ok}'))(n')$,

we get 2 subgoals:

cut_bits_overwrite_bits_disjoined.1.1.1.1:

$$\begin{array}{l}
 \{-1\} \quad \text{divides}(\text{expt}(2, \text{oi}' + \text{ok}'), n' - \text{rem}(\text{expt}(2, \text{oi}' + \text{ok}'))(n')) \supset \\
 \quad \text{ndiv}(n' - \text{rem}(\text{expt}(2, \text{oi}' + \text{ok}'))(n') + \text{rem}(\text{expt}(2, \text{oi}'))(n') + \text{shift_bits_left}(\text{rem}(\text{expt}(2, \text{ok}'))(\text{ndiv}(m', \text{expt}(2, \text{oi}'))), \text{oi}') - \text{rem}(\text{expt}(2, \text{oi}' + \text{ok}'))(n'), \\
 \quad \text{expt}(2, \text{oi}' + \text{ok}')) \\
 \quad = \\
 \quad \text{ndiv}(n' - \text{rem}(\text{expt}(2, \text{oi}' + \text{ok}'))(n') + \text{rem}(\text{expt}(2, \text{oi}' + \text{ok}'))(n'), \\
 \quad \text{expt}(2, \text{oi}' + \text{ok}')) \\
 \{-2\} \quad \text{divides}(\text{expt}(2, \text{oi}' + \text{ok}'), n' - \text{rem}(\text{expt}(2, \text{oi}' + \text{ok}'))(n')) \\
 \{-3\} \quad n' \geq 0 \\
 \{-4\} \quad m' \geq 0 \\
 \{-5\} \quad \text{oi}' \geq 0 \\
 \{-6\} \quad \text{ok}' \geq 0 \\
 \{-7\} \quad \text{ci}' \geq 0 \\
 \{-8\} \quad \text{ck}' \geq 0 \\
 \{-9\} \quad \text{oi}' + \text{ok}' \leq \text{ci}' \\
 \hline
 \{1\} \quad \text{rem}(\text{expt}(2, \text{ck}')) \\
 \quad (\text{ndiv}(\text{ndiv}(\text{rem}(\text{expt}(2, \text{oi}'))(n') + \text{shift_bits_left}(\text{rem}(\text{expt}(2, \text{ok}'))(\text{ndiv}(m', \text{expt}(2, \text{oi}'))), \text{oi}') - \text{rem}(\text{expt}(2, \text{oi}' + \text{ok}'))(n'), \\
 \quad \text{expt}(2, \text{oi}' + \text{ok}')), \\
 \quad \text{expt}(2, -1 \times \text{oi}' - \text{ok}' + \text{ci}')) \\
 \quad = \text{rem}(\text{expt}(2, \text{ck}'))(\text{ndiv}(n', \text{expt}(2, \text{ci}'))))
 \end{array}$$

Simplifying, rewriting, and recording with decision procedures,

Replacing using formula -1,

Hiding formulas: -1,

Rewriting using `ndiv_times_2`, matching in `*`,

Rewriting using `expt_plus`, matching in `*`,

This completes the proof of `cut_bits_overwrite_bits_disjoined.1.1.1`.

`cut_bits_overwrite_bits_disjoined.1.1.2`:

{-1}	$\text{divides}(\text{expt}(2, \text{oi}' + \text{ok}'), n' - \text{rem}(\text{expt}(2, \text{oi}' + \text{ok}'))(n'))$
{-2}	$n' \geq 0$
{-3}	$m' \geq 0$
{-4}	$\text{oi}' \geq 0$
{-5}	$\text{ok}' \geq 0$
{-6}	$\text{ci}' \geq 0$
{-7}	$\text{ck}' \geq 0$
{-8}	$\text{oi}' + \text{ok}' \leq \text{ci}'$
{1}	$\text{rem}(\text{expt}(2, \text{oi}'))(n') + \text{shift_bits_left}(\text{rem}(\text{expt}(2, \text{ok}'))(\text{ndiv}(m', \text{expt}(2, \text{oi}'))), \text{oi}') < \text{expt}(2, \text{oi}' + \text{ok}')$
{2}	$\text{rem}(\text{expt}(2, \text{ck}'))(\text{ndiv}(\text{ndiv}(\text{rem}(\text{expt}(2, \text{oi}'))(n') + \text{shift_bits_left}(\text{rem}(\text{expt}(2, \text{ok}'))(\text{ndiv}(m', \text{expt}(2, \text{oi}'))), \text{oi}') - \text{rem}(\text{expt}(2, \text{oi}' + \text{ok}')), \text{expt}(2, -1 \times \text{oi}' - \text{ok}' + \text{ci}')))) = \text{rem}(\text{expt}(2, \text{ck}'))(\text{ndiv}(n', \text{expt}(2, \text{ci}')))$

Hiding formulas: 2,

Expanding the definition of `shift_bits_left`,

Applying `bit_split_less`

Instantiating the top quantifier in -1 with the terms: `rem(expt(2, ok'))(ndiv(m', expt(2, oi')))`, `rem(expt(2, oi'))(n')`, `ok'`, `oi'`,

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `cut_bits_overwrite_bits_disjoined.1.1.2`.

`cut_bits_overwrite_bits_disjoined.1.2`:

{-1}	$n' \geq 0$
{-2}	$m' \geq 0$
{-3}	$\text{oi}' \geq 0$
{-4}	$\text{ok}' \geq 0$
{-5}	$\text{ci}' \geq 0$
{-6}	$\text{ck}' \geq 0$
{-7}	$\text{oi}' + \text{ok}' \leq \text{ci}'$
{1}	$\text{ndiv}(\text{rem}(\text{expt}(2, \text{oi}'))(n') + \text{shift_bits_left}(\text{rem}(\text{expt}(2, \text{ok}'))(\text{ndiv}(m', \text{expt}(2, \text{oi}'))), \text{oi}') - \text{rem}(\text{expt}(2, \text{oi}' + \text{ok}')), \text{expt}(2, \text{ci}')) = \text{ndiv}(\text{rem}(\text{expt}(2, \text{oi}'))(n') + \text{shift_bits_left}(\text{rem}(\text{expt}(2, \text{ok}'))(\text{ndiv}(m', \text{expt}(2, \text{oi}'))), \text{oi}') - \text{rem}(\text{expt}(2, \text{oi}' + \text{ok}') \times \text{expt}(2, \text{ci}' - \text{oi}' - \text{ok}')), \text{expt}(2, \text{ci}'))$
{2}	$\text{rem}(\text{expt}(2, \text{ck}'))(\text{ndiv}(\text{rem}(\text{expt}(2, \text{oi}'))(n') + \text{shift_bits_left}(\text{rem}(\text{expt}(2, \text{ok}'))(\text{ndiv}(m', \text{expt}(2, \text{oi}'))), \text{oi}') - \text{rem}(\text{expt}(2, \text{oi}' + \text{ok}') \times \text{expt}(2, \text{ci}' - \text{oi}' - \text{ok}')), \text{expt}(2, \text{ci}')) = \text{rem}(\text{expt}(2, \text{ck}'))(\text{ndiv}(n', \text{expt}(2, \text{ci}')))$

Rewriting using `expt_plus`, matching in `*`,

This completes the proof of `cut_bits_overwrite_bits_disjoined.1.2`.

C Proof scripts

cut_bits_overwrite_bits_disjoined.2:

{-1}	$n' \geq 0$
{-2}	$m' \geq 0$
{-3}	$oi' \geq 0$
{-4}	$ok' \geq 0$
{-5}	$ci' \geq 0$
{-6}	$ck' \geq 0$
{-7}	$ci' + ck' \leq oi'$
{1}	$\begin{aligned} & \text{rem}(\text{expt}(2, ck')) \\ & \quad (\text{ndiv}(\text{rem}(\text{expt}(2, oi'))(n') + \text{shift_bits_left}(\text{rem}(\text{expt}(2, ok'))(\text{ndiv}(m', \text{expt}(2, oi')))), oi') - \\ & \quad \quad \text{expt}(2, ci')) \\ & = \text{rem}(\text{expt}(2, ck'))(\text{ndiv}(n', \text{expt}(2, ci'))) \end{aligned}$

Rewriting using `rem_ndiv`, matching in `*`,

Rewriting using `expt_plus`, matching in `*`,

Applying `same_remainder`

Instantiating the top quantifier in -1 with the terms: `expt(2, ci' + ck')`, `rem(expt(2, oi'))(n') + shift_bits_left(rem(`
`n'`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 2 subgoals:

cut_bits_overwrite_bits_disjoined.2.1:

{-1}	$\begin{aligned} & \text{rem}(\text{expt}(2, ci' + ck')) \\ & \quad (\text{rem}(\text{expt}(2, oi'))(n') + \text{shift_bits_left}(\text{rem}(\text{expt}(2, ok'))(\text{ndiv}(m', \text{expt}(2, oi')))), oi') - \text{rem}(\\ & = \text{rem}(\text{expt}(2, ci' + ck'))(n') \end{aligned}$
{-2}	$\begin{aligned} & \text{divides}(\text{expt}(2, ci' + ck'), \\ & \quad \text{rem}(\text{expt}(2, oi'))(n') + \text{shift_bits_left}(\text{rem}(\text{expt}(2, ok'))(\text{ndiv}(m', \text{expt}(2, oi'))), oi') - \end{aligned}$
{-3}	$n' \geq 0$
{-4}	$m' \geq 0$
{-5}	$oi' \geq 0$
{-6}	$ok' \geq 0$
{-7}	$ci' \geq 0$
{-8}	$ck' \geq 0$
{-9}	$ci' + ck' \leq oi'$
{1}	$\begin{aligned} & \text{ndiv}(\text{rem}(\text{expt}(2, ci' + ck')) \\ & \quad (\text{rem}(\text{expt}(2, oi'))(n') + \text{shift_bits_left}(\text{rem}(\text{expt}(2, ok'))(\text{ndiv}(m', \text{expt}(2, oi'))), oi') - \\ & \quad \quad \text{expt}(2, ci')) \\ & = \text{rem}(\text{expt}(2, ck'))(\text{ndiv}(n', \text{expt}(2, ci'))) \end{aligned}$

Replacing using formula -1,

Hiding formulas: -1, -2,

Rewriting using `ndiv_rem_divisible`, matching in `*`,

we get 3 subgoals:

cut_bits_overwrite_bits_disjoined.2.1.1:

{-1}	$n' \geq 0$
{-2}	$m' \geq 0$
{-3}	$oi' \geq 0$
{-4}	$ok' \geq 0$
{-5}	$ci' \geq 0$
{-6}	$ck' \geq 0$
{-7}	$ci' + ck' \leq oi'$
{1}	$\begin{aligned} & \text{rem}(\text{ndiv}(\text{expt}(2, ci' + ck'), \text{expt}(2, ci'))(\text{ndiv}(n', \text{expt}(2, ci')))) = \\ & \quad \text{rem}(\text{expt}(2, ck'))(\text{ndiv}(n', \text{expt}(2, ci'))) \end{aligned}$

Rewriting using `ndiv_expt_expt`, matching in `*`,

This completes the proof of `cut_bits_overwrite_bits_disjoined.2.1.1`.

`cut_bits_overwrite_bits_disjoined.2.1.2`:

{-1}	$n' \geq 0$
{-2}	$m' \geq 0$
{-3}	$oi' \geq 0$
{-4}	$ok' \geq 0$
{-5}	$ci' \geq 0$
{-6}	$ck' \geq 0$
{-7}	$ci' + ck' \leq oi'$
{1}	$\text{divides}(\text{expt}(2, ci'), \text{expt}(2, ci' + ck'))$
{2}	$\text{ndiv}(\text{rem}(\text{expt}(2, ci' + ck'))(n'), \text{expt}(2, ci')) =$ $\text{rem}(\text{expt}(2, ck'))(\text{ndiv}(n', \text{expt}(2, ci')))$

Rewriting using `expt_divides`, matching in `*`,

This completes the proof of `cut_bits_overwrite_bits_disjoined.2.1.2`.

`cut_bits_overwrite_bits_disjoined.2.1.3`:

{-1}	$n' \geq 0$
{-2}	$m' \geq 0$
{-3}	$oi' \geq 0$
{-4}	$ok' \geq 0$
{-5}	$ci' \geq 0$
{-6}	$ck' \geq 0$
{-7}	$ci' + ck' \leq oi'$
{1}	$\text{expt}(2, ci') \leq \text{expt}(2, ci' + ck')$
{2}	$\text{ndiv}(\text{rem}(\text{expt}(2, ci' + ck'))(n'), \text{expt}(2, ci')) =$ $\text{rem}(\text{expt}(2, ck'))(\text{ndiv}(n', \text{expt}(2, ci')))$

Using lemma `both_sides_expt_gt1_le`,

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `cut_bits_overwrite_bits_disjoined.2.1.3`.

`cut_bits_overwrite_bits_disjoined.2.2`:

{-1}	$n' \geq 0$
{-2}	$m' \geq 0$
{-3}	$oi' \geq 0$
{-4}	$ok' \geq 0$
{-5}	$ci' \geq 0$
{-6}	$ck' \geq 0$
{-7}	$ci' + ck' \leq oi'$
{1}	$\text{rem}(\text{expt}(2, ci' + ck'))$ $(\text{rem}(\text{expt}(2, oi'))(n') + \text{shift_bits_left}(\text{rem}(\text{expt}(2, ok'))(\text{ndiv}(m', \text{expt}(2, oi'))), oi') - \text{rem}(\text{expt}(2, oi') -$
{2}	$\text{divides}(\text{expt}(2, ci' + ck'),$ $\text{rem}(\text{expt}(2, oi'))(n') + \text{shift_bits_left}(\text{rem}(\text{expt}(2, ok'))(\text{ndiv}(m', \text{expt}(2, oi'))), oi') - \text{rem}(\text{expt}(2, oi') -$
{3}	$\text{ndiv}(\text{rem}(\text{expt}(2, ci' + ck'))$ $(\text{rem}(\text{expt}(2, oi'))(n') + \text{shift_bits_left}(\text{rem}(\text{expt}(2, ok'))(\text{ndiv}(m', \text{expt}(2, oi'))), oi') - \text{rem}(\text{expt}(2, oi') -$
	$\text{expt}(2, ci'))$ $= \text{rem}(\text{expt}(2, ck'))(\text{ndiv}(n', \text{expt}(2, ci')))$

Hiding formulas: 1, 3,

Applying `divides_sum`

Instantiating the top quantifier in -1 with the terms: $\text{rem}(\text{expt}(2, oi'))(n') - \text{rem}(\text{expt}(2, oi' + ok'))(n')$,

C Proof scripts

`shift_bits_left(rem(expt(2, ok'))(ndiv(m', expt(2, oi'))), oi'), expt(2, ci' + ck')`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 2 subgoals:

`cut_bits_overwrite_bits_disjoined.2.2.1:`

$\{-1\} \quad n' \geq 0$ $\{-2\} \quad m' \geq 0$ $\{-3\} \quad oi' \geq 0$ $\{-4\} \quad ok' \geq 0$ $\{-5\} \quad ci' \geq 0$ $\{-6\} \quad ck' \geq 0$ $\{-7\} \quad ci' + ck' \leq oi'$	<hr style="border: 0.5px solid black;"/> $\{1\} \quad \text{divides}(\text{expt}(2, ci' + ck'),$ $\quad \text{rem}(\text{expt}(2, oi'))(n') - \text{rem}(\text{expt}(2, oi' + ok'))(n'))$ $\{2\} \quad \text{divides}(\text{expt}(2, ci' + ck'),$ $\quad \text{rem}(\text{expt}(2, oi'))(n') + \text{shift_bits_left}(\text{rem}(\text{expt}(2, ok'))(\text{ndiv}(m', \text{expt}(2, oi'))), oi') -$
---	---

Hiding formulas: 2,

Applying `divides_diff`

Instantiating the top quantifier in -1 with the terms: $n' - \text{rem}(\text{expt}(2, oi'))(n')$, $n' - \text{rem}(\text{expt}(2, oi' + ok'))(n')$, $\text{expt}(2, ci' + ck')$,

Simplifying, rewriting, and recording with decision procedures,

Hiding formulas: 2,

Splitting conjunctions,

we get 2 subgoals:

`cut_bits_overwrite_bits_disjoined.2.2.1.1:`

$\{-1\} \quad n' \geq 0$ $\{-2\} \quad m' \geq 0$ $\{-3\} \quad oi' \geq 0$ $\{-4\} \quad ok' \geq 0$ $\{-5\} \quad ci' \geq 0$ $\{-6\} \quad ck' \geq 0$ $\{-7\} \quad ci' + ck' \leq oi'$	<hr style="border: 0.5px solid black;"/> $\{1\} \quad \text{divides}(\text{expt}(2, ci' + ck'), n' - \text{rem}(\text{expt}(2, oi' + ok'))(n'))$
---	--

Applying `rem_def2`

Instantiating the top quantifier in -1 with the terms: $\text{expt}(2, oi' + ok')$, n' , $\text{rem}(\text{expt}(2, oi' + ok'))(n')$,

Simplifying, rewriting, and recording with decision procedures,

Applying `divides_expt_gt`

Instantiating the top quantifier in -1 with the terms: $n' - \text{rem}(\text{expt}(2, oi' + ok'))(n')$, $oi' + ok'$, $ci' + ck'$,

we get 2 subgoals:

cut_bits_overwrite_bits_disjoined.2.2.1.1.1:

{-1}	$ci' + ck' \leq oi' + ok' \wedge$ $\text{divides}(\text{expt}(2, oi' + ok'), n' - \text{rem}(\text{expt}(2, oi' + ok'))(n'))$ $\supset \text{divides}(\text{expt}(2, ci' + ck'), n' - \text{rem}(\text{expt}(2, oi' + ok'))(n'))$
{-2}	$\text{divides}(\text{expt}(2, oi' + ok'), n' - \text{rem}(\text{expt}(2, oi' + ok'))(n'))$
{-3}	$n' \geq 0$
{-4}	$m' \geq 0$
{-5}	$oi' \geq 0$
{-6}	$ok' \geq 0$
{-7}	$ci' \geq 0$
{-8}	$ck' \geq 0$
{-9}	$ci' + ck' \leq oi'$
{1}	$\text{divides}(\text{expt}(2, ci' + ck'), n' - \text{rem}(\text{expt}(2, oi' + ok'))(n'))$

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of cut_bits_overwrite_bits_disjoined.2.2.1.1.1.

cut_bits_overwrite_bits_disjoined.2.2.1.1.2:

{-1}	$\text{divides}(\text{expt}(2, oi' + ok'), n' - \text{rem}(\text{expt}(2, oi' + ok'))(n'))$
{-2}	$n' \geq 0$
{-3}	$m' \geq 0$
{-4}	$oi' \geq 0$
{-5}	$ok' \geq 0$
{-6}	$ci' \geq 0$
{-7}	$ck' \geq 0$
{-8}	$ci' + ck' \leq oi'$
{1}	$n' - \text{rem}(\text{expt}(2, oi' + ok'))(n') \geq 0$
{2}	$\text{divides}(\text{expt}(2, ci' + ck'), n' - \text{rem}(\text{expt}(2, oi' + ok'))(n'))$

Hiding formulas: -1, 2,

Using lemma rem_def_pos,

Simplifying, rewriting, and recording with decision procedures,

Repeatedly Skolemizing and flattening,

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of cut_bits_overwrite_bits_disjoined.2.2.1.1.2.

cut_bits_overwrite_bits_disjoined.2.2.1.2:

{-1}	$n' \geq 0$
{-2}	$m' \geq 0$
{-3}	$oi' \geq 0$
{-4}	$ok' \geq 0$
{-5}	$ci' \geq 0$
{-6}	$ck' \geq 0$
{-7}	$ci' + ck' \leq oi'$
{1}	$\text{divides}(\text{expt}(2, ci' + ck'), n' - \text{rem}(\text{expt}(2, oi'))(n'))$

Applying rem_def2

Instantiating the top quantifier in -1 with the terms: $\text{expt}(2, oi')$, n' , $\text{rem}(\text{expt}(2, oi'))(n')$,

Simplifying, rewriting, and recording with decision procedures,

Applying divides_expt_gt

Instantiating the top quantifier in -1 with the terms: $n' - \text{rem}(\text{expt}(2, oi'))(n')$, oi' , $ci' + ck'$,

we get 2 subgoals:

cut_bits_overwrite_bits_disjoined.2.2.1.2.1:

{-1}	$ci' + ck' \leq oi' \wedge$ $\text{divides}(\text{expt}(2, oi'), n' - \text{rem}(\text{expt}(2, oi'))(n'))$ $\supset \text{divides}(\text{expt}(2, ci' + ck'), n' - \text{rem}(\text{expt}(2, oi'))(n'))$
{-2}	$\text{divides}(\text{expt}(2, oi'), n' - \text{rem}(\text{expt}(2, oi'))(n'))$
{-3}	$n' \geq 0$
{-4}	$m' \geq 0$
{-5}	$oi' \geq 0$
{-6}	$ok' \geq 0$
{-7}	$ci' \geq 0$
{-8}	$ck' \geq 0$
{-9}	$ci' + ck' \leq oi'$
{1}	$\text{divides}(\text{expt}(2, ci' + ck'), n' - \text{rem}(\text{expt}(2, oi'))(n'))$

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of cut_bits_overwrite_bits_disjoined.2.2.1.2.1.

cut_bits_overwrite_bits_disjoined.2.2.1.2.2:

{-1}	$\text{divides}(\text{expt}(2, oi'), n' - \text{rem}(\text{expt}(2, oi'))(n'))$
{-2}	$n' \geq 0$
{-3}	$m' \geq 0$
{-4}	$oi' \geq 0$
{-5}	$ok' \geq 0$
{-6}	$ci' \geq 0$
{-7}	$ck' \geq 0$
{-8}	$ci' + ck' \leq oi'$
{1}	$n' - \text{rem}(\text{expt}(2, oi'))(n') \geq 0$
{2}	$\text{divides}(\text{expt}(2, ci' + ck'), n' - \text{rem}(\text{expt}(2, oi'))(n'))$

Hiding formulas: -1, 2,

Using lemma rem_def_pos,

Simplifying, rewriting, and recording with decision procedures,

Repeatedly Skolemizing and flattening,

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of cut_bits_overwrite_bits_disjoined.2.2.1.2.2.

cut_bits_overwrite_bits_disjoined.2.2.2:

{-1}	$n' \geq 0$
{-2}	$m' \geq 0$
{-3}	$oi' \geq 0$
{-4}	$ok' \geq 0$
{-5}	$ci' \geq 0$
{-6}	$ck' \geq 0$
{-7}	$ci' + ck' \leq oi'$
{1}	$\text{divides}(\text{expt}(2, ci' + ck'),$ $\text{shift_bits_left}(\text{rem}(\text{expt}(2, ok'))(\text{ndiv}(m', \text{expt}(2, oi'))), oi'))$
{2}	$\text{divides}(\text{expt}(2, ci' + ck'),$ $\text{rem}(\text{expt}(2, oi'))(n') + \text{shift_bits_left}(\text{rem}(\text{expt}(2, ok'))(\text{ndiv}(m', \text{expt}(2, oi'))), oi') -$

Hiding formulas: 2,

Expanding the definition of shift_bits_left,

Rewriting using divides_prod2, matching in *,

Rewriting using expt_divides, matching in *,

This completes the proof of cut_bits_overwrite_bits_disjoined.2.2.2.

Q.E.D.

C.17.41 Bits.cut_bits_overwrite_bool_bit_disjoined

Terse proof for cut_bits_overwrite_bool_bit_disjoined.

cut_bits_overwrite_bool_bit_disjoined:

$$\frac{}{\{1\} \quad \forall (n: \text{nat}, b: \text{bool}, oi, ci, ck: \text{nat}): \\ oi < ci \vee ci + ck \leq oi \supset \\ \text{cut_bits}(\text{overwrite_bool_bit}(n, b, oi), ci, ck) = \text{cut_bits}(n, ci, ck)}$$

Repeatedly Skolemizing and flattening,

Expanding the definition of overwrite_bool_bit,

Expanding the definition of overwrite_bit,

Rewriting using cut_bits_overwrite_bits_disjoined, matching in *,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of cut_bits_overwrite_bool_bit_disjoined.

Q.E.D.

C.17.42 Bits.cut_bits_overwrite_bits_contained

Terse proof for cut_bits_overwrite_bits_contained.

cut_bits_overwrite_bits_contained:

$$\frac{}{\{1\} \quad \forall (n, m, oi, ok, ci, ck: \text{nat}): \\ oi \leq ci \wedge ci + ck \leq oi + ok \supset \\ \text{cut_bits}(\text{overwrite_bits}(n, m, oi, ok), ci, ck) = \text{cut_bits}(m, ci, ck)}$$

Repeatedly Skolemizing and flattening,

Expanding the definition of overwrite_bits,

Expanding the definition of shift_bits_left,

Expanding the definition of cut_bits,

Applying number_split

Instantiating the top quantifier in -1 with the terms: $\text{rem}(\text{expt}(2, ok'))(\text{ndiv}(m', \text{expt}(2, oi')))$, $\text{expt}(2, ci' - oi')$,

we get 2 subgoals:

cut_bits_overwrite_bits_contained.1:

$$\frac{\begin{array}{l} \{-1\} \quad \text{rem}(\text{expt}(2, ok'))(\text{ndiv}(m', \text{expt}(2, oi'))) = \\ \quad \text{expt}(2, ci' - oi') \times \text{ndiv}(\text{rem}(\text{expt}(2, ok'))(\text{ndiv}(m', \text{expt}(2, oi'))), \text{expt}(2, ci' - oi')) + \text{rem}(\text{expt}(2, ci' - \\ \{-2\} \quad n' \geq 0 \\ \{-3\} \quad m' \geq 0 \\ \{-4\} \quad oi' \geq 0 \\ \{-5\} \quad ok' \geq 0 \\ \{-6\} \quad ci' \geq 0 \\ \{-7\} \quad ck' \geq 0 \\ \{-8\} \quad oi' \leq ci' \\ \{-9\} \quad ci' + ck' \leq oi' + ok' \end{array}}{\{1\} \quad \text{rem}(\text{expt}(2, ck')) \\ (\text{ndiv}(\text{rem}(\text{expt}(2, oi'))(n') + \text{rem}(\text{expt}(2, ok'))(\text{ndiv}(m', \text{expt}(2, oi')))) \times \text{expt}(2, oi') - \text{rem}(\text{expt}(2, oi') \\ \text{expt}(2, ci'))) \\ = \text{rem}(\text{expt}(2, ck'))(\text{ndiv}(m', \text{expt}(2, ci')))}$$

C Proof scripts

Installing automatic rewrites from: `expt_divides expt_plus`

Rewriting using `rem_rem`, matching in `*`,

we get 2 subgoals:

`cut_bits_overwrite_bits_contained.1.1:`

$$\begin{array}{l}
 \{-1\} \quad \text{rem}(\text{expt}(2, \text{ok}'))(\text{ndiv}(m', \text{expt}(2, \text{oi}'))) = \\
 \quad \text{expt}(2, \text{ci}' - \text{oi}') \times \text{ndiv}(\text{rem}(\text{expt}(2, \text{ok}'))(\text{ndiv}(m', \text{expt}(2, \text{oi}'))), \text{expt}(2, \text{ci}' - \text{oi}')) + \text{rem}(\text{expt}(2, \text{ci}' - \text{oi}'), \text{expt}(2, \text{ci}' - \text{oi}')) \\
 \{-2\} \quad n' \geq 0 \\
 \{-3\} \quad m' \geq 0 \\
 \{-4\} \quad \text{oi}' \geq 0 \\
 \{-5\} \quad \text{ok}' \geq 0 \\
 \{-6\} \quad \text{ci}' \geq 0 \\
 \{-7\} \quad \text{ck}' \geq 0 \\
 \{-8\} \quad \text{oi}' \leq \text{ci}' \\
 \{-9\} \quad \text{ci}' + \text{ck}' \leq \text{oi}' + \text{ok}' \\
 \hline
 \{1\} \quad \text{rem}(\text{expt}(2, \text{ck}')) \\
 \quad (\text{ndiv}(\text{rem}(\text{expt}(2, \text{oi}'))(n') + \text{rem}(\text{expt}(2, \text{ok}'))(\text{ndiv}(m', \text{expt}(2, \text{oi}')))) \times \text{expt}(2, \text{oi}') - \text{rem}(\text{expt}(2, \text{ci}' - \text{oi}'), \text{expt}(2, \text{ci}' - \text{oi}')) \\
 \quad \text{expt}(2, \text{ci}')) \\
 \quad = \text{rem}(\text{expt}(2, \text{ck}'))(\text{ndiv}(m', \text{expt}(2, \text{ci}')))
 \end{array}$$

Rewriting using `ndiv_rem_divisible`, matching in `-1`,

we get 3 subgoals:

`cut_bits_overwrite_bits_contained.1.1.1:`

$$\begin{array}{l}
 \{-1\} \quad \text{rem}(\text{expt}(2, \text{ok}'))(\text{ndiv}(m', \text{expt}(2, \text{oi}'))) = \\
 \quad \text{expt}(2, \text{ci}' - \text{oi}') \times \text{rem}(\text{ndiv}(\text{expt}(2, \text{ok}'), \text{expt}(2, \text{ci}' - \text{oi}'))(\text{ndiv}(\text{ndiv}(m', \text{expt}(2, \text{oi}'))), \text{expt}(2, \text{ci}' - \text{oi}')) \\
 \{-2\} \quad n' \geq 0 \\
 \{-3\} \quad m' \geq 0 \\
 \{-4\} \quad \text{oi}' \geq 0 \\
 \{-5\} \quad \text{ok}' \geq 0 \\
 \{-6\} \quad \text{ci}' \geq 0 \\
 \{-7\} \quad \text{ck}' \geq 0 \\
 \{-8\} \quad \text{oi}' \leq \text{ci}' \\
 \{-9\} \quad \text{ci}' + \text{ck}' \leq \text{oi}' + \text{ok}' \\
 \hline
 \{1\} \quad \text{rem}(\text{expt}(2, \text{ck}')) \\
 \quad (\text{ndiv}(\text{rem}(\text{expt}(2, \text{oi}'))(n') + \text{rem}(\text{expt}(2, \text{ok}'))(\text{ndiv}(m', \text{expt}(2, \text{oi}')))) \times \text{expt}(2, \text{oi}') - \text{rem}(\text{expt}(2, \text{ci}' - \text{oi}'), \text{expt}(2, \text{ci}' - \text{oi}')) \\
 \quad \text{expt}(2, \text{ci}')) \\
 \quad = \text{rem}(\text{expt}(2, \text{ck}'))(\text{ndiv}(m', \text{expt}(2, \text{ci}')))
 \end{array}$$

Rewriting using `ndiv_times_2`, matching in `*`,

Rewriting using `expt_plus`, matching in `*`,

Rewriting using `ndiv_expt_expt`, matching in `*`,

Simplifying, rewriting, and recording with decision procedures,

Replacing using formula `-1`,

Hiding formulas: `-1`,

Simplifying, rewriting, and recording with decision procedures,

Rewriting using `associative_mult`, matching in `*`,

Applying `ndiv_plus_mod`

Instantiating the top quantifier in `-1` with the terms: $n' - \text{rem}(\text{expt}(2, \text{oi}' + \text{ok}'))(n') + \text{rem}(\text{expt}(2, \text{oi}' - \text{ci}' + \text{ok}'))(n')$, $\text{rem}(\text{expt}(2, \text{ci}'))$, $\text{rem}(\text{expt}(2, \text{oi}'))(n') + \text{rem}(\text{expt}(2, \text{ci}' - \text{oi}'))(\text{ndiv}(m', \text{expt}(2, \text{oi}')))) \times \text{expt}(2, \text{oi}')$,

we get 2 subgoals:

cut_bits_overwrite_bits_contained.1.1.1.1:

<pre> {-1} divides(expt(2, ci'), n' - rem(expt(2, oi' + ok'))(n') + rem(expt(2, oi' - ci' + ok'))(ndiv(m', expt(2, ci'))) × expt(2, ci') ⊃ ndiv(n' - rem(expt(2, oi' + ok'))(n') + rem(expt(2, oi' - ci' + ok'))(ndiv(m', expt(2, ci'))) × expt(2, ci') + expt(2, ci')) = ndiv(n' - rem(expt(2, oi' + ok'))(n') + rem(expt(2, oi' - ci' + ok'))(ndiv(m', expt(2, ci'))) × expt(2, ci') expt(2, ci')) {-2} n' ≥ 0 {-3} m' ≥ 0 {-4} oi' ≥ 0 {-5} ok' ≥ 0 {-6} ci' ≥ 0 {-7} ck' ≥ 0 {-8} oi' ≤ ci' {-9} ci' + ck' ≤ oi' + ok' </pre>	<pre> {1} rem(expt(2, ck')) (ndiv(rem(expt(2, oi'))(n') + rem(expt(2, ci' - oi'))(ndiv(m', expt(2, oi'))) × expt(2, oi') - rem(expt(2, ci') expt(2, ci'))) = rem(expt(2, ck'))(ndiv(m', expt(2, ci'))) </pre>
---	--

Splitting conjunctions,

we get 2 subgoals:

cut_bits_overwrite_bits_contained.1.1.1.1.1:

<pre> {-1} ndiv(n' - rem(expt(2, oi' + ok'))(n') + rem(expt(2, oi' - ci' + ok'))(ndiv(m', expt(2, ci'))) × expt(2, ci') + expt(2, ci')) = ndiv(n' - rem(expt(2, oi' + ok'))(n') + rem(expt(2, oi' - ci' + ok'))(ndiv(m', expt(2, ci'))) × expt(2, ci') expt(2, ci')) {-2} n' ≥ 0 {-3} m' ≥ 0 {-4} oi' ≥ 0 {-5} ok' ≥ 0 {-6} ci' ≥ 0 {-7} ck' ≥ 0 {-8} oi' ≤ ci' {-9} ci' + ck' ≤ oi' + ok' </pre>	<pre> {1} rem(expt(2, ck')) (ndiv(rem(expt(2, oi'))(n') + rem(expt(2, ci' - oi'))(ndiv(m', expt(2, oi'))) × expt(2, oi') - rem(expt(2, ci') expt(2, ci'))) = rem(expt(2, ck'))(ndiv(m', expt(2, ci'))) </pre>
--	--

Simplifying, rewriting, and recording with decision procedures,

Replacing using formula -1,

Hiding formulas: -1,

Applying ndiv_plus_1

Instantiating the top quantifier in -1 with the terms: $\text{rem}(\text{expt}(2, \text{oi}' - \text{ci}' + \text{ok}'))(\text{ndiv}(m', \text{expt}(2, \text{ci}')) \times \text{expt}(2, \text{ci}'))$, $n' - \text{rem}(\text{expt}(2, \text{oi}' + \text{ok}'))(n')$, $\text{expt}(2, \text{ci}')$,

Rewriting using divides_prod2, matching in *,

Simplifying, rewriting, and recording with decision procedures,

Replacing using formula -1,

Hiding formulas: -1,

C Proof scripts

Rewriting using `ndiv_times_divident_2`, matching in `*`,
 Rewriting using `ndiv_self`, matching in `*`,
 Simplifying, rewriting, and recording with decision procedures,
 Case splitting on $n! - \text{rem}(\text{expt}(2, \text{oi}! + \text{ok}!))(n!) = \text{ndiv}(n!, \text{expt}(2, \text{oi}! + \text{ok}!)) * \text{expt}(2, \text{oi}! + \text{ok}!)$,

we get 2 subgoals:

`cut_bits_overwrite_bits_contained.1.1.1.1.1.1:`

$\{-1\} \quad n' - \text{rem}(\text{expt}(2, \text{oi}' + \text{ok}'))(n') =$ $\text{ndiv}(n', \text{expt}(2, \text{oi}' + \text{ok}')) \times \text{expt}(2, \text{oi}' + \text{ok}')$	
$\{-2\} \quad n' \geq 0$	
$\{-3\} \quad m' \geq 0$	
$\{-4\} \quad \text{oi}' \geq 0$	
$\{-5\} \quad \text{ok}' \geq 0$	
$\{-6\} \quad \text{ci}' \geq 0$	
$\{-7\} \quad \text{ck}' \geq 0$	
$\{-8\} \quad \text{oi}' \leq \text{ci}'$	
$\{-9\} \quad \text{ci}' + \text{ck}' \leq \text{oi}' + \text{ok}'$	
$\{1\} \quad \text{rem}(\text{expt}(2, \text{ck}'))$ $(\text{rem}(\text{expt}(2, \text{oi}' - \text{ci}' + \text{ok}'))(\text{ndiv}(m', \text{expt}(2, \text{ci}')))) + \text{ndiv}(n' - \text{rem}(\text{expt}(2, \text{oi}' + \text{ok}'))(n'),$ $= \text{rem}(\text{expt}(2, \text{ck}'))(\text{ndiv}(m', \text{expt}(2, \text{ci}')))$	

Replacing using formula -1,

Hiding formulas: -1,

Rewriting using `ndiv_times_divident_2`, matching in `*`,

Rewriting using `ndiv_expt_expt`, matching in `*`,

Applying `rem_sum_elim1`

Instantiating the top quantifier in -1 with the terms: $\text{expt}(2, \text{ck}')$, $\text{rem}(\text{expt}(2, \text{oi}' - \text{ci}' + \text{ok}'))(\text{ndiv}(m', \text{expt}(2, \text{ci}')))$, $\text{ndiv}(n', \text{expt}(2, \text{oi}' + \text{ok}')) \times \text{expt}(2, \text{oi}' - \text{ci}' + \text{ok}')$, 0,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 2 subgoals:

`cut_bits_overwrite_bits_contained.1.1.1.1.1.1.1:`

$\{-1\} \quad \text{rem}(\text{expt}(2, \text{ck}'))$ $(\text{rem}(\text{expt}(2, \text{oi}' - \text{ci}' + \text{ok}'))(\text{ndiv}(m', \text{expt}(2, \text{ci}')))) + \text{ndiv}(n', \text{expt}(2, \text{oi}' + \text{ok}')) \times \text{expt}(2,$ $=$ $\text{rem}(\text{expt}(2, \text{ck}'))$ $(\text{rem}(\text{expt}(2, \text{oi}' - \text{ci}' + \text{ok}'))(\text{ndiv}(m', \text{expt}(2, \text{ci}'))))$	
$\{-2\} \quad \text{rem}(\text{expt}(2, \text{ck}'))$ $(\text{ndiv}(n', \text{expt}(2, \text{oi}' + \text{ok}')) \times \text{expt}(2, \text{oi}' - \text{ci}' + \text{ok}'))$ $= \text{rem}(\text{expt}(2, \text{ck}'))(0)$	
$\{-3\} \quad n' \geq 0$	
$\{-4\} \quad m' \geq 0$	
$\{-5\} \quad \text{oi}' \geq 0$	
$\{-6\} \quad \text{ok}' \geq 0$	
$\{-7\} \quad \text{ci}' \geq 0$	
$\{-8\} \quad \text{ck}' \geq 0$	
$\{-9\} \quad \text{oi}' \leq \text{ci}'$	
$\{-10\} \quad \text{ci}' + \text{ck}' \leq \text{oi}' + \text{ok}'$	
$\{1\} \quad \text{rem}(\text{expt}(2, \text{ck}'))$ $(\text{rem}(\text{expt}(2, \text{oi}' - \text{ci}' + \text{ok}'))(\text{ndiv}(m', \text{expt}(2, \text{ci}')))) + \text{ndiv}(n', \text{expt}(2, \text{oi}' + \text{ok}')) \times \text{expt}(2,$ $= \text{rem}(\text{expt}(2, \text{ck}'))(\text{ndiv}(m', \text{expt}(2, \text{ci}')))$	

Replacing using formula -1,

Rewriting using `rem_rem`, matching in `*`,

This completes the proof of `cut_bits_overwrite_bits_contained.1.1.1.1.1.1.1`.

`cut_bits_overwrite_bits_contained.1.1.1.1.1.1.2`:

{-1}	$n' \geq 0$
{-2}	$m' \geq 0$
{-3}	$oi' \geq 0$
{-4}	$ok' \geq 0$
{-5}	$ci' \geq 0$
{-6}	$ck' \geq 0$
{-7}	$oi' \leq ci'$
{-8}	$ci' + ck' \leq oi' + ok'$
{1}	$\begin{aligned} & \text{rem}(\text{expt}(2, ck')) \\ & \quad (\text{rem}(\text{expt}(2, oi' - ci' + ok'))(\text{ndiv}(m', \text{expt}(2, ci')))) + \text{ndiv}(n', \text{expt}(2, oi' + ok')) \times \text{expt}(2, oi' - ci' + ok') \\ & = \\ & \quad \text{rem}(\text{expt}(2, ck')) \\ & \quad \quad (\text{rem}(\text{expt}(2, oi' - ci' + ok'))(\text{ndiv}(m', \text{expt}(2, ci')))) \end{aligned}$
{2}	$\begin{aligned} & \text{rem}(\text{expt}(2, ck')) \\ & \quad (\text{ndiv}(n', \text{expt}(2, oi' + ok')) \times \text{expt}(2, oi' - ci' + ok')) \\ & = \text{rem}(\text{expt}(2, ck'))(0) \end{aligned}$
{3}	$\begin{aligned} & \text{rem}(\text{expt}(2, ck')) \\ & \quad (\text{rem}(\text{expt}(2, oi' - ci' + ok'))(\text{ndiv}(m', \text{expt}(2, ci')))) + \text{ndiv}(n', \text{expt}(2, oi' + ok')) \times \text{expt}(2, oi' - ci' + ok') \\ & = \text{rem}(\text{expt}(2, ck'))(\text{ndiv}(m', \text{expt}(2, ci'))) \end{aligned}$

Hiding formulas: 1, 3,

Rewriting using `rem_zero`, matching in `*`,

Rewriting using `rem_def2`, matching in `*`,

Rewriting using `divides_prod2`, matching in `*`,

This completes the proof of `cut_bits_overwrite_bits_contained.1.1.1.1.1.1.2`.

`cut_bits_overwrite_bits_contained.1.1.1.1.1.2`:

{-1}	$n' \geq 0$
{-2}	$m' \geq 0$
{-3}	$oi' \geq 0$
{-4}	$ok' \geq 0$
{-5}	$ci' \geq 0$
{-6}	$ck' \geq 0$
{-7}	$oi' \leq ci'$
{-8}	$ci' + ck' \leq oi' + ok'$
{1}	$\begin{aligned} & n' - \text{rem}(\text{expt}(2, oi' + ok'))(n') = \\ & \quad \text{ndiv}(n', \text{expt}(2, oi' + ok')) \times \text{expt}(2, oi' + ok') \end{aligned}$
{2}	$\begin{aligned} & \text{rem}(\text{expt}(2, ck')) \\ & \quad (\text{rem}(\text{expt}(2, oi' - ci' + ok'))(\text{ndiv}(m', \text{expt}(2, ci')))) + \text{ndiv}(n' - \text{rem}(\text{expt}(2, oi' + ok'))(n'), \text{expt}(2, ci')) \\ & = \text{rem}(\text{expt}(2, ck'))(\text{ndiv}(m', \text{expt}(2, ci'))) \end{aligned}$

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `cut_bits_overwrite_bits_contained.1.1.1.1.1.2`.

C Proof scripts

cut_bits_overwrite_bits_contained.1.1.1.1.2:

{-1}	$n' \geq 0$
{-2}	$m' \geq 0$
{-3}	$oi' \geq 0$
{-4}	$ok' \geq 0$
{-5}	$ci' \geq 0$
{-6}	$ck' \geq 0$
{-7}	$oi' \leq ci'$
{-8}	$ci' + ck' \leq oi' + ok'$
{1}	$\text{divides}(\text{expt}(2, ci'),$ $n' - \text{rem}(\text{expt}(2, oi' + ok'))(n') + \text{rem}(\text{expt}(2, oi' - ci' + ok'))(\text{ndiv}(m', \text{expt}(2, ci'))))$
{2}	$\text{rem}(\text{expt}(2, ck'))$ $(\text{ndiv}(\text{rem}(\text{expt}(2, oi'))(n') + \text{rem}(\text{expt}(2, ci' - oi'))(\text{ndiv}(m', \text{expt}(2, oi')))) \times \text{expt}(2, oi') -$ $\text{expt}(2, ci'))$ $= \text{rem}(\text{expt}(2, ck'))(\text{ndiv}(m', \text{expt}(2, ci')))$

Hiding formulas: 2,

Rewriting using divides_sum, matching in *,

we get 2 subgoals:

cut_bits_overwrite_bits_contained.1.1.1.1.2.1:

{-1}	$n' \geq 0$
{-2}	$m' \geq 0$
{-3}	$oi' \geq 0$
{-4}	$ok' \geq 0$
{-5}	$ci' \geq 0$
{-6}	$ck' \geq 0$
{-7}	$oi' \leq ci'$
{-8}	$ci' + ck' \leq oi' + ok'$
{1}	$\text{divides}(\text{expt}(2, ci'), n' - \text{rem}(\text{expt}(2, oi' + ok'))(n'))$
{2}	$\text{divides}(\text{expt}(2, ci'),$ $\text{rem}(\text{expt}(2, oi' - ci' + ok'))(\text{ndiv}(m', \text{expt}(2, ci')))) \times \text{expt}(2, ci') - \text{rem}(\text{expt}(2, oi' +$

Hiding formulas: 2,

Applying rem_def2

Instantiating the top quantifier in -1 with the terms: $\text{expt}(2, oi' + ok')$, n' , $\text{rem}(\text{expt}(2, oi' + ok'))(n')$,

Simplifying, rewriting, and recording with decision procedures,

Applying divides_expt_gt

Instantiating the top quantifier in -1 with the terms: $n' - \text{rem}(\text{expt}(2, oi' + ok'))(n')$, $oi' + ok'$, ci' ,

we get 2 subgoals:

cut_bits_overwrite_bits_contained.1.1.1.1.2.1.1:

{-1}	$ci' \leq oi' + ok' \wedge$ $\text{divides}(\text{expt}(2, oi' + ok'), n' - \text{rem}(\text{expt}(2, oi' + ok'))(n'))$ $\supset \text{divides}(\text{expt}(2, ci'), n' - \text{rem}(\text{expt}(2, oi' + ok'))(n'))$
{-2}	$\text{divides}(\text{expt}(2, oi' + ok'), n' - \text{rem}(\text{expt}(2, oi' + ok'))(n'))$
{-3}	$n' \geq 0$
{-4}	$m' \geq 0$
{-5}	$oi' \geq 0$
{-6}	$ok' \geq 0$
{-7}	$ci' \geq 0$
{-8}	$ck' \geq 0$
{-9}	$oi' \leq ci'$
{-10}	$ci' + ck' \leq oi' + ok'$
{1}	$\text{divides}(\text{expt}(2, ci'), n' - \text{rem}(\text{expt}(2, oi' + ok'))(n'))$

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of cut_bits_overwrite_bits_contained.1.1.1.1.2.1.1.

cut_bits_overwrite_bits_contained.1.1.1.1.2.1.2:

{-1}	$\text{divides}(\text{expt}(2, oi' + ok'), n' - \text{rem}(\text{expt}(2, oi' + ok'))(n'))$
{-2}	$n' \geq 0$
{-3}	$m' \geq 0$
{-4}	$oi' \geq 0$
{-5}	$ok' \geq 0$
{-6}	$ci' \geq 0$
{-7}	$ck' \geq 0$
{-8}	$oi' \leq ci'$
{-9}	$ci' + ck' \leq oi' + ok'$
{1}	$n' - \text{rem}(\text{expt}(2, oi' + ok'))(n') \geq 0$
{2}	$\text{divides}(\text{expt}(2, ci'), n' - \text{rem}(\text{expt}(2, oi' + ok'))(n'))$

Hiding formulas: -1, 2,

Using lemma rem_def_pos,

Simplifying, rewriting, and recording with decision procedures,

Repeatedly Skolemizing and flattening,

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of cut_bits_overwrite_bits_contained.1.1.1.1.2.1.2.

cut_bits_overwrite_bits_contained.1.1.1.1.2.2:

{-1}	$n' \geq 0$
{-2}	$m' \geq 0$
{-3}	$oi' \geq 0$
{-4}	$ok' \geq 0$
{-5}	$ci' \geq 0$
{-6}	$ck' \geq 0$
{-7}	$oi' \leq ci'$
{-8}	$ci' + ck' \leq oi' + ok'$
{1}	$\text{divides}(\text{expt}(2, ci'),$ $\text{rem}(\text{expt}(2, oi' - ci' + ok'))(\text{ndiv}(m', \text{expt}(2, ci')))) \times \text{expt}(2, ci')$
{2}	$\text{divides}(\text{expt}(2, ci'),$ $\text{rem}(\text{expt}(2, oi' - ci' + ok'))(\text{ndiv}(m', \text{expt}(2, ci')))) \times \text{expt}(2, ci') - \text{rem}(\text{expt}(2, oi' + ok'))(n') +$

Hiding formulas: 2,

Rewriting using divides_prod2, matching in *,

This completes the proof of cut_bits_overwrite_bits_contained.1.1.1.1.2.2.

C Proof scripts

cut_bits_overwrite_bits_contained.1.1.1.2:

<div style="display: flex; flex-direction: column; gap: 5px;"> <div style="border-bottom: 1px solid black; padding-bottom: 5px;"> <p>{-1} $n' \geq 0$</p> <p>{-2} $m' \geq 0$</p> <p>{-3} $oi' \geq 0$</p> <p>{-4} $ok' \geq 0$</p> <p>{-5} $ci' \geq 0$</p> <p>{-6} $ck' \geq 0$</p> <p>{-7} $oi' \leq ci'$</p> <p>{-8} $ci' + ck' \leq oi' + ok'$</p> </div> <div style="padding: 5px;"> <p>{1} $\text{rem}(\text{expt}(2, oi')(n') + \text{rem}(\text{expt}(2, ci' - oi'))(\text{ndiv}(m', \text{expt}(2, oi')))) \times \text{expt}(2, oi')$ $< \text{expt}(2, ci')$</p> <p>{2} $\text{rem}(\text{expt}(2, ck')$ $(\text{ndiv}(\text{rem}(\text{expt}(2, oi')(n') + \text{rem}(\text{expt}(2, ci' - oi'))(\text{ndiv}(m', \text{expt}(2, oi')))) \times \text{expt}(2, oi') -$ $\text{expt}(2, ci'))$ $= \text{rem}(\text{expt}(2, ck'))(\text{ndiv}(m', \text{expt}(2, ci'))))$</p> </div> </div>	
---	--

Hiding formulas: 2,

Applying bit_split_less

Instantiating the top quantifier in -1 with the terms: $\text{rem}(\text{expt}(2, ci' - oi'))(\text{ndiv}(m', \text{expt}(2, oi')))$, $\text{rem}(\text{expt}(2, oi')(n'), ci' - oi', oi'$,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of cut_bits_overwrite_bits_contained.1.1.1.2.

cut_bits_overwrite_bits_contained.1.1.2:

<div style="display: flex; flex-direction: column; gap: 5px;"> <div style="border-bottom: 1px solid black; padding-bottom: 5px;"> <p>{-1} $\text{rem}(\text{expt}(2, ok'))(\text{ndiv}(m', \text{expt}(2, oi'))) =$ $\text{rem}(\text{expt}(2, ci' - oi'))(\text{ndiv}(m', \text{expt}(2, oi'))) + \text{expt}(2, ci' - oi') \times \text{ndiv}(\text{rem}(\text{expt}(2, ok'))(\text{ndiv}(m', \text{expt}(2, oi'))))$</p> <p>{-2} $n' \geq 0$</p> <p>{-3} $m' \geq 0$</p> <p>{-4} $oi' \geq 0$</p> <p>{-5} $ok' \geq 0$</p> <p>{-6} $ci' \geq 0$</p> <p>{-7} $ck' \geq 0$</p> <p>{-8} $oi' \leq ci'$</p> <p>{-9} $ci' + ck' \leq oi' + ok'$</p> </div> <div style="padding: 5px;"> <p>{1} $\text{divides}(\text{expt}(2, ci' - oi'), \text{expt}(2, ok'))$</p> <p>{2} $\text{rem}(\text{expt}(2, ck')$ $(\text{ndiv}(\text{rem}(\text{expt}(2, oi')(n') + \text{rem}(\text{expt}(2, ok'))(\text{ndiv}(m', \text{expt}(2, oi')))) \times \text{expt}(2, oi') - \text{rem}(\text{expt}(2, ci'))$ $\text{expt}(2, ci'))$ $= \text{rem}(\text{expt}(2, ck'))(\text{ndiv}(m', \text{expt}(2, ci'))))$</p> </div> </div>	
---	--

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of cut_bits_overwrite_bits_contained.1.1.2.

cut_bits_overwrite_bits_contained.1.1.3:

$$\begin{array}{l}
\{-1\} \quad \text{rem}(\text{expt}(2, \text{ok}'))(\text{ndiv}(m', \text{expt}(2, \text{oi}')))) = \\
\quad \text{rem}(\text{expt}(2, \text{ci}' - \text{oi}'))(\text{ndiv}(m', \text{expt}(2, \text{oi}')))) + \text{expt}(2, \text{ci}' - \text{oi}') \times \text{ndiv}(\text{rem}(\text{expt}(2, \text{ok}'))(\text{ndiv}(m', \text{expt}(2, \text{oi}')))) \\
\{-2\} \quad n' \geq 0 \\
\{-3\} \quad m' \geq 0 \\
\{-4\} \quad \text{oi}' \geq 0 \\
\{-5\} \quad \text{ok}' \geq 0 \\
\{-6\} \quad \text{ci}' \geq 0 \\
\{-7\} \quad \text{ck}' \geq 0 \\
\{-8\} \quad \text{oi}' \leq \text{ci}' \\
\{-9\} \quad \text{ci}' + \text{ck}' \leq \text{oi}' + \text{ok}' \\
\hline
\{1\} \quad \text{expt}(2, \text{ci}' - \text{oi}') \leq \text{expt}(2, \text{ok}') \\
\{2\} \quad \text{rem}(\text{expt}(2, \text{ck}')) \\
\quad (\text{ndiv}(\text{rem}(\text{expt}(2, \text{oi}'))(n') + \text{rem}(\text{expt}(2, \text{ok}'))(\text{ndiv}(m', \text{expt}(2, \text{oi}')))) \times \text{expt}(2, \text{oi}') - \text{rem}(\text{expt}(2, \text{oi}')) \\
\quad \quad \text{expt}(2, \text{ci}')) \\
\quad = \text{rem}(\text{expt}(2, \text{ck}'))(\text{ndiv}(m', \text{expt}(2, \text{ci}'))))
\end{array}$$

Hiding formulas: -1, 2,

Using lemma both_sides_expt_gt1_le,

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of cut_bits_overwrite_bits_contained.1.1.3.

cut_bits_overwrite_bits_contained.1.2:

$$\begin{array}{l}
\{-1\} \quad \text{rem}(\text{expt}(2, \text{ok}'))(\text{ndiv}(m', \text{expt}(2, \text{oi}')))) = \\
\quad \text{rem}(\text{expt}(2, \text{ci}' - \text{oi}'))(\text{rem}(\text{expt}(2, \text{ok}'))(\text{ndiv}(m', \text{expt}(2, \text{oi}')))) + \text{expt}(2, \text{ci}' - \text{oi}') \times \text{ndiv}(\text{rem}(\text{expt}(2, \text{ok}'))(\text{ndiv}(m', \text{expt}(2, \text{oi}')))) \\
\{-2\} \quad n' \geq 0 \\
\{-3\} \quad m' \geq 0 \\
\{-4\} \quad \text{oi}' \geq 0 \\
\{-5\} \quad \text{ok}' \geq 0 \\
\{-6\} \quad \text{ci}' \geq 0 \\
\{-7\} \quad \text{ck}' \geq 0 \\
\{-8\} \quad \text{oi}' \leq \text{ci}' \\
\{-9\} \quad \text{ci}' + \text{ck}' \leq \text{oi}' + \text{ok}' \\
\hline
\{1\} \quad \text{divides}(\text{expt}(2, \text{ci}' - \text{oi}'), \text{expt}(2, \text{ok}')) \\
\{2\} \quad \text{rem}(\text{expt}(2, \text{ck}')) \\
\quad (\text{ndiv}(\text{rem}(\text{expt}(2, \text{oi}'))(n') + \text{rem}(\text{expt}(2, \text{ok}'))(\text{ndiv}(m', \text{expt}(2, \text{oi}')))) \times \text{expt}(2, \text{oi}') - \text{rem}(\text{expt}(2, \text{oi}')) \\
\quad \quad \text{expt}(2, \text{ci}')) \\
\quad = \text{rem}(\text{expt}(2, \text{ck}'))(\text{ndiv}(m', \text{expt}(2, \text{ci}'))))
\end{array}$$

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of cut_bits_overwrite_bits_contained.1.2.

cut_bits_overwrite_bits_contained.2:

{-1}	$n' \geq 0$
{-2}	$m' \geq 0$
{-3}	$oi' \geq 0$
{-4}	$ok' \geq 0$
{-5}	$ci' \geq 0$
{-6}	$ck' \geq 0$
{-7}	$oi' \leq ci'$
{-8}	$ci' + ck' \leq oi' + ok'$
{1}	
{2}	$\begin{aligned} & \text{rem}(\text{expt}(2, ck')) \\ & \quad (\text{ndiv}(\text{rem}(\text{expt}(2, oi'))(n') + \text{rem}(\text{expt}(2, ok'))(\text{ndiv}(m', \text{expt}(2, oi')))) \times \text{expt}(2, oi') - \text{rem}(\text{expt}(2, ci'))) \\ & = \text{rem}(\text{expt}(2, ck'))(\text{ndiv}(m', \text{expt}(2, ci'))) \end{aligned}$

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of cut_bits_overwrite_bits_contained.2.

Q.E.D.

C.17.43 Bits.cut_bits_overwrite_shift_bits_disjoint

Terse proof for cut_bits_overwrite_shift_bits_disjoint.

cut_bits_overwrite_shift_bits_disjoint:

{1}	$\begin{aligned} & \forall (n, m, i_1, k_1, i_2, k_2: \text{nat}): \\ & \quad i_1 + k_1 \leq i_2 \vee i_2 + k_2 \leq i_1 \supset \\ & \quad \text{cut_bits}(\text{overwrite_shift_bits}(n, m, i_1, k_1), i_2, k_2) = \text{cut_bits}(n, i_2, k_2) \end{aligned}$
-----	--

Repeatedly Skolemizing and flattening,

Expanding the definition of overwrite_shift_bits,

Rewriting using cut_bits_overwrite_bits_disjoined, matching in *,

This completes the proof of cut_bits_overwrite_shift_bits_disjoint.

Q.E.D.

C.17.44 Bits.cut_bits_overwrite_shift_bits_contained_TCC1

Terse proof for cut_bits_overwrite_shift_bits_contained_TCC1.

cut_bits_overwrite_shift_bits_contained_TCC1:

{1}	$\forall (i_1, k_1, i_2, k_2: \text{nat}): i_1 \leq i_2 \wedge i_2 + k_2 \leq i_1 + k_1 \supset i_2 - i_1 \geq 0$
-----	---

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of cut_bits_overwrite_shift_bits_contained_TCC1.

Q.E.D.

C.17.45 Bits.cut_bits_overwrite_shift_bits_contained

Terse proof for cut_bits_overwrite_shift_bits_contained.

cut_bits_overwrite_shift_bits_contained:

$$\frac{\{1\} \quad \forall (n, m, i_1, k_1, i_2, k_2: \text{nat}): \quad i_1 \leq i_2 \wedge i_2 + k_2 \leq i_1 + k_1 \supset \quad \text{cut_bits}(\text{overwrite_shift_bits}(n, m, i_1, k_1), i_2, k_2) = \text{cut_bits}(m, i_2 - i_1, k_2)}{}$$

Repeatedly Skolemizing and flattening,
 Expanding the definition of overwrite_shift_bits,
 Rewriting using cut_bits_overwrite_bits_contained, matching in *,
 Rewriting using cut_bits_shift_bits_left, matching in *,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 Rewriting using zero_cut_bits, matching in *,
 This completes the proof of cut_bits_overwrite_shift_bits_contained.
 Q.E.D.

C.17.46 Bits.cut_bit_overwrite_shift_bits_TCC1

Terse proof for cut_bit_overwrite_shift_bits_TCC1.

cut_bit_overwrite_shift_bits_TCC1:

$$\frac{\{1\} \quad \forall (i_1, k_1, i_2: \text{nat}): \quad i_1 \leq i_2 \wedge i_2 < i_1 + k_1 \supset \quad i_2 - i_1 \geq 0}{}$$

Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of cut_bit_overwrite_shift_bits_TCC1.
 Q.E.D.

C.17.47 Bits.cut_bit_overwrite_shift_bits

Terse proof for cut_bit_overwrite_shift_bits.

cut_bit_overwrite_shift_bits:

$$\frac{\{1\} \quad \forall (n, m, i_1, k_1, i_2: \text{nat}): \quad \begin{array}{l} \text{cut_bit}(\text{overwrite_shift_bits}(n, m, i_1, k_1), i_2) = \\ \text{IF } i_1 \leq i_2 \wedge i_2 < i_1 + k_1 \\ \quad \text{THEN } \text{cut_bit}(m, i_2 - i_1) \\ \quad \text{ELSE } \text{cut_bit}(n, i_2) \\ \text{ENDIF} \end{array}}{}$$

Repeatedly Skolemizing and flattening,
 Lifting IF-conditions to the top level,
 Splitting conjunctions,
 we get 2 subgoals:

cut_bit_overwrite_shift_bits.1:

$$\frac{\begin{array}{l} \{-1\} \quad n' \geq 0 \\ \{-2\} \quad m' \geq 0 \\ \{-3\} \quad i'_1 \geq 0 \\ \{-4\} \quad k'_1 \geq 0 \\ \{-5\} \quad i'_2 \geq 0 \end{array} \quad \{1\} \quad i'_1 \leq i'_2 \wedge i'_2 < i'_1 + k'_1 \supset \quad \text{cut_bit}(\text{overwrite_shift_bits}(n', m', i'_1, k'_1), i'_2) = \text{cut_bit}(m', i'_2 - i'_1)}{}$$

Applying disjunctive simplification to flatten sequent,

C Proof scripts

Rewriting using `cut_bit_bits`, matching in *,
 Rewriting using `cut_bits_overwrite_shift_bits_contained`, matching in *,
 Rewriting using `cut_bit_bits`, matching in *,
 This completes the proof of `cut_bit_overwrite_shift_bits.1`.
`cut_bit_overwrite_shift_bits.2:`

$\begin{array}{l} \{-1\} \quad n' \geq 0 \\ \{-2\} \quad m' \geq 0 \\ \{-3\} \quad i'_1 \geq 0 \\ \{-4\} \quad k'_1 \geq 0 \\ \{-5\} \quad i'_2 \geq 0 \end{array}$	$\begin{array}{l} \{1\} \quad \neg (i'_1 \leq i'_2 \wedge i'_2 < i'_1 + k'_1) \supset \\ \quad \text{cut_bit}(\text{overwrite_shift_bits}(n', m', i'_1, k'_1), i'_2) = \text{cut_bit}(n', i'_2) \end{array}$
---	--

Applying disjunctive simplification to flatten sequent,
 Rewriting using `cut_bit_bits`, matching in *,
 Rewriting using `cut_bits_overwrite_shift_bits_disjoint`, matching in *,
 we get 2 subgoals:

`cut_bit_overwrite_shift_bits.2.1:`

$\begin{array}{l} \{-1\} \quad n' \geq 0 \\ \{-2\} \quad m' \geq 0 \\ \{-3\} \quad i'_1 \geq 0 \\ \{-4\} \quad k'_1 \geq 0 \\ \{-5\} \quad i'_2 \geq 0 \end{array}$	$\begin{array}{l} \{1\} \quad i'_1 \leq i'_2 \wedge i'_2 < i'_1 + k'_1 \\ \{2\} \quad \text{IF } \text{cut_bits}(n', i'_2, 1) = 1 \text{ THEN TRUE ELSE FALSE ENDIF} = \text{cut_bit}(n', i'_2) \end{array}$
---	--

Rewriting using `cut_bit_bits`, matching in *,
 This completes the proof of `cut_bit_overwrite_shift_bits.2.1`.
`cut_bit_overwrite_shift_bits.2.2:`

$\begin{array}{l} \{-1\} \quad n' \geq 0 \\ \{-2\} \quad m' \geq 0 \\ \{-3\} \quad i'_1 \geq 0 \\ \{-4\} \quad k'_1 \geq 0 \\ \{-5\} \quad i'_2 \geq 0 \end{array}$	$\begin{array}{l} \{1\} \quad i'_1 + k'_1 \leq i'_2 \vee 1 + i'_2 \leq i'_1 \\ \{2\} \quad i'_1 \leq i'_2 \wedge i'_2 < i'_1 + k'_1 \\ \{3\} \quad \text{IF } \text{cut_bits}(\text{overwrite_shift_bits}(n', m', i'_1, k'_1), i'_2, 1) = 1 \\ \quad \text{THEN TRUE} \\ \quad \text{ELSE FALSE} \\ \quad \text{ENDIF} \\ \quad = \text{cut_bit}(n', i'_2) \end{array}$
---	---

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 This completes the proof of `cut_bit_overwrite_shift_bits.2.2`.
 Q.E.D.

C.17.48 Bits.cut_bit_overwrite_bool_bit

Terse proof for `cut_bit_overwrite_bool_bit`.

cut_bit_overwrite_bool_bit:

$\{1\} \quad \forall (n: \text{nat}, b: \text{bool}, i, j: \text{nat}):$ $\text{cut_bit}(\text{overwrite_bool_bit}(n, b, i), j) =$ $\text{IF } i = j \text{ THEN } b \text{ ELSE } \text{cut_bit}(n, j) \text{ ENDIF}$
--

Repeatedly Skolemizing and flattening,

Lifting IF-conditions to the top level,

Splitting conjunctions,

we get 2 subgoals:

cut_bit_overwrite_bool_bit.1:

$\{-1\} \quad n' \geq 0$ $\{-2\} \quad i' \geq 0$ $\{-3\} \quad j' \geq 0$
$\{1\} \quad i' = j' \supset \text{cut_bit}(\text{overwrite_bool_bit}(n', b', i'), j') = b'$

Applying disjunctive simplification to flatten sequent,

Rewriting using cut_bit_bits, matching in *,

Expanding the definition of overwrite_bool_bit,

Expanding the definition of overwrite_bit,

Rewriting using cut_bits_overwrite_bits_contained, matching in *,

Rewriting using cut_bits_shift_bits_left, matching in *,

Simplifying, rewriting, and recording with decision procedures,

Case splitting on $j!1 - i!1 = 0$,

we get 2 subgoals:

cut_bit_overwrite_bool_bit.1.1:

$\{-1\} \quad j' - i' = 0$ $\{-2\} \quad i' = j'$ $\{-3\} \quad n' \geq 0$ $\{-4\} \quad i' \geq 0$ $\{-5\} \quad j' \geq 0$
$\{1\} \quad \text{IF } \text{cut_bits}(\text{bool_to_nat}(b'), j' - i', 1) = 1 \text{ THEN TRUE ELSE FALSE ENDIF} = b'$

Replacing using formula -1,

Hiding formulas: -1,

Expanding the definition of cut_bits,

Expanding the definition of expt,

Expanding the definition of expt,

Rewriting using ndiv_1, matching in *,

Rewriting using rem_mod, matching in *,

Expanding the definition of bool_to_nat,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of cut_bit_overwrite_bool_bit.1.1.

cut_bit_overwrite_bool_bit.1.2:

$\{-1\} \quad i' = j'$ $\{-2\} \quad n' \geq 0$ $\{-3\} \quad i' \geq 0$ $\{-4\} \quad j' \geq 0$
$\{1\} \quad j' - i' = 0$ $\{2\} \quad \text{IF } \text{cut_bits}(\text{bool_to_nat}(b'), j' - i', 1) = 1 \text{ THEN TRUE ELSE FALSE ENDIF} = b'$

Simplifying, rewriting, and recording with decision procedures,

C Proof scripts

This completes the proof of `cut_bit_overwrite_bool_bit.1.2`.

`cut_bit_overwrite_bool_bit.2`:

$$\frac{\begin{array}{l} \{-1\} \quad n' \geq 0 \\ \{-2\} \quad i' \geq 0 \\ \{-3\} \quad j' \geq 0 \end{array}}{\{1\} \quad \neg i' = j' \supset \text{cut_bit}(\text{overwrite_bool_bit}(n', b', i'), j') = \text{cut_bit}(n', j')}$$

Applying disjunctive simplification to flatten sequent,

Rewriting using `cut_bit_bits`, matching in `*`,

Rewriting using `cut_bit_bits`, matching in `*`,

Expanding the definition of `overwrite_bool_bit`,

Expanding the definition of `overwrite_bit`,

Rewriting using `cut_bits_overwrite_bits_disjoined`, matching in `*`,

Hiding formulas: 3,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `cut_bit_overwrite_bool_bit.2`.

Q.E.D.

C.17.49 Bits.overwrite_shift_bits_merge

Terse proof for `overwrite_shift_bits_merge`.

`overwrite_shift_bits_merge`:

$$\frac{\{1\} \quad \forall (m, ci1, ck1, si, ci2, ck2, os, ok: \text{nat}):}{\text{overwrite_shift_bits}(\text{shift_bits_left}(\text{cut_bits}(m, ci1, ck1), si), \text{cut_bits}(m, ci2, ck2), os, ok) = \text{shift_bits_left}(\text{cut_bits}(m, ci1, ck1 + ck2), si)}$$

Repeatedly Skolemizing and flattening,

Replacing using formula -11,

Replacing using formula -9,

Replacing using formula -10,

Hiding formulas: -9, -10, -11,

Case splitting on `ck2!1 = 0`,

we get 2 subgoals:

`overwrite_shift_bits_merge.1`:

$$\frac{\begin{array}{l} \{-1\} \quad ck2' = 0 \\ \{-2\} \quad m' \geq 0 \\ \{-3\} \quad ci1' \geq 0 \\ \{-4\} \quad ck1' \geq 0 \\ \{-5\} \quad si' \geq 0 \\ \{-6\} \quad ci1' + ck1' \geq 0 \\ \{-7\} \quad ck2' \geq 0 \\ \{-8\} \quad si' + ck1' \geq 0 \\ \{-9\} \quad ck2' \geq 0 \end{array}}{\{1\} \quad \text{overwrite_shift_bits}(\text{shift_bits_left}(\text{cut_bits}(m', ci1', ck1'), si'), \text{cut_bits}(m', ci1' + ck1', ck2'), si' + ck1', ck2') = \text{shift_bits_left}(\text{cut_bits}(m', ci1', ck1' + ck2'), si')}$$

Replacing using formula -1,

Case splitting on `cut_bits(m!1, ci1!1 + ck1!1, 0) = 0`,

we get 2 subgoals:

`overwrite_shift_bits_merge.1.1:`

{-1}	$\text{cut_bits}(m', \text{ci1}' + \text{ck1}', 0) = 0$
{-2}	$\text{ck2}' = 0$
{-3}	$m' \geq 0$
{-4}	$\text{ci1}' \geq 0$
{-5}	$\text{ck1}' \geq 0$
{-6}	$\text{si}' \geq 0$
{-7}	$\text{ci1}' + \text{ck1}' \geq 0$
{-8}	$0 \geq 0$
{-9}	$\text{si}' + \text{ck1}' \geq 0$
{-10}	$0 \geq 0$
{1} $\text{overwrite_shift_bits}(\text{shift_bits_left}(\text{cut_bits}(m', \text{ci1}', \text{ck1}'), \text{si}'),$	
	$\text{cut_bits}(m', \text{ci1}' + \text{ck1}', 0), \text{si}' + \text{ck1}', 0)$
	$= \text{shift_bits_left}(\text{cut_bits}(m', \text{ci1}', \text{ck1}' + 0), \text{si}')$

Replacing using formula -1,

Hiding formulas: -1,

Expanding the definition of `overwrite_shift_bits`,

Expanding the definition of `shift_bits_left`,

Rewriting using `zero_times1`, matching in *,

Expanding the definition of `overwrite_bits`,

Rewriting using `zero_cut_bits`, matching in *,

Expanding the definition of `shift_bits_left`,

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `overwrite_shift_bits_merge.1.1`.

`overwrite_shift_bits_merge.1.2:`

{-1}	$\text{ck2}' = 0$
{-2}	$m' \geq 0$
{-3}	$\text{ci1}' \geq 0$
{-4}	$\text{ck1}' \geq 0$
{-5}	$\text{si}' \geq 0$
{-6}	$\text{ci1}' + \text{ck1}' \geq 0$
{-7}	$0 \geq 0$
{-8}	$\text{si}' + \text{ck1}' \geq 0$
{-9}	$0 \geq 0$
{1} $\text{cut_bits}(m', \text{ci1}' + \text{ck1}', 0) = 0$	
{2}	$\text{overwrite_shift_bits}(\text{shift_bits_left}(\text{cut_bits}(m', \text{ci1}', \text{ck1}'), \text{si}'),$
	$\text{cut_bits}(m', \text{ci1}' + \text{ck1}', 0), \text{si}' + \text{ck1}', 0)$
	$= \text{shift_bits_left}(\text{cut_bits}(m', \text{ci1}', \text{ck1}' + 0), \text{si}')$

Hiding formulas: 2,

Rewriting using `zero_cut_bits`, matching in *,

This completes the proof of `overwrite_shift_bits_merge.1.2`.

overwrite_shift_bits_merge.2:

{-1}	$m' \geq 0$
{-2}	$ci1' \geq 0$
{-3}	$ck1' \geq 0$
{-4}	$si' \geq 0$
{-5}	$ci1' + ck1' \geq 0$
{-6}	$ck2' \geq 0$
{-7}	$si' + ck1' \geq 0$
{-8}	$ck2' \geq 0$
{1}	$ck2' = 0$
{2}	$\begin{aligned} & \text{overwrite_shift_bits}(\text{shift_bits_left}(\text{cut_bits}(m', ci1', ck1'), si'), \\ & \qquad \qquad \qquad \text{cut_bits}(m', ci1' + ck1', ck2'), si' + ck1', ck2') \\ & = \text{shift_bits_left}(\text{cut_bits}(m', ci1', ck1' + ck2'), si') \end{aligned}$

Expanding the definition of overwrite_shift_bits,
 Expanding the definition of overwrite_bits,
 Rewriting using cut_bits_shift_bits_left, matching in *,
 Simplifying, rewriting, and recording with decision procedures,
 Rewriting using cut_bits_cut_bits, matching in *,
 Rewriting using rem_mod, matching in *,
 we get 2 subgoals:

overwrite_shift_bits_merge.2.1:

{-1}	$m' \geq 0$
{-2}	$ci1' \geq 0$
{-3}	$ck1' \geq 0$
{-4}	$si' \geq 0$
{-5}	$ci1' + ck1' \geq 0$
{-6}	$ck2' \geq 0$
{-7}	$ck1' + si' \geq 0$
{-8}	$ck2' \geq 0$
{1}	$ck2' = 0$
{2}	$\begin{aligned} & \text{shift_bits_left}(\text{cut_bits}(m', ci1', ck1'), si') + \text{shift_bits_left}(\text{cut_bits}(m', ci1' + ck1', ck2'), ck1' + \\ & \qquad \qquad \qquad = \text{shift_bits_left}(\text{cut_bits}(m', ci1', ck1' + ck2'), si') \end{aligned}$

Rewriting using rem_mod, matching in *,
 we get 2 subgoals:

overwrite_shift_bits_merge.2.1.1:

{-1}	$m' \geq 0$
{-2}	$ci1' \geq 0$
{-3}	$ck1' \geq 0$
{-4}	$si' \geq 0$
{-5}	$ci1' + ck1' \geq 0$
{-6}	$ck2' \geq 0$
{-7}	$ck1' + si' \geq 0$
{-8}	$ck2' \geq 0$
{1}	$ck2' = 0$
{2}	$\begin{aligned} & \text{shift_bits_left}(\text{cut_bits}(m', ci1', ck1'), si') + \text{shift_bits_left}(\text{cut_bits}(m', ci1' + ck1', ck2'), ck1' + \\ & \qquad \qquad \qquad = \text{shift_bits_left}(\text{cut_bits}(m', ci1', ck1' + ck2'), si') \end{aligned}$

Simplifying, rewriting, and recording with decision procedures,
 Expanding the definition of shift_bits_left,
 Using lemma expt_plus,

Replacing using formula -1,
Hiding formulas: -1,
Rewriting using associative_mult, matching in *,
Applying both_sides_times1
Instantiating the top quantifier in -1 with the terms: $\text{expt}(2, \text{si}')$, $\text{expt}(2, \text{ck1}') \times \text{cut_bits}(m', \text{ci1}' + \text{ck1}', \text{ck2}') + \text{cut_bits}(m', \text{ci1}', \text{ck1}' + \text{ck2}')$,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Hiding formulas: 1, 4,
Expanding the definition of cut_bits,
Installing automatic rewrites from: `expt_divides both_sides_times_pos_lt1 ndiv_expt_expt`
Applying number_split
Instantiating the top quantifier in -1 with the terms: $\text{rem}(\text{expt}(2, \text{ck1}' + \text{ck2}')(\text{ndiv}(m', \text{expt}(2, \text{ci1}'))), \text{expt}(2, \text{ck1}'))$,
Replacing using formula -1,
Hiding formulas: -1,
Rewriting using ndiv_rem_divisible, matching in *,
we get 2 subgoals:
overwrite_shift_bits_merge.2.1.1.1:

$$\begin{array}{l|l}
\{-1\} & m' \geq 0 \\
\{-2\} & \text{ci1}' \geq 0 \\
\{-3\} & \text{ck1}' \geq 0 \\
\{-4\} & \text{si}' \geq 0 \\
\{-5\} & \text{ci1}' + \text{ck1}' \geq 0 \\
\{-6\} & \text{ck2}' \geq 0 \\
\{-7\} & \text{ck1}' + \text{si}' \geq 0 \\
\hline
\{1\} & \text{rem}(\text{expt}(2, \text{ck1}')(\text{ndiv}(m', \text{expt}(2, \text{ci1}')))) + \text{expt}(2, \text{ck1}') \times \text{rem}(\text{expt}(2, \text{ck2}')(\text{ndiv}(m', \text{expt}(2, \text{ci1}' + \text{ck1}')))) \\
& = \\
& \text{expt}(2, \text{ck1}') \times \text{rem}(\text{expt}(2, \text{ck2}')(\text{ndiv}(\text{ndiv}(m', \text{expt}(2, \text{ci1}')), \text{expt}(2, \text{ck1}')))) + \text{rem}(\text{expt}(2, \text{ck1}')(\text{rem}(\text{expt}(2, \text{ck1}' + \text{ck2}')(\text{ndiv}(m', \text{expt}(2, \text{ci1}'))))) \\
\{2\} & \text{ck2}' = 0
\end{array}$$

Rewriting using ndiv_times_2, matching in *,
Rewriting using expt_plus, matching in *,
Simplifying, rewriting, and recording with decision procedures,
Rewriting using rem_rem, matching in *,
This completes the proof of `overwrite_shift_bits_merge.2.1.1.1`.
overwrite_shift_bits_merge.2.1.1.2:

$$\begin{array}{l|l}
\{-1\} & m' \geq 0 \\
\{-2\} & \text{ci1}' \geq 0 \\
\{-3\} & \text{ck1}' \geq 0 \\
\{-4\} & \text{si}' \geq 0 \\
\{-5\} & \text{ci1}' + \text{ck1}' \geq 0 \\
\{-6\} & \text{ck2}' \geq 0 \\
\{-7\} & \text{ck1}' + \text{si}' \geq 0 \\
\hline
\{1\} & \text{expt}(2, \text{ck1}') \leq \text{expt}(2, \text{ck1}' + \text{ck2}') \\
\{2\} & \text{rem}(\text{expt}(2, \text{ck1}')(\text{ndiv}(m', \text{expt}(2, \text{ci1}')))) + \text{expt}(2, \text{ck1}') \times \text{rem}(\text{expt}(2, \text{ck2}')(\text{ndiv}(m', \text{expt}(2, \text{ci1}' + \text{ck1}')))) \\
& = \\
& \text{rem}(\text{expt}(2, \text{ck1}')(\text{rem}(\text{expt}(2, \text{ck1}' + \text{ck2}')(\text{ndiv}(m', \text{expt}(2, \text{ci1}'))))) + \text{expt}(2, \text{ck1}') \times \text{ndiv}(\text{rem}(\text{expt}(2, \text{ck1}' + \text{ck2}')(\text{ndiv}(m', \text{expt}(2, \text{ci1}'))))) \\
\{3\} & \text{ck2}' = 0
\end{array}$$

Hiding formulas: 2,
Using lemma both_sides_expt_gt1_le,
Simplifying, rewriting, and recording with decision procedures,

C Proof scripts

This completes the proof of `overwrite_shift_bits_merge.2.1.1.2`.

`overwrite_shift_bits_merge.2.1.2`:

{-1}	$m' \geq 0$
{-2}	$ci1' \geq 0$
{-3}	$ck1' \geq 0$
{-4}	$si' \geq 0$
{-5}	$ci1' + ck1' \geq 0$
{-6}	$ck2' \geq 0$
{-7}	$ck1' + si' \geq 0$
{-8}	$ck2' \geq 0$
{1}	$\text{shift_bits_left}(\text{cut_bits}(m', ci1', ck1'), si') < \text{expt}(2, ck1' + ck2' + si')$
{2}	$ck2' = 0$
{3}	$\text{shift_bits_left}(\text{cut_bits}(m', ci1' + ck1', ck2'), ck1' + si') + 2 \times \text{shift_bits_left}(\text{cut_bits}(m', ci1', ck1'), ck1' + si') = \text{shift_bits_left}(\text{cut_bits}(m', ci1', ck1' + ck2'), si')$

Hiding formulas: 3,

Expanding the definition of `shift_bits_left`,

Installing automatic rewrites from: `expt_plus`

Applying `both_sides_times_pos_lt1`

Instantiating the top quantifier in -1 with the terms: `expt(2, si')`, `cut_bits(m', ci1', ck1')`, `expt(2, ck1' + ck2')`,

Simplifying, rewriting, and recording with decision procedures,

Applying `both_sides_expt_gt1_le`

Instantiating the top quantifier in -1 with the terms: `2`, `ck1'`, `ck1' + ck2'`,

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `overwrite_shift_bits_merge.2.1.2`.

`overwrite_shift_bits_merge.2.2`:

{-1}	$m' \geq 0$
{-2}	$ci1' \geq 0$
{-3}	$ck1' \geq 0$
{-4}	$si' \geq 0$
{-5}	$ci1' + ck1' \geq 0$
{-6}	$ck2' \geq 0$
{-7}	$ck1' + si' \geq 0$
{-8}	$ck2' \geq 0$
{1}	$\text{shift_bits_left}(\text{cut_bits}(m', ci1', ck1'), si') < \text{expt}(2, ck1' + si')$
{2}	$ck2' = 0$
{3}	$\text{rem}(\text{expt}(2, ck1' + si'))(\text{shift_bits_left}(\text{cut_bits}(m', ci1', ck1'), si')) + \text{shift_bits_left}(\text{cut_bits}(m', ci1', ck1'), ck1' + si') = \text{shift_bits_left}(\text{cut_bits}(m', ci1', ck1' + ck2'), si')$

Hiding formulas: 3,

Expanding the definition of `shift_bits_left`,

Using lemma `expt_plus`,

Replacing using formula -1,

Hiding formulas: -1,

Using lemma `both_sides_times_pos_lt1`,

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `overwrite_shift_bits_merge.2.2`.

Q.E.D.

C.18 Proofs for BitwiseExpressions (expressions.pvs)

This theory contains no provable formal statements.

C.19 Proofs for Block_Disjoint_Rewrites (plain_memory_rewrites.pvs)

C.19.1 Block_Disjoint_Rewrites.different_registers_blocks_disjoint

Terse proof for different_registers_blocks_disjoint.

different_registers_blocks_disjoint:

$$\frac{\{1\} \quad \forall (\text{addr1}, \text{addr2}: \text{Address}, \text{sz1}, \text{sz2}: \text{nat}): \quad \text{type_of}(\text{addr1}) \neq \text{type_of}(\text{addr2}) \supset \text{blocks_disjoint}?(\text{addr1}, \text{sz1}, \text{addr2}, \text{sz2})}{}$$

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of different_registers_blocks_disjoint.
Q.E.D.

C.20 Proofs for BooleanConversions (conversions.pvs)

C.20.1 BooleanConversions.pointer_to_bool_TCC1

Terse proof for pointer_to_bool_TCC1.

pointer_to_bool_TCC1:

$$\frac{\{1\} \quad \forall (\text{p_typ}: \text{Cpp_Subtype}(\text{cv}(\text{pointer}?))): \quad \text{pointer}?(\text{cv_base}(\text{p_typ}))}{}$$

Repeatedly Skolemizing and flattening,
Rewriting using cv_base_result, matching in *,
Keeping (1) and hiding *,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of pointer_to_bool_TCC1.
Q.E.D.

C.21 Proofs for CatchAux (statements.pvs)

This theory contains no provable formal statements.

C.22 Proofs for Challenge_Device_Memory (device_memory.pvs)

C.22.1 Challenge_Device_Memory.pm_dev_plain

Terse proof for pm_dev_plain.

pm_dev_plain:

```

{1}  ∇ (device_read_side_effect,
      device_write_side_effect:
        [[Address, list[Byte], bool] →
         [Device_memory[Physical_memory, pm_phy, Device_state] →
          ExprResult[Device_memory, list[Byte]]]],
      pm: Plain_Memory[Device_memory]):
is_device_plain_memory?(device_read_side_effect, device_write_side_effect)(pm) ⊃
plain_memory?(pm_phy)

```

Expanding the definition of is_device_plain_memory?,
Repeatedly Skolemizing and flattening,
This completes the proof of pm_dev_plain.
Q.E.D.

C.22.2 Challenge_Device_Memory.pm_dev_states

Terse proof for pm_dev_states.

pm_dev_states:

```

{1}  ∇ (device_read_side_effect,
      device_write_side_effect:
        [[Address, list[Byte], bool] →
         [Device_memory[Physical_memory, pm_phy, Device_state] →
          ExprResult[Device_memory, list[Byte]]]],
      pm: Plain_Memory[Device_memory]):
is_device_plain_memory?(device_read_side_effect, device_write_side_effect)(pm) ⊃
pm'states = em_lift(pm_phy'states)

```

Expanding the definition of is_device_plain_memory?,
Repeatedly Skolemizing and flattening,
This completes the proof of pm_dev_states.
Q.E.D.

C.22.3 Challenge_Device_Memory.pm_dev_read

Terse proof for pm_dev_read.

pm_dev_read:

```

{1}  ∇ (device_read_side_effect,
      device_write_side_effect:
        [[Address, list[Byte], bool] →
         [Device_memory[Physical_memory, pm_phy, Device_state] →
          ExprResult[Device_memory, list[Byte]]]],
      pm: Plain_Memory[Device_memory]):
is_device_plain_memory?(device_read_side_effect, device_write_side_effect)(pm) ⊃
memory_read(pm'mem) =
(λ (a: Address):
 em_lift[Physical_memory, Device_state, Byte](memory_read(pm_phy'mem)(a)))

```

Expanding the definition of `is_device_plain_memory?`,
 Expanding the definition of `device_pm`,
 Repeatedly Skolemizing and flattening,
 Replacing using formula -1,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 This completes the proof of `pm_dev_read`.
 Q.E.D.

C.22.4 Challenge_Device_Memory.pm_dev_write

Terse proof for `pm_dev_write`.

`pm_dev_write`:

```
{1}  ∀ (device_read_side_effect,
      device_write_side_effect:
        [[Address, list[Byte], bool] →
         [Device_memory[Physical_memory, pm_phy, Device_state] →
          ExprResult[Device_memory, list[Byte]]]],
      pm: Plain_Memory[Device_memory]):
  is_device_plain_memory?(device_read_side_effect, device_write_side_effect)(pm) ⊃
  memory_write(pm' mem) =
  (λ (a: Address, b: Byte):
   em_lift[Physical_memory, Device_state, Unit](memory_write(pm_phy' mem)(a, b)))
```

Expanding the definition of `is_device_plain_memory?`,
 Repeatedly Skolemizing and flattening,
 Expanding the definition of `device_pm`,
 Replacing using formula -1,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 This completes the proof of `pm_dev_write`.
 Q.E.D.

C.22.5 Challenge_Device_Memory.pm_dev_read_side_effect

Terse proof for `pm_dev_read_side_effect`.

`pm_dev_read_side_effect`:

```
{1}  ∀ (device_read_side_effect,
      device_write_side_effect:
        [[Address, list[Byte], bool] →
         [Device_memory[Physical_memory, pm_phy, Device_state] →
          ExprResult[Device_memory, list[Byte]]]],
      pm: Plain_Memory[Device_memory]):
  is_device_plain_memory?(device_read_side_effect, device_write_side_effect)(pm) ⊃
  memory_read_side_effect(pm' mem) =
  (λ (a: Address, bl: list[Byte], cp: bool):
   em_lift[Physical_memory, Device_state, list[Byte]]
     (memory_read_side_effect(pm_phy' mem)(a, bl, cp))
   ## (λ (bl1: list[Byte]): device_read_side_effect(a, bl1, cp)))
```

Expanding the definition of `is_device_plain_memory?`,

Repeatedly Skolemizing and flattening,
 Replacing using formula -1,
 Expanding the definition of device_pm,
 which is trivially true.
 This completes the proof of pm_dev_read_side_effect.
 Q.E.D.

C.22.6 Challenge_Device_Memory.pm_dev_write_side_effect

Terse proof for pm_dev_write_side_effect.

pm_dev_write_side_effect:

```
{1}  ∃ (device_read_side_effect,
      device_write_side_effect:
        [[Address, list[Byte], bool] →
         [Device_memory[Physical_memory, pm_phy, Device_state] →
          ExprResult[Device_memory, list[Byte]]]],
      pm: Plain_Memory[Device_memory]):
is_device_plain_memory?(device_read_side_effect, device_write_side_effect)(pm) ⊃
memory_write_side_effect(pm mem) =
(λ (a: Address, bl: list[Byte], cp: bool):
  em_lift[Physical_memory, Device_state, list[Byte]]
    (memory_write_side_effect(pm_phy mem)(a, bl, cp))
  ## (λ (bl1: list[Byte]): device_write_side_effect(a, bl1, cp)))
```

Expanding the definition of is_device_plain_memory?,
 Repeatedly Skolemizing and flattening,
 Replacing using formula -1,
 Expanding the definition of device_pm,
 which is trivially true.
 This completes the proof of pm_dev_write_side_effect.
 Q.E.D.

C.22.7 Challenge_Device_Memory.pm_dev_ro_rw_addr

Terse proof for pm_dev_ro_rw_addr.

pm_dev_ro_rw_addr:

```
{1}  ∃ (device_read_side_effect,
      device_write_side_effect:
        [[Address, list[Byte], bool] →
         [Device_memory[Physical_memory, pm_phy, Device_state] →
          ExprResult[Device_memory, list[Byte]]]],
      pm: Plain_Memory[Device_memory]):
is_device_plain_memory?(device_read_side_effect, device_write_side_effect)(pm) ⊃
((pm ro_addr ∪ pm rw_addr) ⊆ (pm_phy ro_addr ∪ pm_phy rw_addr))
```

Expanding the definition of is_device_plain_memory?,
 Repeatedly Skolemizing and flattening,
 Keeping (-5 -6 1) and hiding *,
 Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of pm_dev_ro_rw_addr.
Q.E.D.

C.22.8 Challenge_Device_Memory.device_plain_transformers_ok

Terse proof for device_plain_transformers_ok.

device_plain_transformers_ok:

<pre> {1} ∃ (device_read_side_effect, device_write_side_effect: [[Address, list[Byte], bool] → [Device_memory[Physical_memory, pm_phy, Device_state] → ExprResult[Device_memory, list[Byte]]]], pm: Plain_Memory[Device_memory]): is_device_plain_memory?(device_read_side_effect, device_write_side_effect)(pm) ⊃ transformers_ok?(pm' states, (memory_read_transformers(pm'mem, (pm'ro_addr ∪ pm'rw_addr)) ∪ memory_write_t </pre>

Repeatedly Skolemizing and flattening,

Using lemma pm_dev_states,

Using lemma pm_dev_plain,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Rewriting using transformers_ok_union_transformers, matching in *,

we get 2 subgoals:

device_plain_transformers_ok.1:

<pre> {-1} is_device_plain_memory?(device_read_side_effect', device_write_side_effect')(pm') {-2} plain_memory?(pm_phy) {-3} pm' states = em_lift(pm_phy' states) </pre>
<pre> {1} transformers_ok?(pm' states, memory_read_transformers(pm'mem, (pm'ro_addr ∪ pm'rw_addr))) {2} transformers_ok?(pm' states, (memory_read_transformers(pm'mem, (pm'ro_addr ∪ pm'rw_addr)) ∪ memory_write_t </pre>

Hiding formulas: 2,

Using lemma pm_dev_read,

Using lemma em_memory_read_transformers[Physical_memory, Device_state],

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Replacing using formula -2,

Replacing using formula -5,

Rewriting using em_transformers_ok, matching in *,

Using lemma plain_memory_transformers_ok_read_ro_rw,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Keeping (-2 -5 1) and hiding *,

Rewriting using transformers_ok_mono_transformers, matching in * where transformers.2 gets memory_read_transformers(pm_phy'mem, union(pm_phy'ro_addr, pm_phy'rw_addr)), transformers.1 gets memory_read_transformers(pm_phy'mem, union(pm!1'ro_addr, pm!1'rw_addr)),

Hiding formulas: (-1 2),

Rewriting using memory_read_transformers_mono, matching in *,

Using lemma pm_dev_ro_rw_addr,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of device_plain_transformers_ok.1.

device_plain_transformers_ok.2:

{-1}	is_device_plain_memory?(device_read_side_effect', device_write_side_effect')(pm')
{-2}	plain_memory?(pm_phy)
{-3}	pm' states = em_lift(pm_phy' states)
{1}	transformers_ok?(pm' states, memory_write_transformers(pm' mem, pm' rw_addr))
{2}	transformers_ok?(pm' states, (memory_read_transformers(pm' mem, (pm' ro_addr \cup pm' rw_addr)) \cup mem

Hiding formulas: 2,

Using lemma pm_dev_write,

Using lemma em_memory_write_transformers[Physical_memory, Device_state],

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Replacing using formula -2,

Replacing using formula -5,

Rewriting using em_transformers_ok, matching in *,

Keeping (-3 -4 1) and hiding *,

Using lemma plain_memory_transformers_ok_write_rw,

Rewriting using transformers_ok_mono_transformers, matching in * where transformers_2 gets memory_write_transformers(pm_phy' mem, pm_phy' rw_addr), transformers_1 gets memory_write_transformers(pm_phy' mem, pm!1' rw_addr),

Rewriting using memory_write_transformers_mono, matching in *,

Expanding the definition of is_device_plain_memory?,

which is trivially true.

This completes the proof of device_plain_transformers_ok.2.

Q.E.D.

C.22.9 Challenge Device_Memory.device_plain_unchanged_memory_invariant

Terse proof for device_plain_unchanged_memory_invariant.

device_plain_unchanged_memory_invariant:

{1}	\forall (device_read_side_effect, device_write_side_effect: [[Address, list[Byte], bool] \rightarrow [Device_memory[Physical_memory, pm_phy, Device_state] \rightarrow ExprResult[Device_memory, list[Byte]]], pm: Plain_Memory[Device_memory]): is_device_plain_memory?(device_read_side_effect, device_write_side_effect)(pm) \supset unchanged_memory_invariant?(pm' mem, pm' states, (pm' other_actions \cup memory_read_transformers(pm' mem, (p (pm' ro_addr \cup pm' rw_addr))
-----	--

Repeatedly Skolemizing and flattening,

Rewriting using unchanged_memory_invariant_union_transformers, matching in * where pm gets pm!1' mem,

we get 2 subgoals:

device_plain_unchanged_memory_invariant.1:

{-1}	is_device_plain_memory?(device_read_side_effect', device_write_side_effect')(pm')
{1}	unchanged_memory_invariant?(pm' mem, pm' states, <div style="margin-left: 20px;">(pm' other_actions \cup memory_read_transformers(pm' mem, (pm' ro_addr \cup <div style="margin-left: 20px;">(pm' rw_addr))</div> </div>
{2}	unchanged_memory_invariant?(pm' mem, pm' states, <div style="margin-left: 20px;">((pm' other_actions \cup memory_read_transformers(pm' mem, (pm' ro_addr \cup <div style="margin-left: 20px;">(pm' rw_addr))</div> </div>

Hiding formulas: 2,

Using lemma pm_dev_plain,

Using lemma pm_dev_states,

Using lemma pm_dev_read,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Using lemma em_memory_read_transformers,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Replacing using formula -2,

Replacing using formula -4,

Rewriting using unchanged_memory_invariant_union_transformers, matching in *,

we get 2 subgoals:

device_plain_unchanged_memory_invariant.1.1:

{-1}	memory_read(pm' mem) = <div style="margin-left: 20px;">(λ (a: Address): <div style="margin-left: 20px;">em_lift[Physical_memory, Device_state, Byte](memory_read(pm_phy' mem)(a)))</div> </div>
{-2}	memory_read_transformers(pm' mem, (pm' ro_addr \cup pm' rw_addr)) = <div style="margin-left: 20px;">em_lift(memory_read_transformers(pm_phy' mem, (pm' ro_addr \cup pm' rw_addr)))</div>
{-3}	is_device_plain_memory?(device_read_side_effect', device_write_side_effect')(pm')
{-4}	pm' states = em_lift(pm_phy' states)
{-5}	plain_memory?(pm_phy)
{1}	unchanged_memory_invariant?(pm' mem, em_lift(pm_phy' states), pm' other_actions, <div style="margin-left: 20px;">(pm' ro_addr \cup pm' rw_addr))</div>
{2}	unchanged_memory_invariant?(pm' mem, em_lift(pm_phy' states), <div style="margin-left: 20px;">(pm' other_actions \cup em_lift(memory_read_transformers(pm_phy' mem, (pm <div style="margin-left: 20px;">(pm' ro_addr \cup pm' rw_addr))</div> </div>

Hiding formulas: 2,

Rewriting using unchanged_memory_invariant_mono, matching in * where transformers_1 gets pm!1'other_actions, transformers_2 gets em_lift[Physical_memory, Device_state](pm_phy'other_actions), addresses_1 gets union(pm!1'ro_addr, pm!1'rw_addr), addresses_2 gets union(pm_phy'ro_addr, pm_phy'rw_addr),

we get 3 subgoals:

device_plain_unchanged_memory_invariant.1.1.1:

{-1}	memory_read(pm' mem) = (λ (a: Address): em_lift[Physical_memory, Device_state, Byte](memory_read(pm_phy mem)(a)))
{-2}	memory_read_transformers(pm' mem, (pm' ro_addr ∪ pm' rw_addr)) = em_lift(memory_read_transformers(pm_phy mem, (pm' ro_addr ∪ pm' rw_addr)))
{-3}	is_device_plain_memory?(device_read_side_effect', device_write_side_effect')(pm')
{-4}	pm' states = em_lift(pm_phy states)
{-5}	plain_memory?(pm_phy)
{1}	(pm' other_actions ⊆ em_lift[Physical_memory, Device_state](pm_phy other_actions))
{2}	unchanged_memory_invariant?(pm' mem, em_lift(pm_phy states), pm' other_actions, (pm' ro_addr ∪ pm' rw_addr))

Keeping (-3 1) and hiding *,

Expanding the definition of is_device_plain_memory?,

which is trivially true.

This completes the proof of device_plain_unchanged_memory_invariant.1.1.1.

device_plain_unchanged_memory_invariant.1.1.2:

{-1}	memory_read(pm' mem) = (λ (a: Address): em_lift[Physical_memory, Device_state, Byte](memory_read(pm_phy mem)(a)))
{-2}	memory_read_transformers(pm' mem, (pm' ro_addr ∪ pm' rw_addr)) = em_lift(memory_read_transformers(pm_phy mem, (pm' ro_addr ∪ pm' rw_addr)))
{-3}	is_device_plain_memory?(device_read_side_effect', device_write_side_effect')(pm')
{-4}	pm' states = em_lift(pm_phy states)
{-5}	plain_memory?(pm_phy)
{1}	((pm' ro_addr ∪ pm' rw_addr) ⊆ (pm_phy ro_addr ∪ pm_phy rw_addr))
{2}	unchanged_memory_invariant?(pm' mem, em_lift(pm_phy states), pm' other_actions, (pm' ro_addr ∪ pm' rw_addr))

Keeping (-3 1) and hiding *,

Using lemma pm_dev_ro_rw_addr,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of device_plain_unchanged_memory_invariant.1.1.2.

device_plain_unchanged_memory_invariant.1.1.3:

{-1}	memory_read(pm' mem) = (λ (a: Address): em_lift[Physical_memory, Device_state, Byte](memory_read(pm_phy mem)(a)))
{-2}	memory_read_transformers(pm' mem, (pm' ro_addr ∪ pm' rw_addr)) = em_lift(memory_read_transformers(pm_phy mem, (pm' ro_addr ∪ pm' rw_addr)))
{-3}	is_device_plain_memory?(device_read_side_effect', device_write_side_effect')(pm')
{-4}	pm' states = em_lift(pm_phy states)
{-5}	plain_memory?(pm_phy)
{1}	unchanged_memory_invariant?(pm' mem, em_lift(pm_phy states), em_lift[Physical_memory, Device_state](pm_phy other_actions), (pm_phy ro_addr ∪ pm_phy rw_addr))
{2}	unchanged_memory_invariant?(pm' mem, em_lift(pm_phy states), pm' other_actions, (pm' ro_addr ∪ pm' rw_addr))

Using lemma em_unchanged_memory_invariant[Physical_memory, Device_state],

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Keeping (-5 1) and hiding *,

Using lemma plain_memory_unchanged_invariant,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Using lemma unchanged_memory_invariant_mono[Physical_memory],

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 2 subgoals:

device_plain_unchanged_memory_invariant.1.1.3.1:

{-1}	unchanged_memory_invariant?(pm_phy'mem, pm_phy'states, ((pm_phy'other_actions ∪ memory_read_transformers(pm_phy'mem, (pm_phy'ro_addr ∪ pm_phy'rw_addr)))
{-2}	plain_memory?(pm_phy)
{1}	((pm_phy'ro_addr ∪ pm_phy'rw_addr) ⊆ (pm_phy'ro_addr ∪ pm_phy'rw_addr))
{2}	unchanged_memory_invariant?(pm_phy'mem, pm_phy'states, pm_phy'other_actions, (pm_phy'ro_addr ∪ pm_phy'rw_addr))

Rewriting using subset_reflexive, matching in *,

This completes the proof of device_plain_unchanged_memory_invariant.1.1.3.1.

device_plain_unchanged_memory_invariant.1.1.3.2:

{-1}	unchanged_memory_invariant?(pm_phy'mem, pm_phy'states, ((pm_phy'other_actions ∪ memory_read_transformers(pm_phy'mem, (pm_phy'ro_addr ∪ pm_phy'rw_addr)))
{-2}	plain_memory?(pm_phy)
{1}	(pm_phy'other_actions ⊆ ((pm_phy'other_actions ∪ memory_read_transformers(pm_phy'mem, (pm_phy'ro_addr ∪ pm_phy'rw_addr)))
{2}	unchanged_memory_invariant?(pm_phy'mem, pm_phy'states, pm_phy'other_actions, (pm_phy'ro_addr ∪ pm_phy'rw_addr))

Keeping (1) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of device_plain_unchanged_memory_invariant.1.1.3.2.

device_plain_unchanged_memory_invariant.1.2:

{-1}	memory_read(pm'mem) = (λ (a: Address): em_lift[Physical_memory, Device_state, Byte](memory_read(pm_phy'mem)(a)))
{-2}	memory_read_transformers(pm'mem, (pm'ro_addr ∪ pm'rw_addr)) = em_lift(memory_read_transformers(pm_phy'mem, (pm'ro_addr ∪ pm'rw_addr)))
{-3}	is_device_plain_memory?(device_read_side_effect', device_write_side_effect')(pm')
{-4}	pm'states = em_lift(pm_phy'states)
{-5}	plain_memory?(pm_phy)
{1}	unchanged_memory_invariant?(pm'mem, em_lift(pm_phy'states), em_lift(memory_read_transformers(pm_phy'mem, (pm'ro_addr ∪ pm'rw_addr))), (pm'ro_addr ∪ pm'rw_addr))
{2}	unchanged_memory_invariant?(pm'mem, em_lift(pm_phy'states), (pm'other_actions ∪ em_lift(memory_read_transformers(pm_phy'mem, (pm'ro_addr ∪ pm'rw_addr))), (pm'ro_addr ∪ pm'rw_addr))

Using lemma em_unchanged_memory_invariant[Physical_memory, Device_state],

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Rewriting using unchanged_memory_invariant_mono, matching in * where transformers_1 gets memory_read_transformers(pm_phy'mem, union(pm!1'ro_addr, pm!1'rw_addr)), transformers_2 gets memory_read_transformers(pm_phy'mem, union(pm_phy'ro_addr, pm_phy'rw_addr)), addresses_1 gets union(pm!1'ro_addr, pm!1'rw_addr), addresses_2 gets union(pm_phy'ro_addr, pm_phy'rw_addr),

C Proof scripts

we get 3 subgoals:

`device_plain_unchanged_memory_invariant.1.2.1:`

{-1}	<code>memory_read(pm' mem) =</code> <code>(λ (a: Address):</code> <code>em_lift[Physical_memory, Device_state, Byte](memory_read(pm_phy' mem)(a)))</code>
{-2}	<code>memory_read_transformers(pm' mem, (pm'ro_addr ∪ pm'rw_addr)) =</code> <code>em_lift(memory_read_transformers(pm_phy' mem, (pm'ro_addr ∪ pm'rw_addr)))</code>
{-3}	<code>is_device_plain_memory?(device_read_side_effect', device_write_side_effect')(pm')</code>
{-4}	<code>pm' states = em_lift(pm_phy' states)</code>
{-5}	<code>plain_memory?(pm_phy)</code>
{1}	<code>(memory_read_transformers(pm_phy' mem, (pm'ro_addr ∪ pm'rw_addr)) ⊆ memory_read_transfor</code>
{2}	<code>unchanged_memory_invariant?(pm_phy' mem, pm_phy' states,</code> <code>memory_read_transformers(pm_phy' mem,</code> <code>(pm'ro_addr ∪ pm'rw_addr)),</code>
{3}	<code>unchanged_memory_invariant?(pm' mem, em_lift(pm_phy' states),</code> <code>em_lift(memory_read_transformers(pm_phy' mem,</code> <code>(pm'ro_addr ∪ pm'rw_addr))</code>
{4}	<code>unchanged_memory_invariant?(pm' mem, em_lift(pm_phy' states),</code> <code>(pm'other_actions ∪ em_lift(memory_read_transformers(pm_phy'</code> <code>(pm'ro_addr ∪ pm'rw_addr))</code>

Keeping (-3 1) and hiding *,

Rewriting using `memory_read_transformers_mono`, matching in *,

Using lemma `pm_dev_ro_rw_addr`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `device_plain_unchanged_memory_invariant.1.2.1`.

`device_plain_unchanged_memory_invariant.1.2.2:`

{-1}	<code>memory_read(pm' mem) =</code> <code>(λ (a: Address):</code> <code>em_lift[Physical_memory, Device_state, Byte](memory_read(pm_phy' mem)(a)))</code>
{-2}	<code>memory_read_transformers(pm' mem, (pm'ro_addr ∪ pm'rw_addr)) =</code> <code>em_lift(memory_read_transformers(pm_phy' mem, (pm'ro_addr ∪ pm'rw_addr)))</code>
{-3}	<code>is_device_plain_memory?(device_read_side_effect', device_write_side_effect')(pm')</code>
{-4}	<code>pm' states = em_lift(pm_phy' states)</code>
{-5}	<code>plain_memory?(pm_phy)</code>
{1}	<code>((pm'ro_addr ∪ pm'rw_addr) ⊆ (pm_phy'ro_addr ∪ pm_phy'rw_addr))</code>
{2}	<code>unchanged_memory_invariant?(pm_phy' mem, pm_phy' states,</code> <code>memory_read_transformers(pm_phy' mem,</code> <code>(pm'ro_addr ∪ pm'rw_addr)),</code>
{3}	<code>unchanged_memory_invariant?(pm' mem, em_lift(pm_phy' states),</code> <code>em_lift(memory_read_transformers(pm_phy' mem,</code> <code>(pm'ro_addr ∪ pm'rw_addr))</code>
{4}	<code>unchanged_memory_invariant?(pm' mem, em_lift(pm_phy' states),</code> <code>(pm'other_actions ∪ em_lift(memory_read_transformers(pm_phy'</code> <code>(pm'ro_addr ∪ pm'rw_addr))</code>

Keeping (-3 1) and hiding *,

Using lemma `pm_dev_ro_rw_addr`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `device_plain_unchanged_memory_invariant.1.2.2`.

`device_plain_unchanged_memory_invariant.1.2.3`:

<pre> {-1} memory_read(pm' mem) = (λ (a: Address): em_lift[Physical_memory, Device_state, Byte](memory_read(pm_phy' mem)(a))) {-2} memory_read_transformers(pm' mem, (pm' ro_addr ∪ pm' rw_addr)) = em_lift(memory_read_transformers(pm_phy' mem, (pm' ro_addr ∪ pm' rw_addr))) {-3} is_device_plain_memory?(device_read_side_effect', device_write_side_effect')(pm') {-4} pm' states = em_lift(pm_phy' states) {-5} plain_memory?(pm_phy) </pre>	<hr style="border: 0.5px solid black;"/> <pre> {1} unchanged_memory_invariant?(pm_phy' mem, pm_phy' states, memory_read_transformers(pm_phy' mem, (pm_phy' ro_addr ∪ pm_phy' rw_addr)), (pm_phy' ro_addr ∪ pm_phy' rw_addr)) {2} unchanged_memory_invariant?(pm_phy' mem, pm_phy' states, memory_read_transformers(pm_phy' mem, (pm' ro_addr ∪ pm' rw_addr)), (pm' ro_addr ∪ pm' rw_addr)) {3} unchanged_memory_invariant?(pm' mem, em_lift(pm_phy' states), em_lift(memory_read_transformers(pm_phy' mem, (pm' ro_addr ∪ pm' rw_addr))), (pm' ro_addr ∪ pm' rw_addr)) {4} unchanged_memory_invariant?(pm' mem, em_lift(pm_phy' states), (pm' other_actions ∪ em_lift(memory_read_transformers(pm_phy' mem, (pm' ro_addr ∪ pm' rw_addr))))) </pre>
--	--

Keeping (-5 1) and hiding *,

Using lemma `plain_memory_unchanged_invariant`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Using lemma `unchanged_memory_invariant_mono[Physical_memory]`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 2 subgoals:

`device_plain_unchanged_memory_invariant.1.2.3.1`:

<pre> {-1} unchanged_memory_invariant?(pm_phy' mem, pm_phy' states, ((pm_phy' other_actions ∪ memory_read_transformers(pm_phy' mem, (pm_phy' ro_addr ∪ pm_phy' rw_addr))) (pm_phy' ro_addr ∪ pm_phy' rw_addr)) {-2} plain_memory?(pm_phy) </pre>	<hr style="border: 0.5px solid black;"/> <pre> {1} ((pm_phy' ro_addr ∪ pm_phy' rw_addr) ⊆ (pm_phy' ro_addr ∪ pm_phy' rw_addr)) {2} unchanged_memory_invariant?(pm_phy' mem, pm_phy' states, memory_read_transformers(pm_phy' mem, (pm_phy' ro_addr ∪ pm_phy' rw_addr)), (pm_phy' ro_addr ∪ pm_phy' rw_addr)) </pre>
--	---

Rewriting using `subset_reflexive`, matching in *,

This completes the proof of `device_plain_unchanged_memory_invariant.1.2.3.1`.

C Proof scripts

device_plain_unchanged_memory_invariant.1.2.3.2:

{-1}	unchanged_memory_invariant?(pm_phy' mem, pm_phy' states, ((pm_phy' other_actions \cup memory_read_transformers(pm_phy' mem, (pm_phy' ro_addr \cup pm_phy' rw_addr)))
{-2}	plain_memory?(pm_phy)
{1}	(memory_read_transformers(pm_phy' mem, (pm_phy' ro_addr \cup pm_phy' rw_addr)) \subseteq ((pm_phy' other_actions \cup memory_read_transformers(pm_phy' mem, (pm_phy' ro_addr \cup pm_phy' rw_addr)))
{2}	unchanged_memory_invariant?(pm_phy' mem, pm_phy' states, memory_read_transformers(pm_phy' mem, (pm_phy' ro_addr \cup pm_phy' rw_addr))

Keeping (1) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of device_plain_unchanged_memory_invariant.1.2.3.2.

device_plain_unchanged_memory_invariant.2:

{-1}	is_device_plain_memory?(device_read_side_effect', device_write_side_effect')(pm')
{1}	unchanged_memory_invariant?(pm' mem, pm' states, memory_read_side_effect_super_transformers(pm' mem, (pm' ro_addr \cup pm' rw_addr))
{2}	unchanged_memory_invariant?(pm' mem, pm' states, ((pm' other_actions \cup memory_read_transformers(pm' mem, (pm' ro_addr \cup pm' rw_addr)))

Rewriting using unchanged_memory_invariant_union_transformers, matching in * where pm gets pm!1 mem,

we get 2 subgoals:

device_plain_unchanged_memory_invariant.2.1:

{-1}	is_device_plain_memory?(device_read_side_effect', device_write_side_effect')(pm')
{1}	unchanged_memory_invariant?(pm' mem, pm' states, memory_read_side_effect_super_transformers(pm' mem, (pm' ro_addr \cup pm' rw_addr))
{2}	unchanged_memory_invariant?(pm' mem, pm' states, memory_read_side_effect_super_transformers(pm' mem, (pm' ro_addr \cup pm' rw_addr))
{3}	unchanged_memory_invariant?(pm' mem, pm' states, ((pm' other_actions \cup memory_read_transformers(pm' mem, (pm' ro_addr \cup pm' rw_addr)))

Rewriting using unchanged_memory_invariant_all_transformers, matching in *,

Expanding the definition of memory_read_side_effect_super_transformers,

Repeatedly Skolemizing and flattening,

Replacing using formula -3,

Keeping (-2 -4 1) and hiding *,

Using lemma pm_dev_read_side_effect,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Replacing using formula -2,

Simplifying, rewriting, and recording with decision procedures,

Using lemma pm_dev_states,

Using lemma pm_dev_plain,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Using lemma `fexpr_unchanged_memory_invariant_composition`[Device_memory, list[Byte], list[Byte]],

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting, we get 5 subgoals:

`device_plain_unchanged_memory_invariant.2.1.1:`

<pre> {-1} is_device_plain_memory?(device_read_side_effect', device_write_side_effect')(pm') {-2} plain_memory?(pm_phy) {-3} pm' 'states = em_lift(pm_phy' states) {-4} memory_read_side_effect(pm' 'mem) = (λ (a: Address, bl: list[Byte], cp: bool): em_lift[Physical_memory, Device_state, list[Byte]] (memory_read_side_effect(pm_phy' mem)(a, bl, cp)) ## (λ (bl1: list[Byte]): device_read_side_effect'(a, bl1, cp))) {-5} (address_block(a', length(bl')) ⊆ (pm' 'ro_addr ∪ pm' 'rw_addr)) </pre>	<pre> </pre>
<pre> {1} ∀ (d: (λ (bl: list[Byte]): bl = bl')): unchanged_memory_invariant?(pm' 'mem, pm' 'states, singleton(expr_2_super(device_read_side_effect'(a', </pre>	<pre> </pre>
<pre> {2} unchanged_memory_invariant?(pm' 'mem, pm' 'states, singleton(expr_2_super(em_lift </pre>	<pre> [Physical_memory, Device_state, list[Byte]] (memory_read_side_effect(pm_phy' mem) (a', bl', cp')) ## (λ (bl1: list[Byte]): device_read_side_effect'(a', </pre>
<pre> (pm' 'ro_addr ∪ pm' 'rw_addr)) </pre>	<pre> d, cp'))), bl1, cp'))), </pre>

Repeatedly Skolemizing and flattening,

Hiding formulas: (-1 2),

Expanding the definition of `is_device_plain_memory?`,

Applying disjunctive simplification to flatten sequent,

Using lemma `unchanged_memory_invariant_mono`,

Installing automatic rewrites from: `subset_reflexive`

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Keeping (-2 -13 1) and hiding *,

Expanding the definition of `drse_super_transformers`,

Expanding the definition of `subset?`,

Repeatedly Skolemizing and flattening,

Expanding the definition of `member`,

Expanding the definition of `union`,

Expanding the definition of `member`,

Applying disjunctive simplification to flatten sequent,

Instantiating quantified variables,

Expanding the definition of `singleton`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

C Proof scripts

This completes the proof of `device_plain_unchanged_memory_invariant.2.1.1`.

`device_plain_unchanged_memory_invariant.2.1.2`:

```

{-1} is_device_plain_memory?(device_read_side_effect', device_write_side_effect')(pm')
{-2} plain_memory?(pm_phy)
{-3} pm' states = em_lift(pm_phy' states)
{-4} memory_read_side_effect(pm' mem) =
      (λ (a: Address, bl: list[Byte], cp: bool):
         em_lift[Physical_memory, Device_state, list[Byte]]
           (memory_read_side_effect(pm_phy' mem)(a, bl, cp))
           ## (λ (bl1: list[Byte]): device_read_side_effect'(a, bl1, cp)))
{-5} (address_block(a', length(bl')) ⊆ (pm' ro_addr ∪ pm' rw_addr))
-----
{1}  ∀ (s: (pm' states)):
      OK?(em_lift[Physical_memory, Device_state, list[Byte]]
           (memory_read_side_effect(pm_phy' mem)(a', bl', cp'))(s))
      ⊃
      data(em_lift[Physical_memory, Device_state, list[Byte]]
           (memory_read_side_effect(pm_phy' mem)(a', bl', cp'))(s))
      = bl'
{2}  unchanged_memory_invariant?(pm' mem, pm' states,
                                  singleton(expr_2_super(em_lift
                                                           [Physical_memory,
                                                            Device_state,
                                                            list[Byte]]
                                                           (memory_read_side_effect(pm
                                                                 (a', bl', cp'))
                                                           ##
                                                           (λ (bl1: list[Byte]):
                                                                device_read_side_effect'(a',
                                                                 (pm' ro_addr ∪ pm' rw_addr))

```

Repeatedly Skolemizing and flattening,

Hiding formulas: (-6 2),

Using lemma `plain_memory_side_effect_content_unchanged_read_block`,

Expanding the definition of `em_lift`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 2 subgoals:

device_plain_unchanged_memory_invariant.2.1.2.1:

{-1}	plain_memory?(pm_phy)
{-2}	side_effect_content_unchanged(address_block(a', length(bl')), pm_phy'states, memory_read_side_effect(pm_phy'mem))
{-3}	pm' 'states(s')
{-4}	OK?(memory_read_side_effect(pm_phy'mem)(a', bl', cp')(s' 'state))
{-5}	OK?(OK(s' WITH [(state) := state(memory_read_side_effect(pm_phy'mem)(a', bl', cp')(s' 'state))], data(memory_read_side_effect(pm_phy'mem)(a', bl', cp')(s' 'state))))
{-6}	is_device_plain_memory?(device_read_side_effect', device_write_side_effect')(pm')
{-7}	pm' 'states = (λ (em: Expand_state[Physical_memory, Device_state]): pm_phy'states(em 'state))
{-8}	(address_block(a', length(bl')) ⊆ (pm' 'ro_addr ∪ pm' 'rw_addr))
{1}	data(memory_read_side_effect(pm_phy'mem)(a', bl', cp')(s' 'state)) = bl'

Replacing using formula -7,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Installing automatic rewrites from: subset_reflexive

Using lemma side_effect_content_unchanged_content,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of device_plain_unchanged_memory_invariant.2.1.2.1.

device_plain_unchanged_memory_invariant.2.1.2.2:

{-1}	plain_memory?(pm_phy)
{-2}	pm' 'states(s')
{-3}	OK?(memory_read_side_effect(pm_phy'mem)(a', bl', cp')(s' 'state))
{-4}	OK?(OK(s' WITH [(state) := state(memory_read_side_effect(pm_phy'mem)(a', bl', cp')(s' 'state))], data(memory_read_side_effect(pm_phy'mem)(a', bl', cp')(s' 'state))))
{-5}	is_device_plain_memory?(device_read_side_effect', device_write_side_effect')(pm')
{-6}	pm' 'states = (λ (em: Expand_state[Physical_memory, Device_state]): pm_phy'states(em 'state))
{-7}	(address_block(a', length(bl')) ⊆ (pm' 'ro_addr ∪ pm' 'rw_addr))
{1}	(address_block(a', length(bl')) ⊆ (pm_phy'ro_addr ∪ pm_phy'rw_addr))
{2}	data(memory_read_side_effect(pm_phy'mem)(a', bl', cp')(s' 'state)) = bl'

Rewriting using subset_transitive, matching in * where a gets address_block(a!1, length(bl!1)), b gets union(pm!1'ro_addr, pm!1'rw_addr), c gets union(pm_phy'ro_addr, pm_phy'rw_addr),

Using lemma pm_dev_ro_rw_addr,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of device_plain_unchanged_memory_invariant.2.1.2.2.

device_plain_unchanged_memory_invariant.2.1.3:

```

{-1} is_device_plain_memory?(device_read_side_effect', device_write_side_effect')(pm')
{-2} plain_memory?(pm_phy)
{-3} pm'`states = em_lift(pm_phy'`states)
{-4} memory_read_side_effect(pm'`mem) =
      (λ (a: Address, bl: list[Byte], cp: bool):
         em_lift[Physical_memory, Device_state, list[Byte]]
           (memory_read_side_effect(pm_phy'`mem)(a, bl, cp))
         ## (λ (bl1: list[Byte]): device_read_side_effect'(a, bl1, cp)))
{-5} (address_block(a', length(bl')) ⊆ (pm'`ro_addr ∪ pm'`rw_addr))
-----
{1}  unchanged_memory_invariant?(pm'`mem, pm'`states,
      singleton(expr_2_super(em_lift
                                                                    [Physical_memory,
                                                                    Device_state,
                                                                    list[Byte]]
                                                                    (memory_read_side_effect(pm.
                                                                    (a', bl', cp')))),
                                                                    (pm'`ro_addr ∪ pm'`rw_addr))
{2}  unchanged_memory_invariant?(pm'`mem, pm'`states,
      singleton(expr_2_super(em_lift
                                                                    [Physical_memory,
                                                                    Device_state,
                                                                    list[Byte]]
                                                                    (memory_read_side_effect(pm.
                                                                    (a', bl', cp'))
                                                                    ##
                                                                    (λ (bl1: list[Byte]):
                                                                    device_read_side_effect'(a',
                                                                    (pm'`ro_addr ∪ pm'`rw_addr))

```

Rewriting using em_lift_expr_2_super, matching in *,

Hiding formulas: 2,

Rewriting using em_lift_singleton, matching in *,

Replacing using formula -3,

Using lemma pm_dev_read,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Using lemma em_unchanged_memory_invariant[Physical_memory, Device_state],

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Keeping (-2 -3 -6 1) and hiding *,

Using lemma plain_memory_unchanged_read_side_effect_block_ro_rw,

Using lemma unchanged_memory_invariant_mono,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 3 subgoals:

device_plain_unchanged_memory_invariant.2.1.3.3:

{-1}	unchanged_memory_invariant?(pm_phy' mem, pm_phy' states, memory_read_side_effect_super_transformers(pm_phy' mem, ad- dress_block (pm_phy'ro_addr ∪ pm_phy'rw_addr)) (a', length
{-2}	plain_memory?(pm_phy)
{-3}	is_device_plain_memory?(device_read_side_effect', device_write_side_effect')(pm')
{-4}	(address_block(a', length(bl')) ⊆ (pm'ro_addr ∪ pm'rw_addr))
{1}	(singleton(expr_2_super(memory_read_side_effect(pm_phy' mem)(a', bl', cp'))) ⊆ memory_read_side
{2}	unchanged_memory_invariant?(pm_phy' mem, pm_phy' states, singleton(expr_2_super(memory_read_side_effect(pm_phy' mem) (a', bl', cp'))), (pm'ro_addr ∪ pm'rw_addr))

Keeping (1) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of device_plain_unchanged_memory_invariant.2.1.3.3.

device_plain_unchanged_memory_invariant.2.1.4:

{-1}	is_device_plain_memory?(device_read_side_effect', device_write_side_effect')(pm')
{-2}	plain_memory?(pm_phy)
{-3}	pm' states = em_lift(pm_phy' states)
{-4}	memory_read_side_effect(pm' mem) = (λ (a: Address, bl: list[Byte], cp: bool): em_lift[Physical_memory, Device_state, list[Byte]] (memory_read_side_effect(pm_phy' mem)(a, bl, cp)) ## (λ (bl1: list[Byte]): device_read_side_effect'(a, bl1, cp)))
{-5}	(address_block(a', length(bl')) ⊆ (pm'ro_addr ∪ pm'rw_addr))
{1}	transformers_ok?(pm' states, memory_read_transformers(pm' mem, (pm'ro_addr ∪ pm'rw_addr)))
{2}	unchanged_memory_invariant?(pm' mem, pm' states, singleton(expr_2_super(em_lift [Physical_memory, Device_state, list[Byte]] (memory_read_side_effect(pm (a', bl', cp')) ## (λ (bl1: list[Byte]): device_read_side_effect'(a', (pm'ro_addr ∪ pm'rw_addr))

Using lemma device_plain_transformers_ok,

Using lemma transformers_ok_mono_transformers,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Keeping (1) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of device_plain_unchanged_memory_invariant.2.1.4.

device_plain_unchanged_memory_invariant.2.1.5:

{-1}	is_device_plain_memory?(device_read_side_effect', device_write_side_effect')(pm')
{-2}	plain_memory?(pm_phy)
{-3}	pm' states = em_lift(pm_phy' states)
{-4}	memory_read_side_effect(pm' mem) = $(\lambda (a: \text{Address}, bl: \text{list}[\text{Byte}], cp: \text{bool}):$ em_lift[Physical_memory, Device_state, list[Byte]] (memory_read_side_effect(pm_phy' mem)(a, bl, cp)) ## $(\lambda (bl1: \text{list}[\text{Byte}]): \text{device_read_side_effect}'(a, bl1, cp))$)
{-5}	(address_block(a', length(bl')) \subseteq (pm' ro_addr \cup pm' rw_addr))
{1}	singleton(expr_2_super(em_lift[Physical_memory, Device_state, list[Byte]] (memory_read_side_effect(pm_phy' mem)(a', bl', cp')))) (expr_2_super(em_lift[Physical_memory, Device_state, list[Byte]] (memory_read_side_effect(pm_phy' mem)(a', bl', cp'))))
{2}	unchanged_memory_invariant?(pm' mem, pm' states, singleton(expr_2_super(em_lift [Physical_memory, Device_state, list[Byte]] (memory_read_side_effect(pm_phy' mem) (a', bl', cp')) ## $(\lambda (bl1: \text{list}[\text{Byte}]):$ device_read_side_effect'(a', bl1, cp'))), (pm' ro_addr \cup pm' rw_addr))

Expanding the definition of singleton,
which is trivially true.

This completes the proof of device_plain_unchanged_memory_invariant.2.1.5.

device_plain_unchanged_memory_invariant.2.2:

{-1}	is_device_plain_memory?(device_read_side_effect', device_write_side_effect')(pm')
{1}	unchanged_memory_invariant?(pm' mem, pm' states, memory_write_side_effect_super_transformers(pm' mem, pm' rw_addr), (pm' ro_addr \cup pm' rw_addr))
{2}	unchanged_memory_invariant?(pm' mem, pm' states, (memory_read_side_effect_super_transformers(pm' mem, (pm' ro_addr \cup pm' (pm' ro_addr \cup pm' rw_addr))
{3}	unchanged_memory_invariant?(pm' mem, pm' states, ((pm' other_actions \cup memory_read_transformers(pm' mem, (pm' ro_addr \cup (pm' ro_addr \cup pm' rw_addr))

Rewriting using unchanged_memory_invariant_all_transformers, matching in *,
Expanding the definition of memory_write_side_effect_super_transformers,
Repeatedly Skolemizing and flattening,
Replacing using formula -3,
Keeping (-2 -4 1) and hiding *,
Using lemma pm_dev_write_side_effect,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Replacing using formula -2,
Simplifying, rewriting, and recording with decision procedures,

C Proof scripts

Using lemma `pm_dev_states`,
Using lemma `pm_dev_plain`,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Using lemma `fexpr_unchanged_memory_invariant_composition` [Device_memory, list[Byte],
list[Byte]],
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
we get 5 subgoals:

`device_plain_unchanged_memory_invariant.2.2.1:`

<pre> {-1} is_device_plain_memory?(device_read_side_effect', device_write_side_effect')(pm') {-2} plain_memory?(pm_phy) {-3} pm'`states = em_lift(pm_phy`states) {-4} memory_write_side_effect(pm'`mem) = (λ (a: Address, bl: list[Byte], cp: bool): em_lift [Physical_memory, Device_state, list[Byte]] (memory_write_side_effect(pm_phy`mem)(a, bl, cp)) ## (λ (bl1: list[Byte]): device_write_side_effect'(a, bl1, cp))) {-5} (address_block(a', length(bl')) ⊆ pm'`rw_addr) </pre>	<pre> {1} ∀ (d: (λ (bl: list[Byte]): bl = bl')): unchanged_memory_invariant?(pm'`mem, pm'`states, singleton(expr_2_super(device_write_side_effect'(a', (pm'`ro_addr ∪ pm'`rw_addr)) (pm'`ro_addr ∪ pm'`rw_addr))) {2} unchanged_memory_invariant?(pm'`mem, pm'`states, singleton(expr_2_super(em_lift [Physical_memory, Device_state, list[Byte]] (memory_write_side_effect(pm (a', bl', cp')) ## (λ (bl1: list[Byte]): device_write_side_effect'(a', (pm'`ro_addr ∪ pm'`rw_addr)) </pre>
--	---

Repeatedly Skolemizing and flattening,
Hiding formulas: (-1 2),
Expanding the definition of `is_device_plain_memory?`,
Applying disjunctive simplification to flatten sequent,
Using lemma `unchanged_memory_invariant_mono`,
Installing automatic rewrites from: `subset_reflexive`
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Keeping (-2 -13 1) and hiding *,
Expanding the definition of `subset?`,
Expanding the definition of `union`,
Expanding the definition of `member`,
Repeatedly Skolemizing and flattening,
Expanding the definition of `dwse_super_transformers`,
Instantiating quantified variables,
Expanding the definition of `singleton`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `device_plain_unchanged_memory_invariant.2.2.1`.

`device_plain_unchanged_memory_invariant.2.2.2`:

{-1}	<code>is_device_plain_memory?(device_read_side_effect', device_write_side_effect')(pm')</code>	
{-2}	<code>plain_memory?(pm_phy)</code>	
{-3}	<code>pm' states = em_lift(pm_phy states)</code>	
{-4}	<code>memory_write_side_effect(pm' mem) =</code> $(\lambda (a: \text{Address}, bl: \text{list}[\text{Byte}], cp: \text{bool}):$ $\text{em_lift}[\text{Physical_memory}, \text{Device_state}, \text{list}[\text{Byte}]]$ $(\text{memory_write_side_effect}(pm_phy' \text{ mem})(a, bl, cp))$ $\#\# (\lambda (bl1: \text{list}[\text{Byte}]): \text{device_write_side_effect}'(a, bl1, cp)))$	
{-5}	<code>(address_block(a', length(bl')) \subseteq pm'rw_addr)</code>	
{1}	$\forall (s: (pm' states)):$ $\text{OK?}(\text{em_lift}[\text{Physical_memory}, \text{Device_state}, \text{list}[\text{Byte}]]$ $(\text{memory_write_side_effect}(pm_phy' \text{ mem})(a', bl', cp'))(s))$ \supset $\text{data}(\text{em_lift}[\text{Physical_memory}, \text{Device_state}, \text{list}[\text{Byte}]]$ $(\text{memory_write_side_effect}(pm_phy' \text{ mem})(a', bl', cp'))(s))$ $= bl'$	
{2}	<code>unchanged_memory_invariant?(pm' mem, pm' states,</code> $\text{singleton}(\text{expr_2_super}(\text{em_lift}$ $[\text{Physical_memory},$ $\text{Device_state},$ $\text{list}[\text{Byte}]]$ $(\text{memory_write_side_effect}(pm_phy' \text{ mem}$ $(a', bl', cp'))$ $\#\#$ $(\lambda (bl1: \text{list}[\text{Byte}]):$ $\text{device_write_side_effect}'(a',$ $bl1,$ $cp'))))$ $(pm'ro_addr \cup pm'rw_addr))$	

Repeatedly Skolemizing and flattening,

Hiding formulas: (-6 2),

Using lemma `plain_memory_side_effect_content_unchanged_write_block`,

Expanding the definition of `em_lift`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 2 subgoals:

device_plain_unchanged_memory_invariant.2.2.2.1:

{-1}	plain_memory?(pm_phy)
{-2}	side_effect_content_unchanged(address_block(a', length(bl')), pm_phy' states, memory_write_side_effect(pm_phy' mem))
{-3}	pm' states(s')
{-4}	OK?(memory_write_side_effect(pm_phy' mem)(a', bl', cp')(s' state))
{-5}	OK?(OK(s' WITH [(state) := state(memory_write_side_effect(pm_phy' mem) (a', bl', cp')(s' state))], data(memory_write_side_effect(pm_phy' mem)(a', bl', cp')(s' state))))
{-6}	is_device_plain_memory?(device_read_side_effect', device_write_side_effect')(pm')
{-7}	pm' states = (λ (em: Expand_state[Physical_memory, Device_state]): pm_phy' states(em' state))
{-8}	(address_block(a', length(bl')) ⊆ pm'rw_addr)
{1}	data(memory_write_side_effect(pm_phy' mem)(a', bl', cp')(s' state)) = bl'

Replacing using formula -7,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Installing automatic rewrites from: subset_reflexive

Using lemma side_effect_content_unchanged_content,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of device_plain_unchanged_memory_invariant.2.2.2.1.

device_plain_unchanged_memory_invariant.2.2.2.2:

{-1}	plain_memory?(pm_phy)
{-2}	pm' states(s')
{-3}	OK?(memory_write_side_effect(pm_phy' mem)(a', bl', cp')(s' state))
{-4}	OK?(OK(s' WITH [(state) := state(memory_write_side_effect(pm_phy' mem) (a', bl', cp')(s' state))], data(memory_write_side_effect(pm_phy' mem)(a', bl', cp')(s' state))))
{-5}	is_device_plain_memory?(device_read_side_effect', device_write_side_effect')(pm')
{-6}	pm' states = (λ (em: Expand_state[Physical_memory, Device_state]): pm_phy' states(em' state))
{-7}	(address_block(a', length(bl')) ⊆ pm'rw_addr)
{1}	(address_block(a', length(bl')) ⊆ pm'rw_addr)
{2}	data(memory_write_side_effect(pm_phy' mem)(a', bl', cp')(s' state)) = bl'

Rewriting using subset_transitive, matching in * where a gets address_block(a!1, length(bl!1)), b gets pm!1'rw_addr, c gets pm_phy'rw_addr,

Expanding the definition of is_device_plain_memory?,

which is trivially true.

This completes the proof of device_plain_unchanged_memory_invariant.2.2.2.2.

device_plain_unchanged_memory_invariant.2.2.3:

{-1}	is_device_plain_memory?(device_read_side_effect', device_write_side_effect')(pm')	
{-2}	plain_memory?(pm_phy)	
{-3}	pm' states = em_lift(pm_phy' states)	
{-4}	memory_write_side_effect(pm' mem) = (λ (a: Address, bl: list[Byte], cp: bool): em_lift[Physical_memory, Device_state, list[Byte]] (memory_write_side_effect(pm_phy' mem)(a, bl, cp)) ## (λ (bl1: list[Byte]): device_write_side_effect'(a, bl1, cp)))	
{-5}	(address_block(a', length(bl')) ⊆ pm'rw_addr)	
{1}	unchanged_memory_invariant?(pm' mem, pm' states, singleton(expr_2_super(em_lift [Physical_memory, Device_state, list[Byte]] (memory_write_side_effect(pm_phy' mem (a', bl', cp')))), (pm'ro_addr ∪ pm'rw_addr))	
{2}	unchanged_memory_invariant?(pm' mem, pm' states, singleton(expr_2_super(em_lift [Physical_memory, Device_state, list[Byte]] (memory_write_side_effect(pm_phy' mem (a', bl', cp')) ## (λ (bl1: list[Byte]): device_write_side_effect'(a', bl1, cp')))) (pm'ro_addr ∪ pm'rw_addr))	

Hiding formulas: 2,

Rewriting using em_lift_expr_2_super, matching in *,

Rewriting using em_lift_singleton, matching in *,

Replacing using formula -3,

Using lemma pm_dev_read,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Using lemma em_unchanged_memory_invariant[Physical_memory, Device_state],

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Keeping (-2 -3 -6 1) and hiding *,

Using lemma plain_memory_unchanged_write_side_effect_block_ro_rw,

Using lemma unchanged_memory_invariant_mono,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 3 subgoals:

C Proof scripts

device_plain_unchanged_memory_invariant.2.2.3.1:

{-1}	plain_memory?(pm_phy)
{-2}	is_device_plain_memory?(device_read_side_effect', device_write_side_effect')(pm')
{-3}	(address_block(a', length(bl')) \subseteq pm'rw_addr)
{1}	unchanged_memory_invariant?(pm_phy'mem, pm_phy'states, memory_write_side_effect_super_transformers(pm_phy'mem, ad- dress_block (a', length(b (pm_phy'ro_addr \cup pm_phy'rw_addr))
{2}	unchanged_memory_invariant?(pm_phy'mem, pm_phy'states, singleton(expr_2_super(memory_write_side_effect(pm_phy'mem) (a', bl', cp'))), (pm'ro_addr \cup pm'rw_addr))
{3}	(address_block(a', length(bl')) \subseteq pm_phy'rw_addr)

Rewriting using subset_transitive, matching in * where a gets address_block(a!1, length(bl!1)), b gets pm!1'rw_addr, c gets pm_phy'rw_addr,

Expanding the definition of is_device_plain_memory?,

which is trivially true.

This completes the proof of device_plain_unchanged_memory_invariant.2.2.3.1.

device_plain_unchanged_memory_invariant.2.2.3.2:

{-1}	unchanged_memory_invariant?(pm_phy'mem, pm_phy'states, memory_write_side_effect_super_transformers(pm_phy'mem, ad- dress_block (a', length(b (pm_phy'ro_addr \cup pm_phy'rw_addr))
{-2}	plain_memory?(pm_phy)
{-3}	is_device_plain_memory?(device_read_side_effect', device_write_side_effect')(pm')
{-4}	(address_block(a', length(bl')) \subseteq pm'rw_addr)
{1}	((pm'ro_addr \cup pm'rw_addr) \subseteq (pm_phy'ro_addr \cup pm_phy'rw_addr))
{2}	unchanged_memory_invariant?(pm_phy'mem, pm_phy'states, singleton(expr_2_super(memory_write_side_effect(pm_phy'mem) (a', bl', cp'))), (pm'ro_addr \cup pm'rw_addr))

Using lemma pm_dev_ro_rw_addr,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of device_plain_unchanged_memory_invariant.2.2.3.2.

device_plain_unchanged_memory_invariant.2.2.3.3:

{-1}	unchanged_memory_invariant?(pm_phy'mem, pm_phy'states, memory_write_side_effect_super_transformers(pm_phy'mem, ad- dress_block (a', length(bl')), (pm_phy'ro_addr ∪ pm_phy'rw_addr))
{-2}	plain_memory?(pm_phy)
{-3}	is_device_plain_memory?(device_read_side_effect', device_write_side_effect')(pm')
{-4}	(address_block(a', length(bl')) ⊆ pm'rw_addr)
{1}	(singleton(expr_2_super(memory_write_side_effect(pm_phy'mem)(a', bl', cp'))) ⊆ memory_write_side_effect_s-
{2}	unchanged_memory_invariant?(pm_phy'mem, pm_phy'states, singleton(expr_2_super(memory_write_side_effect(pm_phy'mem) (a', bl', cp'))), (pm'ro_addr ∪ pm'rw_addr))

Keeping (1) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of device_plain_unchanged_memory_invariant.2.2.3.3.

device_plain_unchanged_memory_invariant.2.2.4:

{-1}	is_device_plain_memory?(device_read_side_effect', device_write_side_effect')(pm')
{-2}	plain_memory?(pm_phy)
{-3}	pm'states = em_lift(pm_phy'states)
{-4}	memory_write_side_effect(pm'mem) = (λ (a: Address, bl: list[Byte], cp: bool): em_lift[Physical_memory, Device_state, list[Byte]] (memory_write_side_effect(pm_phy'mem)(a, bl, cp)) ## (λ (bl1: list[Byte]): device_write_side_effect'(a, bl1, cp)))
{-5}	(address_block(a', length(bl')) ⊆ pm'rw_addr)
{1}	transformers_ok?(pm'states, memory_read_transformers(pm'mem, (pm'ro_addr ∪ pm'rw_addr)))
{2}	unchanged_memory_invariant?(pm'mem, pm'states, singleton(expr_2_super(em_lift [Physical_memory, Device_state, list[Byte]] (memory_write_side_effect(pm_phy'mem (a', bl', cp')) ## (λ (bl1: list[Byte]): device_write_side_effect'(a', bl1, cp')))) (pm'ro_addr ∪ pm'rw_addr))

Hiding formulas: 2,

Using lemma device_plain_transformers_ok,

Using lemma transformers_ok_mono_transformers,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Keeping (1) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `device_plain_unchanged_memory_invariant.2.2.4`.

`device_plain_unchanged_memory_invariant.2.2.5`:

```

{-1} is_device_plain_memory?(device_read_side_effect', device_write_side_effect')(pm')
{-2} plain_memory?(pm_phy)
{-3} pm'`states = em_lift(pm_phy`states)
{-4} memory_write_side_effect(pm'`mem) =
      (λ (a: Address, bl: list[Byte], cp: bool):
        em_lift[Physical_memory, Device_state, list[Byte]]
          (memory_write_side_effect(pm_phy`mem)(a, bl, cp))
          ## (λ (bl1: list[Byte]): device_write_side_effect'(a, bl1, cp)))
{-5} (address_block(a', length(bl')) ⊆ pm'`rw_addr)
-----
{1} singleton(expr_2_super(em_lift[Physical_memory, Device_state, list[Byte]]
      (memory_write_side_effect(pm_phy`mem)(a', bl', cp'))))
      (expr_2_super(em_lift[Physical_memory, Device_state, list[Byte]]
      (memory_write_side_effect(pm_phy`mem)(a', bl', cp'))))
{2} unchanged_memory_invariant?(pm'`mem, pm'`states,
      singleton(expr_2_super(em_lift
      [Physical_memory,
      Device_state,
      list[Byte]]
      (memory_write_side_effect(pm
      (a', bl', cp'))
      ##
      (λ (bl1: list[Byte]):
      device_write_side_effect'(a',
      (pm'`ro_addr ∪ pm'`rw_addr))

```

Expanding the definition of `singleton`,
which is trivially true.

This completes the proof of `device_plain_unchanged_memory_invariant.2.2.5`.

Q.E.D.

C.22.10 Challenge_Device_Memory.device_plain_unchanged_memory_invariant_write

Terse proof for `device_plain_unchanged_memory_invariant_write`.

`device_plain_unchanged_memory_invariant_write`:

```

{1} ∃ (device_read_side_effect,
      device_write_side_effect:
      [[Address, list[Byte], bool] →
      [Device_memory[Physical_memory, pm_phy, Device_state] →
      ExprResult[Device_memory, list[Byte]]]],
      pm: Plain_Memory[Device_memory]):
      is_device_plain_memory?(device_read_side_effect, device_write_side_effect)(pm) ⊃
      unchanged_memory_invariant?(pm'`mem, pm'`states, mem-
      ory_write_transformers(pm'`mem, pm'`rw_addr),
      pm'`ro_addr)

```

Repeatedly Skolemizing and flattening,
 Rewriting using em_memory_write_transformers, matching in * where em_pm gets pm!1'mem, pm gets pm_phy'mem,
 we get 2 subgoals:

device_plain_unchanged_memory_invariant_write.1:

{-1}	is_device_plain_memory?(device_read_side_effect', device_write_side_effect')(pm')
{1}	unchanged_memory_invariant?(pm' mem, pm' states, em_lift(memory_write_transformers(pm_phy' mem, pm' rw_addr)), pm' ro_addr)

Using lemma pm_dev_states,
 Using lemma pm_dev_plain,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 Using lemma em_unchanged_memory_invariant,
 Using lemma pm_dev_read,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 Hiding formulas: 2,
 Using lemma unchanged_memory_invariant_mono,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 we get 3 subgoals:

device_plain_unchanged_memory_invariant_write.1.1:

{-1}	is_device_plain_memory?(device_read_side_effect', device_write_side_effect')(pm')
{-2}	memory_read(pm' mem) = (λ (a: Address): em_lift[Physical_memory, Device_state, Byte](memory_read(pm_phy' mem)(a)))
{-3}	plain_memory?(pm_phy)
{-4}	pm' states = em_lift(pm_phy' states)
{1}	unchanged_memory_invariant?(pm_phy' mem, pm_phy' states, memory_write_transformers(pm_phy' mem, pm' rw_addr), (pm_phy' ro_addr ∪ (pm_phy' rw_addr \ pm' rw_addr)))
{2}	unchanged_memory_invariant?(pm_phy' mem, pm_phy' states, memory_write_transformers(pm_phy' mem, pm' rw_addr), pm' ro_addr)

Hiding formulas: 2,
 Rewriting using unchanged_memory_invariant_union_addresses[Physical_memory], matching in *,
 we get 2 subgoals:

device_plain_unchanged_memory_invariant_write.1.1.1:

{-1}	is_device_plain_memory?(device_read_side_effect', device_write_side_effect')(pm')
{-2}	memory_read(pm' mem) = (λ (a: Address): em_lift[Physical_memory, Device_state, Byte](memory_read(pm_phy' mem)(a)))
{-3}	plain_memory?(pm_phy)
{-4}	pm' states = em_lift(pm_phy' states)
{1}	unchanged_memory_invariant?(pm_phy' mem, pm_phy' states, memory_write_transformers(pm_phy' mem, pm' rw_addr), pm_phy' ro_addr)
{2}	unchanged_memory_invariant?(pm_phy' mem, pm_phy' states, memory_write_transformers(pm_phy' mem, pm' rw_addr), (pm_phy' ro_addr ∪ (pm_phy' rw_addr \ pm' rw_addr)))

Using lemma unchanged_memory_invariant_mono,
 Rewriting using subset_reflexive, matching in *,
 Using lemma plain_memory_unchanged_memory_invariant_write,

C Proof scripts

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 Rewriting using `memory_write_transformers_mono`, matching in `*`,
 Expanding the definition of `is_device_plain_memory?`,
 which is trivially true.

This completes the proof of `device_plain_unchanged_memory_invariant_write.1.1.1`.
`device_plain_unchanged_memory_invariant_write.1.1.2`:

{-1}	is_device_plain_memory?(device_read_side_effect', device_write_side_effect')(pm')
{-2}	memory_read(pm' mem) = (λ (a: Address): em_lift[Physical_memory, Device_state, Byte](memory_read(pm_phy mem)(a)))
{-3}	plain_memory?(pm_phy)
{-4}	pm' states = em_lift(pm_phy states)
{1}	unchanged_memory_invariant?(pm_phy mem, pm_phy states, memory_write_transformers(pm_phy mem, pm' rw_addr), (pm_phy rw_addr \ pm' rw_addr))
{2}	unchanged_memory_invariant?(pm_phy mem, pm_phy states, memory_write_transformers(pm_phy mem, pm' rw_addr), (pm_phy ro_addr ∪ (pm_phy rw_addr \ pm' rw_addr)))

Rewriting using `unchanged_memory_invariant_all_transformers`, matching in `*`,
 Expanding the definition of `memory_write_transformers`,
 Repeatedly Skolemizing and flattening,
 Replacing using formula -3,
 Hiding formulas: (-3 2 3),
 Using lemma `plain_memory_unchanged_memory_write_invariant`,
 Expanding the definition of `unchanged_memory_write_invariant?`,
 Instantiating quantified variables,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 we get 2 subgoals:

`device_plain_unchanged_memory_invariant_write.1.1.2.1`:

{-1}	unchanged_memory_invariant?(pm_phy mem, pm_phy states, singleton(expr_2_super(memory_write(pm_phy mem)(a', b')), (pm_phy rw_addr \ {a'}))
{-2}	b' < max_byte
{-3}	pm' rw_addr(a')
{-4}	is_device_plain_memory?(device_read_side_effect', device_write_side_effect')(pm')
{-5}	memory_read(pm' mem) = (λ (a: Address): em_lift[Physical_memory, Device_state, Byte](memory_read(pm_phy mem)(a)))
{-6}	plain_memory?(pm_phy)
{-7}	pm' states = em_lift(pm_phy states)
{1}	unchanged_memory_invariant?(pm_phy mem, pm_phy states, singleton(expr_2_super(memory_write(pm_phy mem)(a', b')), (pm_phy rw_addr \ pm' rw_addr))

Using lemma `unchanged_memory_invariant_mono`,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 we get 2 subgoals:

device_plain_unchanged_memory_invariant_write.1.1.2.1.1:

{-1}	unchanged_memory_invariant?(pm_phy'mem, pm_phy'states, singleton(expr_2_super(memory_write(pm_phy'mem)(a', b'))), (pm_phy'rw_addr \ {a'}))
{-2}	b' < max_byte
{-3}	pm'rw_addr(a')
{-4}	is_device_plain_memory?(device_read_side_effect', device_write_side_effect')(pm')
{-5}	memory_read(pm'mem) = (λ (a: Address): em_lift[Physical_memory, Device_state, Byte](memory_read(pm_phy'mem)(a)))
{-6}	plain_memory?(pm_phy)
{-7}	pm'states = em_lift(pm_phy'states)
{1}	((pm_phy'rw_addr \ pm'rw_addr) ⊆ (pm_phy'rw_addr \ {a'}))
{2}	unchanged_memory_invariant?(pm_phy'mem, pm_phy'states, singleton(expr_2_super(memory_write(pm_phy'mem)(a', b'))), (pm_phy'rw_addr \ pm'rw_addr))

Keeping (-3 1) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of device_plain_unchanged_memory_invariant_write.1.1.2.1.1.

device_plain_unchanged_memory_invariant_write.1.1.2.1.2:

{-1}	unchanged_memory_invariant?(pm_phy'mem, pm_phy'states, singleton(expr_2_super(memory_write(pm_phy'mem)(a', b'))), (pm_phy'rw_addr \ {a'}))
{-2}	b' < max_byte
{-3}	pm'rw_addr(a')
{-4}	is_device_plain_memory?(device_read_side_effect', device_write_side_effect')(pm')
{-5}	memory_read(pm'mem) = (λ (a: Address): em_lift[Physical_memory, Device_state, Byte](memory_read(pm_phy'mem)(a)))
{-6}	plain_memory?(pm_phy)
{-7}	pm'states = em_lift(pm_phy'states)
{1}	(singleton(expr_2_super(memory_write(pm_phy'mem)(a', b'))) ⊆ singleton(expr_2_super(memory_write(pm_phy'mem)(a', b'))))
{2}	unchanged_memory_invariant?(pm_phy'mem, pm_phy'states, singleton(expr_2_super(memory_write(pm_phy'mem)(a', b'))), (pm_phy'rw_addr \ pm'rw_addr))

Rewriting using subset_reflexive, matching in *,

This completes the proof of device_plain_unchanged_memory_invariant_write.1.1.2.1.2.

device_plain_unchanged_memory_invariant_write.1.1.2.2:

{-1}	b' < max_byte
{-2}	pm'rw_addr(a')
{-3}	is_device_plain_memory?(device_read_side_effect', device_write_side_effect')(pm')
{-4}	memory_read(pm'mem) = (λ (a: Address): em_lift[Physical_memory, Device_state, Byte](memory_read(pm_phy'mem)(a)))
{-5}	plain_memory?(pm_phy)
{-6}	pm'states = em_lift(pm_phy'states)
{1}	pm_phy'rw_addr(a')
{2}	unchanged_memory_invariant?(pm_phy'mem, pm_phy'states, singleton(expr_2_super(memory_write(pm_phy'mem)(a', b'))), (pm_phy'rw_addr \ pm'rw_addr))

C Proof scripts

Expanding the definition of `is_device_plain_memory?`,
Applying disjunctive simplification to flatten sequent,
Keeping (-2 -5 1) and hiding *,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `device_plain_unchanged_memory_invariant_write.1.1.2.2`.
`device_plain_unchanged_memory_invariant_write.1.2:`

{-1}	<code>is_device_plain_memory?(device_read_side_effect', device_write_side_effect')(pm')</code>
{-2}	<code>memory_read(pm' mem) =</code> <code>(λ (a: Address):</code> <code> em_lift[Physical_memory, Device_state, Byte](memory_read(pm_phy' mem)(a)))</code>
{-3}	<code>plain_memory?(pm_phy)</code>
{-4}	<code>pm' states = em_lift(pm_phy' states)</code>
{1}	<code>(pm' ro_addr ⊆ (pm_phy' ro_addr ∪ (pm_phy' rw_addr \ pm' rw_addr)))</code>
{2}	<code>unchanged_memory_invariant?(pm_phy' mem, pm_phy' states,</code> <code> memory_write_transformers(pm_phy' mem, pm' rw_addr), pm</code>

Expanding the definition of `is_device_plain_memory?`,
which is trivially true.

This completes the proof of `device_plain_unchanged_memory_invariant_write.1.2`.
`device_plain_unchanged_memory_invariant_write.1.3:`

{-1}	<code>is_device_plain_memory?(device_read_side_effect', device_write_side_effect')(pm')</code>
{-2}	<code>memory_read(pm' mem) =</code> <code>(λ (a: Address):</code> <code> em_lift[Physical_memory, Device_state, Byte](memory_read(pm_phy' mem)(a)))</code>
{-3}	<code>plain_memory?(pm_phy)</code>
{-4}	<code>pm' states = em_lift(pm_phy' states)</code>
{1}	<code>(memory_write_transformers(pm_phy' mem, pm' rw_addr) ⊆ memory_write_transformers(pm_phy' mem,</code>
{2}	<code>unchanged_memory_invariant?(pm_phy' mem, pm_phy' states,</code> <code> memory_write_transformers(pm_phy' mem, pm' rw_addr), pm</code>

Rewriting using `memory_write_transformers_mono`, matching in *,

Rewriting using `subset_reflexive`, matching in *,

This completes the proof of `device_plain_unchanged_memory_invariant_write.1.3`.

`device_plain_unchanged_memory_invariant_write.2:`

{-1}	<code>is_device_plain_memory?(device_read_side_effect', device_write_side_effect')(pm')</code>
{1}	<code>(pm' mem' memory_write =</code> <code>(λ (a: Address, b: Byte):</code> <code> em_lift[Physical_memory, Device_state, Unit](pm_phy' mem' memory_write(a, b)))</code>
{2}	<code>unchanged_memory_invariant?(pm' mem, pm' states,</code> <code> memory_write_transformers(pm' mem, pm' rw_addr), pm' ro</code>

Using lemma `pm_dev_write`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `device_plain_unchanged_memory_invariant_write.2`.

Q.E.D.

C.22.11 Challenge Device Memory.device_plain_unchanged_memory_write_invariant

Terse proof for `device_plain_unchanged_memory_write_invariant`.

device_plain_unchanged_memory_write_invariant:

<pre> {1} ∃ (device_read_side_effect, device_write_side_effect: [[Address, list[Byte], bool] → [Device_memory[Physical_memory, pm_phy, Device_state] → ExprResult[Device_memory, list[Byte]]]], pm: Plain_Memory[Device_memory]): is_device_plain_memory?(device_read_side_effect, device_write_side_effect)(pm) ⊃ unchanged_memory_write_invariant?(pm' mem, pm' states, pm' rw_addr) </pre>

Expanding the definition of unchanged_memory_write_invariant?,

Repeatedly Skolemizing and flattening,

Using lemma pm_dev_write,

Using lemma pm_dev_states,

Using lemma pm_dev_plain,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Replacing using formula -4,

Simplifying, rewriting, and recording with decision procedures,

Rewriting using em_lift_expr_2_super, matching in *,

Rewriting using em_lift_singleton, matching in *,

Using lemma em_unchanged_memory_invariant[Physical_memory, Device_state],

Using lemma pm_dev_read,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Using lemma plain_memory_unchanged_memory_write_invariant,

Expanding the definition of unchanged_memory_write_invariant?,

Instantiating quantified variables,

Using lemma unchanged_memory_invariant_mono,

Rewriting using subset_reflexive, matching in *,

Case splitting on subset?(pm!1'rw_addr, pm_phy'rw_addr),

we get 2 subgoals:

device_plain_unchanged_memory_write_invariant.1:

{-1}	$(pm_rw_addr \subseteq pm_phy_rw_addr)$
{-2}	$((pm_rw_addr \setminus \{waddr'\}) \subseteq (pm_phy_rw_addr \setminus \{waddr'\})) \wedge$ $unchanged_memory_invariant?(pm_phy_mem, pm_phy_states,$ $singleton(expr_2_super(memory_write(pm_phy_mem)$ $(waddr', b')),$ $(pm_phy_rw_addr \setminus \{waddr'\}))$
	\supset $unchanged_memory_invariant?(pm_phy_mem, pm_phy_states,$ $singleton(expr_2_super(memory_write(pm_phy_mem)$ $(waddr', b')),$ $(pm_rw_addr \setminus \{waddr'\}))$
{-3}	$pm_phy_rw_addr(waddr') \supset$ $unchanged_memory_invariant?(pm_phy_mem, pm_phy_states,$ $singleton(expr_2_super(memory_write(pm_phy_mem)$ $(waddr', b')),$ $(pm_phy_rw_addr \setminus \{waddr'\}))$
{-4}	$is_device_plain_memory?(device_read_side_effect', device_write_side_effect')(pm')$
{-5}	$memory_read(pm_mem) =$ $(\lambda (a: Address):$ $em_lift[Physical_memory, Device_state, Byte](memory_read(pm_phy_mem)(a)))$
{-6}	$plain_memory?(pm_phy)$
{-7}	$pm_states = em_lift(pm_phy_states)$
{-8}	$memory_write(pm_mem) =$ $(\lambda (a: Address, b: Byte):$ $em_lift[Physical_memory, Device_state, Unit](memory_write(pm_phy_mem)(a, b)))$
{-9}	$b' < max_byte$
{-10}	$pm_rw_addr(waddr')$
{1}	$unchanged_memory_invariant?(pm_phy_mem, pm_phy_states,$ $singleton(expr_2_super(memory_write(pm_phy_mem)$ $(waddr', b')),$ $(pm_rw_addr \setminus \{waddr'\}))$
{2}	$unchanged_memory_invariant?(pm_mem, pm_states,$ $em_lift(singleton(expr_2_super(memory_write(pm_phy_mem)$ $(waddr', b'))),$ $(pm_rw_addr \setminus \{waddr'\}))$

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 2 subgoals:

device_plain_unchanged_memory_write_invariant.1.1:

{-1}	$(pm'rw_addr \subseteq pm_phy'rw_addr)$
{-2}	$is_device_plain_memory?(device_read_side_effect', device_write_side_effect')(pm')$
{-3}	$memory_read(pm'mem) =$ $(\lambda (a: Address):$ $em_lift[Physical_memory, Device_state, Byte](memory_read(pm_phy'mem)(a)))$
{-4}	$plain_memory?(pm_phy)$
{-5}	$pm'states = em_lift(pm_phy'states)$
{-6}	$memory_write(pm'mem) =$ $(\lambda (a: Address, b: Byte):$ $em_lift[Physical_memory, Device_state, Unit](memory_write(pm_phy'mem)(a, b)))$
{-7}	$b' < max_byte$
{-8}	$pm'rw_addr(waddr')$

{1}	$unchanged_memory_invariant?(pm_phy'mem, pm_phy'states,$ $singleton(expr_2_super(memory_write(pm_phy'mem)$ $(waddr', b'))),$ $(pm_phy'rw_addr \setminus \{waddr'\}))$
{2}	$unchanged_memory_invariant?(pm_phy'mem, pm_phy'states,$ $singleton(expr_2_super(memory_write(pm_phy'mem)$ $(waddr', b'))),$ $(pm'rw_addr \setminus \{waddr'\}))$
{3}	$pm_phy'rw_addr(waddr')$
{4}	$unchanged_memory_invariant?(pm'mem, pm'states,$ $em_lift(singleton(expr_2_super(memory_write(pm_phy'mem)$ $(waddr', b')))),$ $(pm'rw_addr \setminus \{waddr'\}))$

Keeping (-1 -8 3) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of device_plain_unchanged_memory_write_invariant.1.1.

C Proof scripts

device_plain_unchanged_memory_write_invariant.1.2:

{-1}	$(pm'rw_addr \subseteq pm_phy'rw_addr)$
{-2}	unchanged_memory_invariant?(pm_phy'mem, pm_phy'states, singleton(expr_2_super(memory_write(pm_phy'mem) (waddr', b'))), (pm_phy'rw_addr \ {waddr'}))
{-3}	is_device_plain_memory?(device_read_side_effect', device_write_side_effect')(pm')
{-4}	memory_read(pm'mem) = ($\lambda (a: \text{Address}):$ em_lift[Physical_memory, Device_state, Byte](memory_read(pm_phy'mem)(a)))
{-5}	plain_memory?(pm_phy)
{-6}	pm'states = em_lift(pm_phy'states)
{-7}	memory_write(pm'mem) = ($\lambda (a: \text{Address}, b: \text{Byte}):$ em_lift[Physical_memory, Device_state, Unit](memory_write(pm_phy'mem)(a, b)))
{-8}	$b' < \text{max_byte}$
{-9}	pm'rw_addr(waddr')
<hr/>	
{1}	$((pm'rw_addr \ \{waddr'\}) \subseteq (pm_phy'rw_addr \ \{waddr'\}))$
{2}	unchanged_memory_invariant?(pm_phy'mem, pm_phy'states, singleton(expr_2_super(memory_write(pm_phy'mem) (waddr', b'))), (pm'rw_addr \ {waddr'}))
{3}	unchanged_memory_invariant?(pm'mem, pm'states, em_lift(singleton(expr_2_super(memory_write(pm_phy'mem) (waddr', b')))), (pm'rw_addr \ {waddr'}))

Expanding the definition of subset?,

Expanding the definition of member,

Repeatedly Skolemizing and flattening,

Expanding the definition of remove,

Expanding the definition of /=,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Expanding the definition of member,

Instantiating quantified variables,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of device_plain_unchanged_memory_write_invariant.1.2.

device_plain_unchanged_memory_write_invariant.2:

<p>{-1} ((pm'rw_addr \ {waddr'}) \subseteq (pm_phy'rw_addr \ {waddr'})) \wedge unchanged_memory_invariant?(pm_phy'mem, pm_phy'states, singleton(expr_2_super(memory_write(pm_phy'mem) (waddr', b'))), (pm_phy'rw_addr \ {waddr'})) \supset unchanged_memory_invariant?(pm_phy'mem, pm_phy'states, singleton(expr_2_super(memory_write(pm_phy'mem) (waddr', b'))), (pm'rw_addr \ {waddr'}))</p> <p>{-2} pm_phy'rw_addr(waddr') \supset unchanged_memory_invariant?(pm_phy'mem, pm_phy'states, singleton(expr_2_super(memory_write(pm_phy'mem) (waddr', b'))), (pm_phy'rw_addr \ {waddr'}))</p> <p>{-3} is_device_plain_memory?(device_read_side_effect', device_write_side_effect')(pm')</p> <p>{-4} memory_read(pm'mem) = (λ (a: Address): em_lift[Physical_memory, Device_state, Byte](memory_read(pm_phy'mem)(a)))</p> <p>{-5} plain_memory?(pm_phy)</p> <p>{-6} pm'states = em_lift(pm_phy'states)</p> <p>{-7} memory_write(pm'mem) = (λ (a: Address, b: Byte): em_lift[Physical_memory, Device_state, Unit](memory_write(pm_phy'mem)(a, b)))</p> <p>{-8} b' < max_byte</p> <p>{-9} pm'rw_addr(waddr')</p>	<hr style="border: 0.5px solid black;"/> <p>{1} (pm'rw_addr \subseteq pm_phy'rw_addr)</p> <p>{2} unchanged_memory_invariant?(pm_phy'mem, pm_phy'states, singleton(expr_2_super(memory_write(pm_phy'mem) (waddr', b'))), (pm'rw_addr \ {waddr'}))</p> <p>{3} unchanged_memory_invariant?(pm'mem, pm'states, em_lift(singleton(expr_2_super(memory_write(pm_phy'mem) (waddr', b'))), (pm'rw_addr \ {waddr'}))</p>
---	--

Expanding the definition of is_device_plain_memory?,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of device_plain_unchanged_memory_write_invariant.2.

Q.E.D.

C.22.12

Challenge_Device_Memory.device_plain_changed_memory_invariant

Terse proof for device_plain_changed_memory_invariant.

device_plain_changed_memory_invariant:

```
{1}  ∃ (device_read_side_effect,
      device_write_side_effect:
        [[Address, list[Byte], bool] →
         [Device_memory[Physical_memory, pm_phy, Device_state] →
          ExprResult[Device_memory, list[Byte]]]],
      pm: Plain_Memory[Device_memory]):
  is_device_plain_memory?(device_read_side_effect, device_write_side_effect)(pm) ⊃
  changed_memory_invariant?(pm' mem, pm' states, pm' rw_addr)
```

Repeatedly Skolemizing and flattening,

Using lemma pm_dev_plain,

Using lemma pm_dev_states,

Using lemma pm_dev_read,

Using lemma pm_dev_write,

Using lemma em_changed_memory_invariant[Physical_memory, Device_state],

Using lemma plain_memory_changed_memory_invariant,

Using lemma changed_memory_invariant_mono,

Expanding the definition of is_device_plain_memory?,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of device_plain_changed_memory_invariant.

Q.E.D.

C.22.13

Challenge_Device_Memory.device_plain_read_side_effect_unchanged

Terse proof for device_plain_read_side_effect_unchanged.

device_plain_read_side_effect_unchanged:

```
{1}  ∃ (device_read_side_effect,
      device_write_side_effect:
        [[Address, list[Byte], bool] →
         [Device_memory[Physical_memory, pm_phy, Device_state] →
          ExprResult[Device_memory, list[Byte]]]],
      pm: Plain_Memory[Device_memory]):
  is_device_plain_memory?(device_read_side_effect, device_write_side_effect)(pm) ⊃
  side_effect_content_unchanged((pm' ro_addr ∪ pm' rw_addr), pm' states,
                                memory_read_side_effect(pm' mem))
```

Repeatedly Skolemizing and flattening,

Using lemma pm_dev_read_side_effect,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Replacing using formula -2,

Rewriting using side_effect_content_unchanged_composition, matching in * where states gets pm!1'states,

we get 3 subgoals:

device_plain_read_side_effect_unchanged.1:

{-1}	is_device_plain_memory?(device_read_side_effect', device_write_side_effect')(pm')
{-2}	memory_read_side_effect(pm' mem) = $(\lambda (a: \text{Address}, \text{bl}: \text{list}[\text{Byte}], \text{cp}: \text{bool}):$ $\text{em_lift}[\text{Physical_memory}, \text{Device_state}, \text{list}[\text{Byte}]]$ $(\text{memory_read_side_effect}(\text{pm_phy}' \text{mem})(a, \text{bl}, \text{cp}))$ $\text{## } (\lambda (\text{bl1}: \text{list}[\text{Byte}]): \text{device_read_side_effect}'(a, \text{bl1}, \text{cp}))$)
{1}	transformer_invariant?(pm' states, $\text{se_super_transformers}((\text{pm}' \text{ro_addr} \cup \text{pm}' \text{rw_addr}),$ $\lambda (a: \text{Ad-}$ $\text{dress}, \text{bl}: \text{list}[\text{Byte}], \text{cp}: \text{bool}):$ em_lift $[\text{Physical_memory}, \text{De-}$ $\text{vice_state}, \text{list}[\text{Byte}]]$ $(\text{memory_read_side_effect}(\text{pm_phy}' \text{mem})$ $(a, \text{bl}, \text{cp})))$)
{2}	side_effect_content_unchanged((pm' ro_addr \cup pm' rw_addr), pm' states, $\lambda (a: \text{Address}, \text{bl}: \text{list}[\text{Byte}], \text{cp}: \text{bool}):$ $\text{em_lift}[\text{Physical_memory}, \text{De-}$ $\text{vice_state}, \text{list}[\text{Byte}]]$ $(\text{memory_read_side_effect}(\text{pm_phy}' \text{mem})(a, \text{bl}, \text{cp}))$ ## $(\lambda (\text{bl1}: \text{list}[\text{Byte}]):$ $\text{device_read_side_effect}'(a, \text{bl1}, \text{cp}))$)

Hiding formulas: 2,

Using lemma pm_dev_plain,

Using lemma pm_dev_states,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Using lemma plain_memory_invariant,

Using lemma transformer_invariant_all_transformers,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Expanding the definition of se_super_transformers,

Repeatedly Skolemizing and flattening,

Replacing using formula -3,

Rewriting using em_lift_expr_2_super, matching in *,

Rewriting using em_lift_singleton, matching in *,

Using lemma em_transformer_invariant [Physical_memory, Device_state],

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Keeping (-2 -5 -6 1) and hiding *,

Rewriting using transformer_invariant_mono_transformers, matching in * where transformers_1 gets singleton(expr_2_super(memory_read_side_effect(pm_phy mem) (a!1, bl!1, cp!1))), transformers_2 gets union(pm_phy other_actions, union(union(memory_read_transformers(pm_phy mem, union(pm_phy ro_addr, pm_phy rw_addr)), memory_write_transformers(pm_phy mem, pm_phy rw_addr)), union(memory_read_side_effect_super_transformers(pm_phy mem, union(pm_phy ro_addr, pm_phy rw_addr)), memory_write_side_effect_super_transformers(pm_phy mem, pm_phy rw_addr))),

Using lemma pm_dev_ro_rw_addr,

Using lemma subset_transitive,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Keeping (-3 1) and hiding *,

Expanding the definition of subset?,

Repeatedly Skolemizing and flattening,

C Proof scripts

Expanding the definition of union,

Expanding the definition of member,

Applying disjunctive simplification to flatten sequent,

Expanding the definition of singleton,

Expanding the definition of memory_read_side_effect_super_transformers,

Instantiating quantified variables,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `device_plain_read_side_effect_unchanged.1`.

`device_plain_read_side_effect_unchanged.2`:

{-1}	is_device_plain_memory?(device_read_side_effect', device_write_side_effect')(pm')
{-2}	memory_read_side_effect(pm' mem) = $(\lambda (a: \text{Address}, bl: \text{list}[\text{Byte}], cp: \text{bool}):$ $\quad \text{em_lift}[\text{Physical_memory}, \text{Device_state}, \text{list}[\text{Byte}]]$ $\quad (\text{memory_read_side_effect}(pm_phy' \text{mem})(a, bl, cp))$ $\quad \#\# (\lambda (bl1: \text{list}[\text{Byte}]): \text{device_read_side_effect}'(a, bl1, cp)))$
{1}	side_effect_content_unchanged((pm'ro_addr \cup pm'rw_addr), pm' states, $\lambda (a: \text{Address}, bl: \text{list}[\text{Byte}], cp: \text{bool}):$ $\quad \text{em_lift}[\text{Physical_memory}, \text{De-}$ $\quad \text{vice_state}, \text{list}[\text{Byte}]]$ $\quad (\text{memory_read_side_effect}(pm_phy' \text{mem})(a, bl, cp)))$
{2}	side_effect_content_unchanged((pm'ro_addr \cup pm'rw_addr), pm' states, $\lambda (a: \text{Address}, bl: \text{list}[\text{Byte}], cp: \text{bool}):$ $\quad \text{em_lift}[\text{Physical_memory}, \text{De-}$ $\quad \text{vice_state}, \text{list}[\text{Byte}]]$ $\quad (\text{memory_read_side_effect}(pm_phy' \text{mem})(a, bl, cp))$ $\quad \#\#$ $\quad (\lambda (bl1: \text{list}[\text{Byte}]):$ $\quad \quad \text{device_read_side_effect}'(a, bl1, cp)))$

Hiding formulas: 2,

Using lemma `pm_dev_plain`,

Using lemma `pm_dev_states`,

Using lemma `pm_dev_ro_rw_addr`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Using lemma `side_effect_content_unchanged_mono[Device_memory]`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Replacing using formula -3,

Rewriting using `em_side_effect_content_unchanged`, matching in `*`,

Expanding the definition of `plain_memory?`,

which is trivially true.

This completes the proof of `device_plain_read_side_effect_unchanged.2`.

device_plain_read_side_effect_unchanged.3:

{-1}	is_device_plain_memory?(device_read_side_effect', device_write_side_effect')(pm')
{-2}	memory_read_side_effect(pm' mem) = $(\lambda (a: \text{Address}, bl: \text{list}[\text{Byte}], cp: \text{bool}):$ $\text{em_lift}[\text{Physical_memory}, \text{Device_state}, \text{list}[\text{Byte}]]$ $(\text{memory_read_side_effect}(pm_phy' \text{mem})(a, bl, cp))$ $\#\# (\lambda (bl1: \text{list}[\text{Byte}]): \text{device_read_side_effect}'(a, bl1, cp)))$
{1}	side_effect_content_unchanged((pm'ro_addr \cup pm'rw_addr), pm'states, device_read_side_effect')
{2}	side_effect_content_unchanged((pm'ro_addr \cup pm'rw_addr), pm'states, $\lambda (a: \text{Address}, bl: \text{list}[\text{Byte}], cp: \text{bool}):$ $\text{em_lift}[\text{Physical_memory}, \text{De-}$ $\text{vice_state}, \text{list}[\text{Byte}]]$ $(\text{memory_read_side_effect}(pm_phy' \text{mem})(a, bl, cp))$ $\#\#$ $(\lambda (bl1: \text{list}[\text{Byte}]):$ $\text{device_read_side_effect}'(a, bl1, cp)))$

Hiding formulas: 2,

Expanding the definition of is_device_plain_memory?,
 which is trivially true.

This completes the proof of device_plain_read_side_effect_unchanged.3.

Q.E.D.

C.22.14

Challenge_Device_Memory.device_plain_write_side_effect_unchanged

Terse proof for device_plain_write_side_effect_unchanged.

device_plain_write_side_effect_unchanged:

{1}	$\forall (\text{device_read_side_effect},$ $\text{device_write_side_effect}:$ $[[\text{Address}, \text{list}[\text{Byte}], \text{bool}] \rightarrow$ $[\text{Device_memory}[\text{Physical_memory}, pm_phy, \text{Device_state}] \rightarrow$ $\text{ExprResult}[\text{Device_memory}, \text{list}[\text{Byte}]]],$ $pm: \text{Plain_Memory}[\text{Device_memory}]):$ $\text{is_device_plain_memory?}(\text{device_read_side_effect}, \text{device_write_side_effect})(pm) \supset$ $\text{side_effect_content_unchanged}(pm'rw_addr, pm'states, \text{mem-}$ $\text{ory_write_side_effect}(pm' \text{mem}))$
-----	--

Repeatedly Skolemizing and flattening,

Using lemma pm_dev_write_side_effect,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Replacing using formula -2,

Rewriting using side_effect_content_unchanged_composition, matching in * where states gets pm!1'states,

we get 3 subgoals:

device_plain_write_side_effect_unchanged.1:

{-1}	is_device_plain_memory?(device_read_side_effect', device_write_side_effect')(pm')
{-2}	memory_write_side_effect(pm' mem) = $(\lambda (a: \text{Address}, bl: \text{list}[\text{Byte}], cp: \text{bool}):$ em_lift[Physical_memory, Device_state, list[Byte]] (memory_write_side_effect(pm_phy' mem)(a, bl, cp)) ## $(\lambda (bl1: \text{list}[\text{Byte}]): \text{device_write_side_effect}'(a, bl1, cp))$)
{1}	transformer_invariant?(pm' states, se_super_transformers(pm' rw_addr, $\lambda (a: \text{Ad-}$ dress, bl: list[Byte], cp: bool): em_lift [Physical_memory, De- vice_state, list[Byte]] (memory_write_side_effect(pm_phy' (a, bl, cp))))
{2}	side_effect_content_unchanged(pm' rw_addr, pm' states, $\lambda (a: \text{Address}, bl: \text{list}[\text{Byte}], cp: \text{bool}):$ em_lift[Physical_memory, De- vice_state, list[Byte]] (memory_write_side_effect(pm_phy' mem)(a, bl, cp)) ## $(\lambda (bl1: \text{list}[\text{Byte}]):$ device_write_side_effect'(a, bl1, cp))

Hiding formulas: 2,

Using lemma pm_dev_plain,

Using lemma pm_dev_states,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Using lemma plain_memory_invariant,

Using lemma transformer_invariant_all_transformers,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Expanding the definition of se_super_transformers,

Repeatedly Skolemizing and flattening,

Replacing using formula -3,

Rewriting using em_lift_expr_2_super, matching in *,

Rewriting using em_lift_singleton, matching in *,

Using lemma em_transformer_invariant[Physical_memory, Device_state],

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Keeping (-2 -5 -6 1) and hiding *,

Rewriting using transformer_invariant_mono_transformers, matching in * where transformers_1 gets singleton(expr_2_super(memory_write_side_effect(pm_phy' mem) (a!1, bl!1, cp!1))), transformers_2 gets union(pm_phy' other_actions, union(union(memory_read_transformers(pm_phy' mem, union(pm_phy' ro_addr, pm_phy' rw_addr)), memory_write_transformers(pm_phy' mem, pm_phy' rw_addr)), union(memory_read_side_effect_super_transformers(pm_phy' mem, union(pm_phy' ro_addr, pm_phy' rw_addr)), memory_write_side_effect_super_transformers(pm_phy' mem, pm_phy' rw_addr)))).

Using lemma subset_transitive,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 2 subgoals:

device_plain_write_side_effect_unchanged.1.1:

{-1}	$(\text{address_block}(a', \text{length}(bl')) \subseteq \text{pm}'\text{rw_addr})$
{-2}	$(\text{address_block}(a', \text{length}(bl')) \subseteq \text{pm_phy}'\text{rw_addr})$
{-3}	$\text{transformer_invariant}?(\text{pm_phy}'\text{states},$ $\text{pm_phy}'\text{other_actions} \cup ((\text{memory_read_transformers}(\text{pm_phy}'\text{mem}, (\text{pm_phy}'\text{ro})))$
{-4}	$\text{is_device_plain_memory}?(\text{device_read_side_effect}', \text{device_write_side_effect}')(\text{pm}')$
{1}	$(\text{singleton}(\text{expr_2_super}(\text{memory_write_side_effect}(\text{pm_phy}'\text{mem})(a', bl', cp')))) \subseteq (\text{pm_phy}'\text{other_actions} \cup ((\text{memory_read_transformers}(\text{pm_phy}'\text{mem}, (\text{pm_phy}'\text{ro}))))$
{2}	$\text{transformer_invariant}?(\text{pm_phy}'\text{states},$ $\text{singleton}(\text{expr_2_super}(\text{memory_write_side_effect}(\text{pm_phy}'\text{mem})(a', bl', cp'))))$

Keeping (-2 1) and hiding *,

Expanding the definition of subset?,

Expanding the definition of union,

Expanding the definition of member,

Repeatedly Skolemizing and flattening,

Expanding the definition of memory_write_side_effect_super_transformers,

Expanding the definition of singleton,

Instantiating quantified variables,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of device_plain_write_side_effect_unchanged.1.1.

device_plain_write_side_effect_unchanged.1.2:

{-1}	$(\text{address_block}(a', \text{length}(bl')) \subseteq \text{pm}'\text{rw_addr})$
{-2}	$\text{transformer_invariant}?(\text{pm_phy}'\text{states},$ $\text{pm_phy}'\text{other_actions} \cup ((\text{memory_read_transformers}(\text{pm_phy}'\text{mem}, (\text{pm_phy}'\text{ro}))))$
{-3}	$\text{is_device_plain_memory}?(\text{device_read_side_effect}', \text{device_write_side_effect}')(\text{pm}')$
{1}	$(\text{pm}'\text{rw_addr} \subseteq \text{pm_phy}'\text{rw_addr})$
{2}	$(\text{singleton}(\text{expr_2_super}(\text{memory_write_side_effect}(\text{pm_phy}'\text{mem})(a', bl', cp')))) \subseteq (\text{pm_phy}'\text{other_actions} \cup ((\text{memory_read_transformers}(\text{pm_phy}'\text{mem}, (\text{pm_phy}'\text{ro}))))$
{3}	$\text{transformer_invariant}?(\text{pm_phy}'\text{states},$ $\text{singleton}(\text{expr_2_super}(\text{memory_write_side_effect}(\text{pm_phy}'\text{mem})(a', bl', cp'))))$

Expanding the definition of is_device_plain_memory?,

which is trivially true.

This completes the proof of device_plain_write_side_effect_unchanged.1.2.

device_plain_write_side_effect_unchanged.2:

{-1}	is_device_plain_memory?(device_read_side_effect', device_write_side_effect')(pm')
{-2}	memory_write_side_effect(pm' mem) = $(\lambda (a: \text{Address}, \text{bl}: \text{list}[\text{Byte}], \text{cp}: \text{bool}):$ em_lift[Physical_memory, Device_state, list[Byte]] (memory_write_side_effect(pm_phy' mem)(a, bl, cp)) ## $(\lambda (\text{bl1}: \text{list}[\text{Byte}]): \text{device_write_side_effect}'(a, \text{bl1}, \text{cp}))$)
{1}	side_effect_content_unchanged(pm' rw_addr, pm' states, $\lambda (a: \text{Address}, \text{bl}: \text{list}[\text{Byte}], \text{cp}: \text{bool}):$ em_lift[Physical_memory, De- vice_state, list[Byte]] (memory_write_side_effect(pm_phy' mem)(a, bl, cp))
{2}	side_effect_content_unchanged(pm' rw_addr, pm' states, $\lambda (a: \text{Address}, \text{bl}: \text{list}[\text{Byte}], \text{cp}: \text{bool}):$ em_lift[Physical_memory, De- vice_state, list[Byte]] (memory_write_side_effect(pm_phy' mem)(a, bl, cp)) ## $(\lambda (\text{bl1}: \text{list}[\text{Byte}]):$ device_write_side_effect'(a, bl1, cp))

Hiding formulas: 2,

Using lemma pm_dev_plain,

Using lemma pm_dev_states,

Using lemma pm_dev_ro_rw_addr,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Using lemma side_effect_content_unchanged_mono[Device_memory],

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 2 subgoals:

device_plain_write_side_effect_unchanged.2.1:

{-1}	is_device_plain_memory?(device_read_side_effect', device_write_side_effect')(pm')
{-2}	$((\text{pm}'\text{ro_addr} \cup \text{pm}'\text{rw_addr}) \subseteq (\text{pm_phy}'\text{ro_addr} \cup \text{pm_phy}'\text{rw_addr}))$
{-3}	$\text{pm}'\text{states} = \text{em_lift}(\text{pm_phy}'\text{states})$
{-4}	plain_memory?(pm_phy)
{-5}	memory_write_side_effect(pm' mem) = $(\lambda (a: \text{Address}, \text{bl}: \text{list}[\text{Byte}], \text{cp}: \text{bool}):$ em_lift[Physical_memory, Device_state, list[Byte]] (memory_write_side_effect(pm_phy' mem)(a, bl, cp)) ## $(\lambda (\text{bl1}: \text{list}[\text{Byte}]): \text{device_write_side_effect}'(a, \text{bl1}, \text{cp}))$)
{1}	side_effect_content_unchanged(pm_phy' rw_addr, pm' states, $\lambda (a: \text{Address}, \text{bl}: \text{list}[\text{Byte}], \text{cp}: \text{bool}):$ em_lift[Physical_memory, De- vice_state, list[Byte]] (memory_write_side_effect(pm_phy' mem)(a, bl, cp))
{2}	side_effect_content_unchanged(pm' rw_addr, pm' states, $\lambda (a: \text{Address}, \text{bl}: \text{list}[\text{Byte}], \text{cp}: \text{bool}):$ em_lift[Physical_memory, De- vice_state, list[Byte]] (memory_write_side_effect(pm_phy' mem)(a, bl, cp))

Replacing using formula -3,

Rewriting using em_side_effect_content_unchanged, matching in *,

Expanding the definition of plain_memory?,

which is trivially true.

This completes the proof of device_plain_write_side_effect_unchanged.2.1.

device_plain_write_side_effect_unchanged.2.2:

<pre> {-1} is_device_plain_memory?(device_read_side_effect', device_write_side_effect')(pm') {-2} ((pm'ro_addr ∪ pm'rw_addr) ⊆ (pm_phy'ro_addr ∪ pm_phy'rw_addr)) {-3} pm'states = em_lift(pm_phy'states) {-4} plain_memory?(pm_phy) {-5} memory_write_side_effect(pm'mem) = (λ (a: Address, bl: list[Byte], cp: bool): em_lift[Physical_memory, Device_state, list[Byte]] (memory_write_side_effect(pm_phy'mem)(a, bl, cp)) ## (λ (bl1: list[Byte]): device_write_side_effect'(a, bl1, cp))) </pre>	<pre> {1} (pm'rw_addr ⊆ pm_phy'rw_addr) {2} side_effect_content_unchanged(pm'rw_addr, pm'states, λ (a: Address, bl: list[Byte], cp: bool): em_lift[Physical_memory, Device_state, list[Byte]] (memory_write_side_effect(pm_phy'mem)(a, bl, cp))) </pre>
--	---

Expanding the definition of is_device_plain_memory?,

which is trivially true.

This completes the proof of device_plain_write_side_effect_unchanged.2.2.

device_plain_write_side_effect_unchanged.3:

<pre> {-1} is_device_plain_memory?(device_read_side_effect', device_write_side_effect')(pm') {-2} memory_write_side_effect(pm'mem) = (λ (a: Address, bl: list[Byte], cp: bool): em_lift[Physical_memory, Device_state, list[Byte]] (memory_write_side_effect(pm_phy'mem)(a, bl, cp)) ## (λ (bl1: list[Byte]): device_write_side_effect'(a, bl1, cp))) </pre>	<pre> {1} side_effect_content_unchanged(pm'rw_addr, pm'states, device_write_side_effect') {2} side_effect_content_unchanged(pm'rw_addr, pm'states, λ (a: Address, bl: list[Byte], cp: bool): em_lift[Physical_memory, Device_state, list[Byte]] (memory_write_side_effect(pm_phy'mem)(a, bl, cp)) ## (λ (bl1: list[Byte]): device_write_side_effect'(a, bl1, cp))) </pre>
--	---

Hiding formulas: 2,

Expanding the definition of is_device_plain_memory?,

which is trivially true.

This completes the proof of device_plain_write_side_effect_unchanged.3.

Q.E.D.

C.22.15 Challenge_Device_Memory.device_plain_transformers_ok2

Terse proof for device_plain_transformers_ok2.

C Proof scripts

device_plain_transformers_ok2:

<pre> {1} ∃ (device_read_side_effect, device_write_side_effect: [[Address, list[Byte], bool] → [Device_memory[Physical_memory, pm_phy, Device_state] → ExprResult[Device_memory, list[Byte]]]], pm: Plain_Memory[Device_memory]): is_device_plain_memory?(device_read_side_effect, device_write_side_effect)(pm) ⊃ transformers_ok?(pm' states, (memory_read_side_effect_super_transformers(pm' mem, (pm' ro_addr ∪ pm' rw_addr))) </pre>
--

Repeatedly Skolemizing and flattening,

Rewriting using transformers_ok_all_transformers, matching in *,

Repeatedly Skolemizing and flattening,

Expanding the definition of union,

Expanding the definition of member,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 2 subgoals:

device_plain_transformers_ok2.1:

<pre> {-1} memory_read_side_effect_super_transformers(pm' mem, (pm' ro_addr ∪ pm' rw_addr))(q') {-2} is_device_plain_memory?(device_read_side_effect', device_write_side_effect')(pm') </pre> <hr/> <pre> {1} transformers_ok?(pm' states, singleton(q')) {2} transformers_ok?(pm' states, (memory_read_side_effect_super_transformers(pm' mem, (pm' ro_addr ∪ pm' rw_addr))) </pre>
--

Expanding the definition of memory_read_side_effect_super_transformers,

Repeatedly Skolemizing and flattening,

Replacing using formula -3,

Keeping (-2 -4 1) and hiding *,

Using lemma pm_dev_read_side_effect,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Replacing using formula -2,

Simplifying, rewriting, and recording with decision procedures,

Rewriting using fexpr_composition_transformers_ok, matching in * where states gets pm!1'states, P gets LAMBDA (bl: list[Byte]): bl = bl!1,

we get 4 subgoals:

device_plain_transformers_ok2.1.1:

<pre> {-1} is_device_plain_memory?(device_read_side_effect', device_write_side_effect')(pm') {-2} memory_read_side_effect(pm'`mem) = (λ (a: Address, bl: list[Byte], cp: bool): em_lift[Physical_memory, Device_state, list[Byte]] (memory_read_side_effect(pm_phy`mem)(a, bl, cp)) ## (λ (bl1: list[Byte]): device_read_side_effect'(a, bl1, cp))) {-3} (address_block(a', length(bl')) ⊆ (pm'`ro_addr ∪ pm'`rw_addr)) </pre>	<pre> {1} transformers_ok?(pm'`states, singleton(expr_2_super(em_lift[Physical_memory, Device_state, list[Byte]] (memory_read_side_effect(pm_phy`mem) (a', bl', cp'))))) {2} transformers_ok?(pm'`states, singleton(expr_2_super(em_lift[Physical_memory, Device_state, list[Byte]] (memory_read_side_effect(pm_phy`mem) (a', bl', cp'))))) ## (λ (bl1: list[Byte]): device_read_side_effect'(a', bl1, cp')))) </pre>
---	--

Rewriting using em_lift_expr_2_super, matching in *

Rewriting using em_lift_singleton, matching in *

Using lemma pm_dev_states,

Using lemma pm_dev_plain,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Using lemma em_transformers_ok[Physical_memory, Device_state],

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Using lemma plain_memory_transformers_ok_read_side_effects_ro_rw,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Keeping (-2 -3 -6 1) and hiding *,

Rewriting using transformers_ok_mono_transformers[Physical_memory], matching in * where transformers_1 gets singleton(expr_2_super(memory_read_side_effect(pm_phy`mem) (a!1, bl!1, cp!1))), transformers_2 gets memory_read_side_effect_super_transformers(pm_phy`mem, union (pm_phy`ro_addr, pm_phy`rw_addr)),

Using lemma pm_dev_ro_rw_addr,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Keeping (-2 -4 1) and hiding *,

Expanding the definition of subset?,

Repeatedly Skolemizing and flattening,

Expanding the definition of member,

Expanding the definition of singleton,

Expanding the definition of memory_read_side_effect_super_transformers,

Instantiating quantified variables,

Rewriting using subset_transitive, matching in * where a gets address_block(a!1, length(bl!1)), c gets union(pm_phy`ro_addr, pm_phy`rw_addr), b gets union(pm!1`ro_addr, pm!1`rw_addr),

This completes the proof of device_plain_transformers_ok2.1.1.

device_plain_transformers_ok2.1.2:

{-1}	is_device_plain_memory?(device_read_side_effect', device_write_side_effect')(pm')
{-2}	memory_read_side_effect(pm' mem) = $(\lambda (a: \text{Address}, bl: \text{list}[\text{Byte}], cp: \text{bool}):$ $\text{em_lift}[\text{Physical_memory}, \text{Device_state}, \text{list}[\text{Byte}]]$ $(\text{memory_read_side_effect}(pm_phy' \text{mem})(a, bl, cp))$ $\#\# (\lambda (bl1: \text{list}[\text{Byte}]): \text{device_read_side_effect}'(a, bl1, cp)))$
{-3}	$(\text{address_block}(a', \text{length}(bl')) \subseteq (pm' \text{ro_addr} \cup pm' \text{rw_addr}))$
{1}	transformer_invariant?(pm' states, $\text{singleton}(\text{expr_2_super}(\text{em_lift}$ $[\text{Physical_memory}, \text{De-}$ $\text{vice_state}, \text{list}[\text{Byte}]]$ $(\text{memory_read_side_effect}(pm_phy' \text{mem})$ $(a', bl', cp'))))$
{2}	transformers_ok?(pm' states, $\text{singleton}(\text{expr_2_super}(\text{em_lift}[\text{Physical_memory}, \text{De-}$ $\text{vice_state}, \text{list}[\text{Byte}]]$ $(\text{memory_read_side_effect}(pm_phy' \text{mem})$ $(a', bl', cp'))$ $\#\#$ $(\lambda (bl1: \text{list}[\text{Byte}]):$ $\text{device_read_side_effect}'(a', bl1, cp'))))$

Hiding formulas: 2,

Rewriting using em_lift_expr_2_super, matching in *,

Rewriting using em_lift_singleton, matching in *,

Using lemma pm_dev_states,

Using lemma pm_dev_plain,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Replacing using formula -3,

Rewriting using em_transformer_invariant, matching in *,

Using lemma plain_memory_invariant,

Using lemma transformer_invariant_mono_transformers,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Using lemma pm_dev_ro_rw_addr,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Keeping (-2 -7 1) and hiding *,

Expanding the definition of subset?,

Expanding the definition of singleton,

Expanding the definition of union,

Expanding the definition of member,

Repeatedly Skolemizing and flattening,

Expanding the definition of memory_read_side_effect_super_transformers,

Instantiating quantified variables,

Rewriting using subset_transitive, matching in * where a gets address_block(a!1, length(bl!1)), c gets union(pm_phy'ro_addr, pm_phy'rw_addr), b gets union(pm!1'ro_addr, pm!1'rw_addr),

This completes the proof of device_plain_transformers_ok2.1.2.

device_plain_transformers_ok2.1.3:

<p>{-1} is_device_plain_memory?(device_read_side_effect', device_write_side_effect')(pm')</p> <p>{-2} memory_read_side_effect(pm' 'mem) = (λ (a: Address, bl: list[Byte], cp: bool): em_lift[Physical_memory, Device_state, list[Byte]] (memory_read_side_effect(pm_phy' 'mem)(a, bl, cp)) ## (λ (bl1: list[Byte]): device_read_side_effect'(a, bl1, cp)))</p> <p>{-3} (address_block(a', length(bl')) ⊆ (pm' 'ro_addr ∪ pm' 'rw_addr))</p> <hr style="border: 0.5px solid black;"/> <p>{1} ∇ (s: (pm' 'states)): OK?(em_lift[Physical_memory, Device_state, list[Byte]] (memory_read_side_effect(pm_phy' 'mem)(a', bl', cp'))(s)) ⊃ data(em_lift[Physical_memory, Device_state, list[Byte]] (memory_read_side_effect(pm_phy' 'mem)(a', bl', cp'))(s)) = bl'</p> <p>{2} transformers_ok?(pm' 'states, singleton(expr_2_super(em_lift[Physical_memory, Device_state, list[Byte]] (memory_read_side_effect(pm_phy' 'mem) (a', bl', cp')) ## (λ (bl1: list[Byte]): device_read_side_effect'(a', bl1, cp'))))))</p>	
---	--

Repeatedly Skolemizing and flattening,

Expanding the definition of em_lift,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Hiding formulas: (-3 -5 2),

Using lemma pm_dev_plain,

Using lemma pm_dev_ro_rw_addr,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Expanding the definition of plain_memory?,

Applying disjunctive simplification to flatten sequent,

Using lemma side_effect_content_unchanged_content,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 2 subgoals:

device_plain_transformers_ok2.1.3.1:

{-1}	side_effect_content_unchanged((pm_phy'ro_addr ∪ pm_phy'rw_addr), pm_phy'states, memory_read_side_effect(pm_phy'mem))
{-2}	OK?(memory_read_side_effect(pm_phy'mem)(a', bl', cp')(s'state))
{-3}	is_device_plain_memory?(device_read_side_effect', device_write_side_effect')(pm')
{-4}	((pm'ro_addr ∪ pm'rw_addr) ⊆ (pm_phy'ro_addr ∪ pm_phy'rw_addr))
{-5}	unchanged_memory_invariant?(pm_phy'mem, pm_phy'states, ((pm_phy'other_actions ∪ memory_read_transformers(pm_phy'mem, pm_phy'ro_addr ∪ pm_phy'rw_addr)))
{-6}	unchanged_memory_invariant?(pm_phy'mem, pm_phy'states, memory_write_transformers(pm_phy'mem, pm_phy'rw_addr, pm_phy'ro_addr))
{-7}	unchanged_memory_write_invariant?(pm_phy'mem, pm_phy'states, pm_phy'rw_addr)
{-8}	changed_memory_invariant?(pm_phy'mem, pm_phy'states, pm_phy'rw_addr)
{-9}	transformers_ok?(pm_phy'states, ((memory_read_transformers(pm_phy'mem, (pm_phy'ro_addr ∪ pm_phy'rw_addr))))
{-10}	side_effect_content_unchanged(pm_phy'rw_addr, pm_phy'states, memory_write_side_effect(pm_phy'mem))
{-11}	pm'states(s')
{-12}	(address_block(a', length(bl')) ⊆ (pm'ro_addr ∪ pm'rw_addr))
{1}	(address_block(a', length(bl')) ⊆ (pm_phy'ro_addr ∪ pm_phy'rw_addr))
{2}	data(memory_read_side_effect(pm_phy'mem)(a', bl', cp')(s'state)) = bl'

Rewriting using subset_transitive, matching in * where a gets address_block(a!1, length(bl!1)), c gets union(pm_phy'ro_addr, pm_phy'rw_addr), b gets union(pm!1'ro_addr, pm!1'rw_addr),

This completes the proof of device_plain_transformers_ok2.1.3.1.

device_plain_transformers_ok2.1.3.2:

{-1}	side_effect_content_unchanged((pm_phy'ro_addr ∪ pm_phy'rw_addr), pm_phy'states, memory_read_side_effect(pm_phy'mem))
{-2}	OK?(memory_read_side_effect(pm_phy'mem)(a', bl', cp')(s'state))
{-3}	is_device_plain_memory?(device_read_side_effect', device_write_side_effect')(pm')
{-4}	((pm'ro_addr ∪ pm'rw_addr) ⊆ (pm_phy'ro_addr ∪ pm_phy'rw_addr))
{-5}	unchanged_memory_invariant?(pm_phy'mem, pm_phy'states, ((pm_phy'other_actions ∪ memory_read_transformers(pm_phy'mem, pm_phy'ro_addr ∪ pm_phy'rw_addr)))
{-6}	unchanged_memory_invariant?(pm_phy'mem, pm_phy'states, memory_write_transformers(pm_phy'mem, pm_phy'rw_addr, pm_phy'ro_addr))
{-7}	unchanged_memory_write_invariant?(pm_phy'mem, pm_phy'states, pm_phy'rw_addr)
{-8}	changed_memory_invariant?(pm_phy'mem, pm_phy'states, pm_phy'rw_addr)
{-9}	transformers_ok?(pm_phy'states, ((memory_read_transformers(pm_phy'mem, (pm_phy'ro_addr ∪ pm_phy'rw_addr))))
{-10}	side_effect_content_unchanged(pm_phy'rw_addr, pm_phy'states, memory_write_side_effect(pm_phy'mem))
{-11}	pm'states(s')
{-12}	(address_block(a', length(bl')) ⊆ (pm'ro_addr ∪ pm'rw_addr))
{1}	pm_phy'states(s'state)
{2}	data(memory_read_side_effect(pm_phy'mem)(a', bl', cp')(s'state)) = bl'

Using lemma pm_dev_states,

Expanding the definition of em_lift,

Replacing using formula -1,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of device_plain_transformers_ok2.1.3.2.

device_plain_transformers_ok2.1.4:

<p>{-1} is_device_plain_memory?(device_read_side_effect', device_write_side_effect')(pm')</p> <p>{-2} memory_read_side_effect(pm' 'mem) = (λ (a: Address, bl: list[Byte], cp: bool): em_lift[Physical_memory, Device_state, list[Byte]] (memory_read_side_effect(pm_phy' 'mem)(a, bl, cp)) ## (λ (bl1: list[Byte]): device_read_side_effect'(a, bl1, cp)))</p> <p>{-3} (address_block(a', length(bl')) ⊆ (pm' 'ro_addr ∪ pm' 'rw_addr))</p>	<hr style="border: 0.5px solid black;"/> <p>{1} ∀ (d: (λ (bl: list[Byte]): bl = bl')): transformers_ok?(pm' 'states, singleton(expr_2_super(device_read_side_effect'(a', d, cp'))))</p> <p>{2} transformers_ok?(pm' 'states, singleton(expr_2_super(em_lift[Physical_memory, Device_state, list[Byte]] (memory_read_side_effect(pm_phy' 'mem) (a', bl', cp')) ## (λ (bl1: list[Byte]): device_read_side_effect'(a', bl1, cp')))))</p>
---	---

Repeatedly Skolemizing and flattening,
 Expanding the definition of is_device_plain_memory?,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 Using lemma transformers_ok_mono_transformers,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 Keeping (-14 1) and hiding *,
 Expanding the definition of subset?,
 Expanding the definition of member,
 Repeatedly Skolemizing and flattening,
 Expanding the definition of singleton,
 Expanding the definition of drse_super_transformers,
 Expanding the definition of union,
 Expanding the definition of member,
 Applying disjunctive simplification to flatten sequent,
 Instantiating quantified variables,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 This completes the proof of device_plain_transformers_ok2.1.4.

device_plain_transformers_ok2.2:

<p>{-1} memory_write_side_effect_super_transformers(pm' 'mem, pm' 'rw_addr)(q')</p> <p>{-2} is_device_plain_memory?(device_read_side_effect', device_write_side_effect')(pm')</p>	<hr style="border: 0.5px solid black;"/> <p>{1} transformers_ok?(pm' 'states, singleton(q'))</p> <p>{2} transformers_ok?(pm' 'states, (memory_read_side_effect_super_transformers(pm' 'mem, (pm' 'ro_addr ∪ pm' 'rw_addr)) ∪</p>
---	---

Hiding formulas: 2,
 Expanding the definition of memory_write_side_effect_super_transformers,
 Repeatedly Skolemizing and flattening,
 Replacing using formula -3,
 Keeping (-2 -4 1) and hiding *,
 Using lemma pm_dev_write_side_effect,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

C Proof scripts

Replacing using formula -2,
 Simplifying, rewriting, and recording with decision procedures,
 Rewriting using `fexpr_composition_transformers_ok`, matching in `*` where states gets `pm!1'states`, P
 gets `LAMBDA (bl: list[Byte]): bl = bl!1`,
 we get 4 subgoals:
`device_plain_transformers_ok2.2.1:`

<pre>{-1} is_device_plain_memory?(device_read_side_effect', device_write_side_effect')(pm')</pre>	<pre>memory_write_side_effect(pm' 'mem) =</pre>
<pre>{-2}</pre>	<pre>(λ (a: Address, bl: list[Byte], cp: bool): em_lift[Physical_memory, Device_state, list[Byte]] (memory_write_side_effect(pm_phy 'mem)(a, bl, cp)) ## (λ (bl1: list[Byte]): device_write_side_effect'(a, bl1, cp)))</pre>
<pre>{-3}</pre>	<pre>(address_block(a', length(bl')) ⊆ pm'rw_addr)</pre>
<pre>{1}</pre>	<pre>transformers_ok?(pm' 'states, singleton(expr_2_super(em_lift[Physical_memory, De- vice_state, list[Byte]] (memory_write_side_effect(pm_phy 'mem) (a', bl', cp')))))</pre>
<pre>{2}</pre>	<pre>transformers_ok?(pm' 'states, singleton(expr_2_super(em_lift[Physical_memory, De- vice_state, list[Byte]] (memory_write_side_effect(pm_phy 'mem) (a', bl', cp')))) ## (λ (bl1: list[Byte]): device_write_side_effect'(a', bl1, cp'))))</pre>

Rewriting using `em_lift_expr_2_super`, matching in `*`,
 Rewriting using `em_lift_singleton`, matching in `*`,
 Using lemma `pm_dev_states`,
 Using lemma `pm_dev_plain`,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 Using lemma `em_transformers_ok[Physical_memory, Device_state]`,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 Using lemma `plain_memory_transformers_ok_write_side_effects_ro_rw`,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 Keeping (-2 -3 -6 1) and hiding `*`,
 Rewriting using `transformers_ok_mono_transformers[Physical_memory]`, matching in `*` where `trans-
 formers_1` gets `singleton(expr_2_super(memory_write_side_effect(pm_phy 'mem) (a!1, bl!1, cp!1)))`, `trans-
 formers_2` gets `memory_write_side_effect_super_transformers(pm_phy 'mem, pm_phy 'rw_addr)`,
 Keeping (-2 -3 1) and hiding `*`,
 Expanding the definition of `subset?`,
 Repeatedly Skolemizing and flattening,
 Expanding the definition of `member`,
 Expanding the definition of `singleton`,
 Expanding the definition of `memory_write_side_effect_super_transformers`,
 Instantiating quantified variables,
 Rewriting using `subset_transitive`, matching in `*` where `a` gets `address_block(a!1, length(bl!1))`, `c` gets
`pm_phy 'rw_addr`, `b` gets `pm!1 'rw_addr`,
 Expanding the definition of `is_device_plain_memory?`,
 which is trivially true.
 This completes the proof of `device_plain_transformers_ok2.2.1`.

device_plain_transformers_ok2.2.2:

{-1}	is_device_plain_memory?(device_read_side_effect', device_write_side_effect')(pm')
{-2}	memory_write_side_effect(pm' mem) = $(\lambda (a: \text{Address}, bl: \text{list}[\text{Byte}], cp: \text{bool}):$ em_lift[Physical_memory, Device_state, list[Byte]] (memory_write_side_effect(pm_phy' mem)(a, bl, cp)) ## $(\lambda (bl1: \text{list}[\text{Byte}]): \text{device_write_side_effect}'(a, bl1, cp))$)
{-3}	(address_block(a', length(bl')) \subseteq pm'rw_addr)
{1}	transformer_invariant?(pm' states, singleton(expr_2_super(em_lift [Physical_memory, De- vice_state, list[Byte]] (memory_write_side_effect(pm_phy' mem) (a', bl', cp')))))
{2}	transformers_ok?(pm' states, singleton(expr_2_super(em_lift[Physical_memory, De- vice_state, list[Byte]] (memory_write_side_effect(pm_phy' mem) (a', bl', cp')) ## $(\lambda (bl1: \text{list}[\text{Byte}]):$ device_write_side_effect'(a', bl1, cp')))))

Hiding formulas: 2,

Rewriting using em_lift_expr_2_super, matching in *

Rewriting using em_lift_singleton, matching in *

Using lemma pm_dev_states,

Using lemma pm_dev_plain,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Replacing using formula -3,

Rewriting using em_transformer_invariant, matching in *

Using lemma plain_memory_invariant,

Using lemma transformer_invariant_mono_transformers,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Keeping (-3 -6 1) and hiding *,

Expanding the definition of subset?,

Expanding the definition of singleton,

Expanding the definition of union,

Expanding the definition of member,

Repeatedly Skolemizing and flattening,

Expanding the definition of memory_write_side_effect_super_transformers,

Instantiating quantified variables,

Rewriting using subset_transitive, matching in * where a gets address_block(a!1, length(bl!1)), c gets pm_phy'rw_addr, b gets pm!1'rw_addr,

Expanding the definition of is_device_plain_memory?,

which is trivially true.

This completes the proof of device_plain_transformers_ok2.2.2.

C Proof scripts

device_plain_transformers_ok2.2.3:

<pre> {-1} is_device_plain_memory?(device_read_side_effect', device_write_side_effect')(pm') {-2} memory_write_side_effect(pm' mem) = (λ (a: Address, bl: list[Byte], cp: bool): em_lift[Physical_memory, Device_state, list[Byte]] (memory_write_side_effect(pm_phy' mem)(a, bl, cp)) ## (λ (bl1: list[Byte]): device_write_side_effect'(a, bl1, cp))) {-3} (address_block(a', length(bl')) ⊆ pm'rw_addr) </pre>	<hr style="border: 0.5px solid black;"/> <pre> {1} ∀ (s: (pm' states)): OK?(em_lift[Physical_memory, Device_state, list[Byte]] (memory_write_side_effect(pm_phy' mem)(a', bl', cp'))(s)) ⊃ data(em_lift[Physical_memory, Device_state, list[Byte]] (memory_write_side_effect(pm_phy' mem)(a', bl', cp'))(s)) = bl' {2} transformers_ok?(pm' states, singleton(expr_2_super(em_lift[Physical_memory, De- vice_state, list[Byte]] (memory_write_side_effect(pm_phy' mem) (a', bl', cp')) ## (λ (bl1: list[Byte]): device_write_side_effect'(a', bl1, cp'))))) </pre>
--	--

Repeatedly Skolemizing and flattening,

Expanding the definition of em_lift,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Hiding formulas: (-3 -5 2),

Using lemma pm_dev_plain,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Expanding the definition of plain_memory?,

Applying disjunctive simplification to flatten sequent,

Using lemma side_effect_content_unchanged_content,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 2 subgoals:

device_plain_transformers_ok2.2.3.1:

{-1}	side_effect_content_unchanged(pm_phy'rw_addr, pm_phy'states, memory_write_side_effect(pm_phy'mem))
{-2}	OK?(memory_write_side_effect(pm_phy'mem)(a', bl', cp')(s'state))
{-3}	is_device_plain_memory?(device_read_side_effect', device_write_side_effect')(pm')
{-4}	unchanged_memory_invariant?(pm_phy'mem, pm_phy'states, ((pm_phy'other_actions ∪ memory_read_transformers(pm_phy'mem, (pm_phy'ro_addr ∪ pm_phy'rw_addr)))
{-5}	unchanged_memory_invariant?(pm_phy'mem, pm_phy'states, memory_write_transformers(pm_phy'mem, pm_phy'rw_addr), pm_phy'ro_addr)
{-6}	unchanged_memory_write_invariant?(pm_phy'mem, pm_phy'states, pm_phy'rw_addr)
{-7}	changed_memory_invariant?(pm_phy'mem, pm_phy'states, pm_phy'rw_addr)
{-8}	transformers_ok?(pm_phy'states, ((memory_read_transformers(pm_phy'mem, (pm_phy'ro_addr ∪ pm_phy'rw_addr)) ∪ me
{-9}	side_effect_content_unchanged((pm_phy'ro_addr ∪ pm_phy'rw_addr), pm_phy'states, memory_read_side_effect(pm_phy'mem))
{-10}	pm'states(s')
{-11}	(address_block(a', length(bl')) ⊆ pm'rw_addr)
{1}	(address_block(a', length(bl')) ⊆ pm_phy'rw_addr)
{2}	data(memory_write_side_effect(pm_phy'mem)(a', bl', cp')(s'state)) = bl'

Rewriting using subset_transitive, matching in * where a gets address_block(a!1, length(bl!1)), c gets pm_phy'rw_addr, b gets pm!1'rw_addr,

Expanding the definition of is_device_plain_memory?, which is trivially true.

This completes the proof of device_plain_transformers_ok2.2.3.1.

device_plain_transformers_ok2.2.3.2:

{-1}	side_effect_content_unchanged(pm_phy'rw_addr, pm_phy'states, memory_write_side_effect(pm_phy'mem))
{-2}	OK?(memory_write_side_effect(pm_phy'mem)(a', bl', cp')(s'state))
{-3}	is_device_plain_memory?(device_read_side_effect', device_write_side_effect')(pm')
{-4}	unchanged_memory_invariant?(pm_phy'mem, pm_phy'states, ((pm_phy'other_actions ∪ memory_read_transformers(pm_phy'mem, (pm_phy'ro_addr ∪ pm_phy'rw_addr)))
{-5}	unchanged_memory_invariant?(pm_phy'mem, pm_phy'states, memory_write_transformers(pm_phy'mem, pm_phy'rw_addr), pm_phy'ro_addr)
{-6}	unchanged_memory_write_invariant?(pm_phy'mem, pm_phy'states, pm_phy'rw_addr)
{-7}	changed_memory_invariant?(pm_phy'mem, pm_phy'states, pm_phy'rw_addr)
{-8}	transformers_ok?(pm_phy'states, ((memory_read_transformers(pm_phy'mem, (pm_phy'ro_addr ∪ pm_phy'rw_addr)) ∪ me
{-9}	side_effect_content_unchanged((pm_phy'ro_addr ∪ pm_phy'rw_addr), pm_phy'states, memory_read_side_effect(pm_phy'mem))
{-10}	pm'states(s')
{-11}	(address_block(a', length(bl')) ⊆ pm'rw_addr)
{1}	pm_phy'states(s'state)
{2}	data(memory_write_side_effect(pm_phy'mem)(a', bl', cp')(s'state)) = bl'

Using lemma pm_dev_states,

Expanding the definition of em_lift,

Replacing using formula -1,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

C Proof scripts

This completes the proof of `device_plain_transformers_ok2.2.3.2`.

`device_plain_transformers_ok2.2.4`:

```
{-1} is_device_plain_memory?(device_read_side_effect', device_write_side_effect')(pm')
{-2} memory_write_side_effect(pm' 'mem) =
      (λ (a: Address, bl: list[Byte], cp: bool):
         em_lift[Physical_memory, Device_state, list[Byte]]
           (memory_write_side_effect(pm_phy 'mem)(a, bl, cp))
         ## (λ (bl1: list[Byte]): device_write_side_effect'(a, bl1, cp)))
{-3} (address_block(a', length(bl')) ⊆ pm' 'rw_addr)
-----
{1}  ∀ (d: (λ (bl: list[Byte]): bl = bl')):
      transformers_ok?(pm' 'states,
                       singleton(expr_2_super(device_write_side_effect'(a', d, cp'))))
{2}  transformers_ok?(pm' 'states,
                       singleton(expr_2_super(em_lift[Physical_memory, De-
                                               vice_state, list[Byte]]
                                               (memory_write_side_effect(pm_phy 'mem)
                                               (a', bl', cp'))
                                               ##
                                               (λ (bl1: list[Byte]):
                                                  device_write_side_effect'(a', bl1, cp')))))
```

Repeatedly Skolemizing and flattening,

Expanding the definition of `is_device_plain_memory?`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Using lemma `transformers_ok_mono_transformers`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Keeping (-14 1) and hiding *,

Expanding the definition of `subset?`,

Expanding the definition of `member`,

Repeatedly Skolemizing and flattening,

Expanding the definition of `singleton`,

Expanding the definition of `dwse_super_transformers`,

Expanding the definition of `union`,

Expanding the definition of `member`,

Applying disjunctive simplification to flatten sequent,

Instantiating quantified variables,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `device_plain_transformers_ok2.2.4`.

Q.E.D.

C.22.16 Challenge_Device_Memory.device_memory_plain_memory

Terse proof for `device_memory_plain_memory`.

device_memory_plain_memory:

$\{1\} \quad \forall (\text{device_read_side_effect},$ $\quad \text{device_write_side_effect}:$ $\quad \quad [[\text{Address}, \text{list}[\text{Byte}], \text{bool}] \rightarrow$ $\quad \quad \quad [\text{Device_memory}[\text{Physical_memory}, \text{pm_phy}, \text{Device_state}] \rightarrow$ $\quad \quad \quad \quad \text{ExprResult}[\text{Device_memory}, \text{list}[\text{Byte}]]],$ $\quad \text{pm}):$ $\quad \text{is_device_plain_memory?}(\text{device_read_side_effect}, \text{device_write_side_effect})(\text{pm}) \supset$ $\quad \text{plain_memory?}(\text{pm})$
--

Expanding the definition of plain_memory?,
 Repeatedly Skolemizing and flattening,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 we get 7 subgoals:

device_memory_plain_memory.1:

$\{-1\} \quad \text{is_device_plain_memory?}(\text{device_read_side_effect}', \text{device_write_side_effect}')(\text{pm}')$
$\{1\} \quad \text{side_effect_content_unchanged}(\text{pm}'\text{'rw_addr}, \text{pm}'\text{'states}, \text{mem-}$ $\quad \text{ory_write_side_effect}(\text{pm}'\text{'mem}))$

Using lemma device_plain_write_side_effect_unchanged,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 This completes the proof of device_memory_plain_memory.1.

device_memory_plain_memory.2:

$\{-1\} \quad \text{is_device_plain_memory?}(\text{device_read_side_effect}', \text{device_write_side_effect}')(\text{pm}')$
$\{1\} \quad \text{side_effect_content_unchanged}((\text{pm}'\text{'ro_addr} \cup \text{pm}'\text{'rw_addr}), \text{pm}'\text{'states},$ $\quad \text{memory_read_side_effect}(\text{pm}'\text{'mem}))$

Using lemma device_plain_read_side_effect_unchanged,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 This completes the proof of device_memory_plain_memory.2.

device_memory_plain_memory.3:

$\{-1\} \quad \text{is_device_plain_memory?}(\text{device_read_side_effect}', \text{device_write_side_effect}')(\text{pm}')$
$\{1\} \quad \text{transformers_ok?}(\text{pm}'\text{'states},$ $\quad ((\text{memory_read_transformers}(\text{pm}'\text{'mem}, (\text{pm}'\text{'ro_addr} \cup \text{pm}'\text{'rw_addr})) \cup \text{memory_write_t}$

Using lemma device_plain_transformers_ok,
 Using lemma device_plain_transformers_ok2,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 Rewriting using transformers_ok_union_transformers, matching in * where states gets
 pm!1'states, transformers_1 gets union(memory_read_transformers(pm!1'mem, union(pm!1'ro_addr,
 pm!1'rw_addr)), memory_write_transformers(pm!1'mem, pm!1'rw_addr)), transformers_2 gets
 union(memory_read_side_effect_super_transformers(pm!1'mem, union(pm!1'ro_addr, pm!1'rw_addr)),
 memory_write_side_effect_super_transformers(pm!1'mem, pm!1'rw_addr)),

This completes the proof of device_memory_plain_memory.3.

device_memory_plain_memory.4:

$\{-1\} \quad \text{is_device_plain_memory?}(\text{device_read_side_effect}', \text{device_write_side_effect}')(\text{pm}')$
$\{1\} \quad \text{changed_memory_invariant?}(\text{pm}'\text{'mem}, \text{pm}'\text{'states}, \text{pm}'\text{'rw_addr})$

Using lemma device_plain_changed_memory_invariant,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 This completes the proof of device_memory_plain_memory.4.

device_memory_plain_memory.5:

$$\frac{\{-1\} \text{ is_device_plain_memory?}(\text{device_read_side_effect}', \text{device_write_side_effect}')(\text{pm}')}{\{1\} \text{ unchanged_memory_write_invariant?}(\text{pm}'\text{'mem}, \text{pm}'\text{'states}, \text{pm}'\text{'rw_addr})}$$

Using lemma device_plain_unchanged_memory_write_invariant,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of device_memory_plain_memory.5.

device_memory_plain_memory.6:

$$\frac{\{-1\} \text{ is_device_plain_memory?}(\text{device_read_side_effect}', \text{device_write_side_effect}')(\text{pm}')}{\{1\} \text{ unchanged_memory_invariant?}(\text{pm}'\text{'mem}, \text{pm}'\text{'states}, \text{memory_write_transformers}(\text{pm}'\text{'mem}, \text{pm}'\text{'rw_addr}), \text{pm}'\text{'ro_addr})}$$

Using lemma device_plain_unchanged_memory_invariant_write,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of device_memory_plain_memory.6.

device_memory_plain_memory.7:

$$\frac{\{-1\} \text{ is_device_plain_memory?}(\text{device_read_side_effect}', \text{device_write_side_effect}')(\text{pm}')}{\{1\} \text{ unchanged_memory_invariant?}(\text{pm}'\text{'mem}, \text{pm}'\text{'states}, ((\text{pm}'\text{'other_actions} \cup \text{memory_read_transformers}(\text{pm}'\text{'mem}, (\text{pm}'\text{'ro_addr} \cup \text{pm}'\text{'rw_addr})))$$

Using lemma device_plain_unchanged_memory_invariant,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of device_memory_plain_memory.7.
Q.E.D.

C.23 Proofs for Challenge_Random_Device (random_device.pvs)

C.23.1 Challenge_Random_Device.IMP_Random_Device_TCC1

Terse proof for IMP_Random_Device_TCC1.

IMP_Random_Device_TCC1:

$$\frac{}{\{1\} 0 < \text{size}(\text{uidt}(\text{dt_uint})) \wedge 3 \times \text{size}(\text{uidt}(\text{dt_uint})) \leq \text{max_linear_offset}}$$

Using lemma dt_uint_size,
This completes the proof of IMP_Random_Device_TCC1.
Q.E.D.

C.23.2

Challenge_Random_Device.in_blessed_memory_disjoint_device

Terse proof for in_blessed_memory_disjoint_device.

in_blessed_memory_disjoint_device:

$$\frac{}{\{1\} \forall (a: \text{Address}, l: \text{nat}): (\text{address_block}(a, l) \subseteq (\text{pm}'\text{'ro_addr} \cup \text{pm}'\text{'rw_addr})) \supset \text{disjoint?}(\text{address_block}(a, l), \text{address_block}(\text{base}, \text{size}))}$$

Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of `in_blessed_memory_disjoint_device`.
 Q.E.D.

C.23.3

Challenge_Random_Device.in_blessed_memory_not_device_address

Terse proof for `in_blessed_memory_not_device_address`.

`in_blessed_memory_not_device_address`:

$\{1\} \quad \forall (a: \text{Address}, \text{bl}: \text{list}[\text{Byte}]):$ $(\text{address_block}(a, \text{length}(\text{bl})) \subseteq (\text{pm}'\text{ro_addr} \cup \text{pm}'\text{rw_addr})) \wedge \text{cons}?(\text{bl}) \supset$ $a \neq \text{rnd_seed} \wedge a \neq \text{mem_cnt} \wedge a \neq \text{rnd_val}$
--

Repeatedly Skolemizing and flattening,
 Using lemma `in_blessed_memory_disjoint_device`,
 Replacing using formula -3,
 Keeping (-1 -4 1) and hiding *,
 Expanding the definition of `length`,
 Simplifying, rewriting, and recording with decision procedures,
 Adding type constraints for `base`,
 Expanding the definition of `disjoint?`,
 Expanding the definition of `empty?`,
 Instantiating the top quantifier in -5 with the terms: (a!1),
 Case splitting on `size(uidt(dt_uint)) > 0`,
 we get 2 subgoals:

`in_blessed_memory_not_device_address.1`:

$\{-1\} \quad \text{size}(\text{uidt}(\text{dt_uint})) > 0$ $\{-2\} \quad \text{Mem}?(\text{base}'\text{type_of})$ $\{-3\} \quad 0 \leq \text{base}'\text{offset}$ $\{-4\} \quad \text{base}'\text{offset} < \text{max_linear_offset}$ $\{-5\} \quad \text{base} + \text{size}[\text{Physical_memory}, \text{pm_phy}] \leq \text{max_linear}$ $\{-6\} \quad \text{cons}?(\text{bl}')$
$\{1\} \quad (a' \in (\text{address_block}(a', 1 + \text{length}(\text{cdr}(\text{bl}')))) \cap \text{address_block}(\text{base}, \text{size}))$ $\{2\} \quad a' \neq \text{rnd_seed} \wedge a' \neq \text{mem_cnt} \wedge a' \neq \text{rnd_val}$

Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of `in_blessed_memory_not_device_address.1`.

`in_blessed_memory_not_device_address.2`:

$\{-1\} \quad \text{Mem}?(\text{base}'\text{type_of})$ $\{-2\} \quad 0 \leq \text{base}'\text{offset}$ $\{-3\} \quad \text{base}'\text{offset} < \text{max_linear_offset}$ $\{-4\} \quad \text{base} + \text{size}[\text{Physical_memory}, \text{pm_phy}] \leq \text{max_linear}$ $\{-5\} \quad \text{cons}?(\text{bl}')$
$\{1\} \quad \text{size}(\text{uidt}(\text{dt_uint})) > 0$ $\{2\} \quad (a' \in (\text{address_block}(a', 1 + \text{length}(\text{cdr}(\text{bl}')))) \cap \text{address_block}(\text{base}, \text{size}))$ $\{3\} \quad a' \neq \text{rnd_seed} \wedge a' \neq \text{mem_cnt} \wedge a' \neq \text{rnd_val}$

Using lemma `dt_uchar_size`,
 Using lemma `dt_uint_size`,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `in_blessed_memory_not_device_address.2`.
Q.E.D.

C.23.4

Challenge_Random_Device.random_side_effects_unchanged_invariant

Terse proof for `random_side_effects_unchanged_invariant`.

`random_side_effects_unchanged_invariant`:

{1}	$\text{unchanged_memory_invariant?}(\text{pm}'\text{mem}, \text{pm}'\text{states},$ $(\text{drse_super_transformers}((\text{pm}'\text{ro_addr} \cup \text{pm}'\text{rw_addr}), \text{rnd_dev_r}$ $(\text{pm}'\text{ro_addr} \cup \text{pm}'\text{rw_addr}))$
-----	--

Installing automatic rewrites from: `pm random_device_pm random_device_pm_mem`
 Rewriting using `unchanged_memory_invariant_all_transformers`, matching in *,
 Hiding formulas: 2,
 Repeatedly Skolemizing and flattening,
 Rewriting using `em_const_unchanged_memory_invariant`, matching in * where `em_pm` gets `device_pm(rnd_dev_read_side_effect, rnd_dev_write_side_effect)`, `pm` gets `pm_phy'mem`,
 we get 2 subgoals:

`random_side_effects_unchanged_invariant.1`:

{-1}	$\text{union}(\text{drse_super_transformers}(((\text{pm_phy}'\text{ro_addr} \setminus \text{address_block}(\text{base}, \text{size})) \cup (\text{pm_phy}'\text{rw_addr} \setminus$ $\text{rnd_dev_read_side_effect}),$ $\text{dwse_super_transformers}((\text{pm_phy}'\text{rw_addr} \setminus \text{address_block}(\text{base}, \text{size})),$ $\text{rnd_dev_write_side_effect}))$ (q')
{1}	$(\text{device_pm}(\text{rnd_dev_read_side_effect}, \text{rnd_dev_write_side_effect})' \text{memory_read} =$ $(\lambda (a: \text{Address}):$ $\text{em_lift}[\text{Physical_memory}, \text{Random_device_state}[\text{Physical_memory}, \text{pm_phy}], \text{Byte}]$ $(\text{pm_phy}'\text{mem}' \text{memory_read}(a))))$
{2}	$\text{unchanged_memory_invariant?}(\text{device_pm}(\text{rnd_dev_read_side_effect}, \text{rnd_dev_write_side_effect}),$ $\text{em_lift}(\text{pm_phy}'\text{states}), \text{singleton}(q'),$ $((\text{pm_phy}'\text{ro_addr} \setminus \text{address_block}(\text{base}, \text{size})) \cup (\text{pm_phy}'\text{rw_ad}$

Expanding the definition of `device_pm`,
 which is trivially true.

This completes the proof of `random_side_effects_unchanged_invariant.1`.

`random_side_effects_unchanged_invariant.2`:

{-1}	$\text{union}(\text{drse_super_transformers}(((\text{pm_phy}'\text{ro_addr} \setminus \text{address_block}(\text{base}, \text{size})) \cup (\text{pm_phy}'\text{rw_addr} \setminus$ $\text{rnd_dev_read_side_effect}),$ $\text{dwse_super_transformers}((\text{pm_phy}'\text{rw_addr} \setminus \text{address_block}(\text{base}, \text{size})),$ $\text{rnd_dev_write_side_effect}))$ (q')
{1}	$\forall (s: \text{Expand_state}[\text{Physical_memory}, \text{Random_device_state}[\text{Physical_memory}, \text{pm_phy}]]):$ $\text{has_next_state}(q'(s)) \supset \text{state}(q'(s))' \text{state} = s' \text{state}$
{2}	$\text{unchanged_memory_invariant?}(\text{device_pm}(\text{rnd_dev_read_side_effect}, \text{rnd_dev_write_side_effect}),$ $\text{em_lift}(\text{pm_phy}'\text{states}), \text{singleton}(q'),$ $((\text{pm_phy}'\text{ro_addr} \setminus \text{address_block}(\text{base}, \text{size})) \cup (\text{pm_phy}'\text{rw_ad}$

Hiding formulas: 2,
 Repeatedly Skolemizing and flattening,
 Expanding the definition of `union`,

Expanding the definition of member,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 2 subgoals:

random_side_effects_unchanged_invariant.2.1:

$$\begin{array}{l|l}
 \{-1\} & \text{has_next_state}(q'(s')) \\
 \{-2\} & \text{drse_super_transformers}(((\text{pm_phy}'\text{ro_addr} \setminus \text{address_block}(\text{base}, \text{size})) \cup (\text{pm_phy}'\text{rw_addr} \setminus \text{address_block}(\text{base}, \text{size})) \setminus \text{rnd_dev_read_side_effect})) \\
 & \text{rnd_dev_read_side_effect} \\
 & (q') \\
 \hline
 \{1\} & \text{state}(q'(s'))' \text{state} = s' \text{state}
 \end{array}$$

Expanding the definition of drse_super_transformers,

Repeatedly Skolemizing and flattening,

Replacing using formula -4,

Keeping (-2 1) and hiding *,

Rewriting using has_next_state_expr_2_super, matching in *,

Rewriting using state_expr_2_super, matching in *,

Expanding the definition of rnd_dev_read_side_effect,

Expanding the definition of unaligned_access,

Case splitting on $\text{null?}(a!1) \text{ OR } \text{disjoint?}(\text{address_block}(a!1, \text{length}(a!1)), \text{address_block}(\text{base}, \text{size}[\text{Physical_memory}, \text{pm_phy}])) \text{ OR } (\text{NOT } \text{cp}!1 \text{ AND } \text{length}(a!1) = \text{size}(\text{uidt}(\text{dt_uint})) \text{ AND } (a!1 = \text{rnd_seed} \text{ OR } a!1 = \text{mem_cnt} \text{ OR } a!1 = \text{rnd_val}))$,

we get 2 subgoals:

random_side_effects_unchanged_invariant.2.1.1:

```

{-1} null?(bl') ∨
      disjoint?(address_block(a', length(bl')),
                address_block(base, size[Physical_memory, pm_phy]))
      ∨
      (¬ cp' ∧
       length(bl') = size(uidt(dt_uint)) ∧
       (a' = rnd_seed ∨ a' = mem_cnt ∨ a' = rnd_val))
{-2} has_next_state((IF null?(bl') ∨
                    disjoint?(address_block(a', length(bl')),
                                address_block(base, size))
                    ∨
                    (¬ cp' ∧
                     length(bl') = size(uidt(dt_uint)) ∧
                     (a' = rnd_seed ∨ a' = mem_cnt ∨ a' = rnd_val))
                    THEN ok_result(bl')
                    ELSE fatal_result
                    ENDIF
                    ##
                    (λ (bl: list[Byte]):
                     access_count(a', bl, cp') ##
                     (λ (bl: list[Byte]):
                      read_rnd_val(a', bl, cp') ##
                      (λ (bl: list[Byte]):
                       read_mem_cnt(a', bl, cp') ##
                       (λ (bl: list[Byte]): read_seed(a', bl, cp'))))))))
                    (s'))
{1} state((IF null?(bl') ∨
           disjoint?(address_block(a', length(bl')), address_block(base, size)) ∨
           (¬ cp' ∧
            length(bl') = size(uidt(dt_uint)) ∧
            (a' = rnd_seed ∨ a' = mem_cnt ∨ a' = rnd_val))
           THEN ok_result(bl')
           ELSE fatal_result
           ENDIF
           ##
           (λ (bl: list[Byte]):
            access_count(a', bl, cp') ##
            (λ (bl: list[Byte]):
             read_rnd_val(a', bl, cp') ##
             (λ (bl: list[Byte]):
              read_mem_cnt(a', bl, cp') ##
              (λ (bl: list[Byte]): read_seed(a', bl, cp'))))))))
           (s'))'state
      = s' 'state

```

Replacing using formula -1,
Expanding the definition of access_count,
Expanding the definition of read_rnd_val,
Expanding the definition of read_mem_cnt,
Expanding the definition of read_seed,
Expanding the definition of ##,
Expanding the definition of ok_result,

Expanding the definition of `has_next_state`,
 Lifting IF-conditions to the top level,
 Expanding the definition of `increase_access_count`,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 This completes the proof of `random_side_effects_unchanged_invariant.2.1.1`.
`random_side_effects_unchanged_invariant.2.1.2`:

{-1}	<pre> has_next_state((IF null?(bl') ∨ disjoint?(address_block(a', length(bl')), address_block(base, size)) ∨ (¬ cp' ∧ length(bl') = size(uidt(dt_uint)) ∧ (a' = rnd_seed ∨ a' = mem_cnt ∨ a' = rnd_val)) THEN ok_result(bl') ELSE fatal_result ENDIF ## (λ (bl: list[Byte]): access_count(a', bl, cp') ## (λ (bl: list[Byte]): read_rnd_val(a', bl, cp') ## (λ (bl: list[Byte]): read_mem_cnt(a', bl, cp') ## (λ (bl: list[Byte]): read_seed(a', bl, cp')))))) (s')) </pre>
{1}	<pre> null?(bl') ∨ disjoint?(address_block(a', length(bl')), address_block(base, size[Physical_memory, pm_phy])) ∨ (¬ cp' ∧ length(bl') = size(uidt(dt_uint)) ∧ (a' = rnd_seed ∨ a' = mem_cnt ∨ a' = rnd_val)) </pre>
{2}	<pre> state((IF null?(bl') ∨ disjoint?(address_block(a', length(bl')), address_block(base, size)) ∨ (¬ cp' ∧ length(bl') = size(uidt(dt_uint)) ∧ (a' = rnd_seed ∨ a' = mem_cnt ∨ a' = rnd_val)) THEN ok_result(bl') ELSE fatal_result ENDIF ## (λ (bl: list[Byte]): access_count(a', bl, cp') ## (λ (bl: list[Byte]): read_rnd_val(a', bl, cp') ## (λ (bl: list[Byte]): read_mem_cnt(a', bl, cp') ## (λ (bl: list[Byte]): read_seed(a', bl, cp')))))) (s'))'state = s' state </pre>

Replacing using formula 1,
 Expanding the definition of `fatal_result`,

C Proof scripts

Expanding the definition of ##,

Expanding the definition of has_next_state,

which is trivially true.

This completes the proof of `random_side_effects_unchanged_invariant.2.1.2`.

`random_side_effects_unchanged_invariant.2.2`:

$$\frac{\begin{array}{l} \{-1\} \text{ has_next_state}(q'(s')) \\ \{-2\} \text{ dwse_super_transformers}((\text{pm_phy}'\text{rw_addr} \setminus \text{address_block}(\text{base}, \text{size})), \\ \qquad \qquad \qquad \text{rnd_dev_write_side_effect}) \\ \qquad \qquad \qquad (q') \end{array}}{\{1\} \text{ state}(q'(s'))' \text{state} = s' \text{state}}$$

Expanding the definition of `dwse_super_transformers`,

Repeatedly Skolemizing and flattening,

Replacing using formula -4,

Keeping (-2 1) and hiding *,

Rewriting using `has_next_state_expr_2_super`, matching in *,

Rewriting using `state_expr_2_super`, matching in *,

Expanding the definition of `rnd_dev_write_side_effect`,

Expanding the definition of `unaligned_access`,

Case splitting on `null?(bl!1) OR disjoint?(address_block(a!1, length(bl!1)), address_block(base, size[Physical_memory, pm_phy])) OR (NOT cp!1 AND length(bl!1) = size(uidt(dt_uint)) AND (a!1 = rnd_seed OR a!1 = mem_cnt OR a!1 = rnd_val))`,

we get 2 subgoals:

random_side_effects_unchanged_invariant.2.2.1:

```

{-1} null?(bl') ∨
      disjoint?(address_block(a', length(bl')),
                address_block(base, size[Physical_memory, pm_phy]))
      ∨
      (¬ cp' ∧
       length(bl') = size(uidt(dt_uint)) ∧
       (a' = rnd_seed ∨ a' = mem_cnt ∨ a' = rnd_val))
{-2} has_next_state((IF null?(bl') ∨
                    disjoint?(address_block(a', length(bl')),
                              address_block(base, size))
                    ∨
                    (¬ cp' ∧
                     length(bl') = size(uidt(dt_uint)) ∧
                     (a' = rnd_seed ∨ a' = mem_cnt ∨ a' = rnd_val))
                    THEN ok_result(bl')
                    ELSE fatal_result
                    ENDIF
                    ##
                    (λ (bl: list[Byte]):
                     access_count(a', bl, cp') ##
                     (λ (bl: list[Byte]):
                      write_rnd_dev(a', bl, cp') ##
                      (λ (bl: list[Byte]): reset_count(a', bl, cp')))))
                    (s'))


---


{1} state((IF null?(bl') ∨
           disjoint?(address_block(a', length(bl')), address_block(base, size)) ∨
           (¬ cp' ∧
            length(bl') = size(uidt(dt_uint)) ∧
            (a' = rnd_seed ∨ a' = mem_cnt ∨ a' = rnd_val))
           THEN ok_result(bl')
           ELSE fatal_result
           ENDIF
           ##
           (λ (bl: list[Byte]):
            access_count(a', bl, cp') ##
            (λ (bl: list[Byte]):
             write_rnd_dev(a', bl, cp') ##
             (λ (bl: list[Byte]): reset_count(a', bl, cp')))))
           (s'))'state
= s' state

```

Replacing using formula -1,
Expanding the definition of access_count,
Expanding the definition of write_rnd_dev,
Expanding the definition of increase_access_count,
Expanding the definition of reset_count,
Expanding the definition of ok_result,
Expanding the definition of ##,
Expanding the definition of clear_counter,
Expanding the definition of fatal_result,
Expanding the definition of has_next_state,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `random_side_effects_unchanged_invariant.2.2.1`.

`random_side_effects_unchanged_invariant.2.2.2`:

```

{-1} has_next_state((IF null?(bl') ∨
                    disjoint?(address_block(a', length(bl')),
                              address_block(base, size))
                    ∨
                    (¬ cp' ∧
                     length(bl') = size(uidt(dt_uint)) ∧
                     (a' = rnd_seed ∨ a' = mem_cnt ∨ a' = rnd_val))
                    THEN ok_result(bl')
                    ELSE fatal_result
                    ENDIF
                    ##
                    (λ (bl: list[Byte]):
                     access_count(a', bl, cp') ##
                     (λ (bl: list[Byte]):
                      write_rnd_dev(a', bl, cp') ##
                      (λ (bl: list[Byte]): reset_count(a', bl, cp')))))
                    (s'))

{1} null?(bl') ∨
    disjoint?(address_block(a', length(bl')),
              address_block(base, size[Physical_memory, pm_phy]))
    ∨
    (¬ cp' ∧
     length(bl') = size(uidt(dt_uint)) ∧
     (a' = rnd_seed ∨ a' = mem_cnt ∨ a' = rnd_val))
{2} state((IF null?(bl') ∨
          disjoint?(address_block(a', length(bl')), address_block(base, size)) ∨
          (¬ cp' ∧
           length(bl') = size(uidt(dt_uint)) ∧
           (a' = rnd_seed ∨ a' = mem_cnt ∨ a' = rnd_val))
          THEN ok_result(bl')
          ELSE fatal_result
          ENDIF
          ##
          (λ (bl: list[Byte]):
           access_count(a', bl, cp') ##
           (λ (bl: list[Byte]):
            write_rnd_dev(a', bl, cp') ##
            (λ (bl: list[Byte]): reset_count(a', bl, cp')))))
          (s'))'state
    = s''state

```

Replacing using formula 1,

Expanding the definition of `fatal_result`,

Expanding the definition of `##`,

Expanding the definition of `has_next_state`,

which is trivially true.

This completes the proof of `random_side_effects_unchanged_invariant.2.2.2`.

Q.E.D.

C.23.5

Challenge_Random_Device.random_side_effects_transformers_ok

Terse proof for random_side_effects_transformers_ok.

```
random_side_effects_transformers_ok:
```

{1}	transformers_ok?(pm' states, <div style="text-align: right; margin-right: 20px;">(drse_super_transformers((pm'ro_addr U pm'rw_addr), rnd_dev_read_side_effect) U dwse_s</div>
-----	--

Installing automatic rewrites from: ok_expr_2_super has_next_state_expr_2_super fatal_result state_expr_2_super ok_result_state ok_result_ok expr_2_super_ok_result expr_2_super_fatal_result has_next_state_ok_result has_next_state_fatal_result ok_result_data in_blessed_memory_disjoint_device

Rewriting using transformers_ok_union_transformers, matching in *,

we get 2 subgoals:

```
random_side_effects_transformers_ok.1:
```

{1}	transformers_ok?(pm' states, <div style="text-align: right; margin-right: 20px;">drse_super_transformers((pm'ro_addr U pm'rw_addr), rnd_dev_read_side_effect)) </div>
{2}	transformers_ok?(pm' states, <div style="text-align: right; margin-right: 20px;">(drse_super_transformers((pm'ro_addr U pm'rw_addr), rnd_dev_read_side_effect) U dwse_s</div>

Rewriting using transformers_ok_all_transformers, matching in *,

Expanding the definition of drse_super_transformers,

Repeatedly Skolemizing and flattening,

Replacing using formula -3,

Keeping (-2 1) and hiding *,

Expanding the definition of rnd_dev_read_side_effect,

Using lemma in_blessed_memory_disjoint_device,

Using lemma in_blessed_memory_not_device_address,

Replacing using formula -3,

Case splitting on transformers_ok?(pm' states, singleton(expr_2_super(ok_result [Random_device_memory, list[Byte]] (bl!1))))),

we get 2 subgoals:

random_side_effects_transformers_ok.1.1:

```

{-1} transformers_ok?(pm'states,
                      singleton(expr_2_super(ok_result [Random_device_memory, list[Byte]]
                                           (bl'))))
{-2} cons?(bl')  $\supset$   $a' \neq \text{rnd\_seed} \wedge a' \neq \text{mem\_cnt} \wedge a' \neq \text{rnd\_val}$ 
{-3} disjoint?(address_block(a', length(bl')), address_block(base, size))
{-4} (address_block(a', length(bl'))  $\subseteq$  (pm'ro_addr  $\cup$  pm'rw_addr))
-----
{1} transformers_ok?(pm'states,
                    singleton(expr_2_super((unaligned_access(a', bl', cp') ##
                                             ( $\lambda$  (bl: list[Byte]):
                                              access_count(a', bl, cp') ##
                                              ( $\lambda$  (bl: list[Byte]):
                                               read_rnd_val(a', bl, cp') ##
                                               ( $\lambda$  (bl: list[Byte]):
                                                read_mem_cnt(a', bl, cp') ##
                                                ##
                                                ( $\lambda$ 
                                                 (bl: list[Byte]):
                                                  read_seed
                                                  (a', bl, cp'))))))))))))

```

Case splitting on transformer_invariant?(pm'states, singleton(expr_2_super (ok_result [Random_device_memory, list[Byte]] (bl!1))),

we get 2 subgoals:

random_side_effects_transformers_ok.1.1.1:

```

{-1} transformer_invariant?(pm'states,
                            singleton(expr_2_super(ok_result [Random_device_memory, list[Byte]]
                                                     (bl'))))
{-2} transformers_ok?(pm'states,
                    singleton(expr_2_super(ok_result [Random_device_memory, list[Byte]]
                                           (bl'))))
{-3} cons?(bl')  $\supset$   $a' \neq \text{rnd\_seed} \wedge a' \neq \text{mem\_cnt} \wedge a' \neq \text{rnd\_val}$ 
{-4} disjoint?(address_block(a', length(bl')), address_block(base, size))
{-5} (address_block(a', length(bl'))  $\subseteq$  (pm'ro_addr  $\cup$  pm'rw_addr))
-----
{1} transformers_ok?(pm'states,
                    singleton(expr_2_super((unaligned_access(a', bl', cp') ##
                                             ( $\lambda$  (bl: list[Byte]):
                                              access_count(a', bl, cp') ##
                                              ( $\lambda$  (bl: list[Byte]):
                                               read_rnd_val(a', bl, cp') ##
                                               ( $\lambda$  (bl: list[Byte]):
                                                read_mem_cnt(a', bl, cp') ##
                                                ##
                                                ( $\lambda$ 
                                                 (bl: list[Byte]):
                                                  read_seed
                                                  (a', bl, cp'))))))))))))

```

Rewriting using fexpr_composition_transformers_ok, matching in * where P gets LAMBDA (bl: list[Byte]): bl = bl!1,

we get 4 subgoals:

C.23 Proofs for Challenge_Random_Device (random_device.pvs)

random_side_effects_transformers_ok.1.1.1.1:

<pre>{-1} transformer_invariant?(pm' states, singleton(expr_2_super(ok_result [Random_device_memory, list[Byte]] (bl')))) {-2} transformers_ok?(pm' states, singleton(expr_2_super(ok_result [Random_device_memory, list[Byte]] (bl')))) {-3} cons?(bl') \supset a' \neq rnd_seed \wedge a' \neq mem_cnt \wedge a' \neq rnd_val {-4} disjoint?(address_block(a', length(bl')), address_block(base, size)) {-5} (address_block(a', length(bl')) \subseteq (pm'ro_addr \cup pm'rw_addr))</pre>	<pre> {1} transformers_ok?(pm' states, singleton(expr_2_super(unaligned_access(a', bl', cp')))) {2} transformers_ok?(pm' states, singleton(expr_2_super((unaligned_access(a', bl', cp') ## (λ (bl: list[Byte]): access_count(a', bl, cp') ## (λ (bl: list[Byte]): read_rnd_val(a', bl, cp') ## (λ (bl: list[Byte]): read_mem_cnt(a', bl, cp') ## (λ (bl: list[Byte]): read_seed (a', bl, cp'))))))))))</pre>
--	--

Expanding the definition of unaligned_access,
which is trivially true.

This completes the proof of random_side_effects_transformers_ok.1.1.1.1.

random_side_effects_transformers_ok.1.1.1.2:

<pre>{-1} transformer_invariant?(pm' states, singleton(expr_2_super(ok_result [Random_device_memory, list[Byte]] (bl')))) {-2} transformers_ok?(pm' states, singleton(expr_2_super(ok_result [Random_device_memory, list[Byte]] (bl')))) {-3} cons?(bl') \supset a' \neq rnd_seed \wedge a' \neq mem_cnt \wedge a' \neq rnd_val {-4} disjoint?(address_block(a', length(bl')), address_block(base, size)) {-5} (address_block(a', length(bl')) \subseteq (pm'ro_addr \cup pm'rw_addr))</pre>	<pre> {1} transformer_invariant?(pm' states, singleton(expr_2_super(unaligned_access(a', bl', cp')))) {2} transformers_ok?(pm' states, singleton(expr_2_super((unaligned_access(a', bl', cp') ## (λ (bl: list[Byte]): access_count(a', bl, cp') ## (λ (bl: list[Byte]): read_rnd_val(a', bl, cp') ## (λ (bl: list[Byte]): read_mem_cnt(a', bl, cp') ## (λ (bl: list[Byte]): read_seed (a', bl, cp'))))))))))</pre>
--	---

C Proof scripts

Expanding the definition of `unaligned_access`,

which is trivially true.

This completes the proof of `random_side_effects_transformers_ok.1.1.1.2`.

`random_side_effects_transformers_ok.1.1.1.3`:

<pre> {-1} transformer_invariant?(pm' states, singleton(expr_2_super(ok_result [Random_device_memory, list[Byte]] (bl')))) {-2} transformers_ok?(pm' states, singleton(expr_2_super(ok_result [Random_device_memory, list[Byte]] (bl')))) {-3} cons?(bl') \supset a' \neq rnd_seed \wedge a' \neq mem_cnt \wedge a' \neq rnd_val {-4} disjoint?(address_block(a', length(bl')), address_block(base, size)) {-5} (address_block(a', length(bl')) \subseteq (pm'ro_addr \cup pm'rw_addr)) </pre>	<hr style="border: 0.5px solid black;"/> <pre> {1} \forall (s: (pm' states)): OK?(unaligned_access(a', bl', cp')(s) \supset data(unaligned_access(a', bl', cp')(s)) = bl') {2} transformers_ok?(pm' states, singleton(expr_2_super((unaligned_access(a', bl', cp') ## (λ (bl: list[Byte]): access_count(a', bl, cp') ## (λ (bl: list[Byte]): read_rnd_val(a', bl, cp') ## (λ (bl: list[Byte]): read_mem_cnt(a', bl, cp') ## (λ (bl: list[Byte]): read_seed (a', bl, cp')))))))))))) </pre>
--	---

Expanding the definition of `unaligned_access`,

Repeatedly Skolemizing and flattening,

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `random_side_effects_transformers_ok.1.1.1.3`.

random_side_effects_transformers_ok.1.1.1.4:

{-1}	transformer_invariant?(pm' states, singleton(expr_2_super(ok_result [Random_device_memory, list[Byte]] (bl'))))
{-2}	transformers_ok?(pm' states, singleton(expr_2_super(ok_result [Random_device_memory, list[Byte]] (bl'))))
{-3}	cons?(bl') $\supset a' \neq \text{rnd_seed} \wedge a' \neq \text{mem_cnt} \wedge a' \neq \text{rnd_val}$
{-4}	disjoint?(address_block(a', length(bl')), address_block(base, size))
{-5}	(address_block(a', length(bl')) \subseteq (pm'ro_addr \cup pm'rw_addr))
{1}	$\forall (d: (\lambda (bl: \text{list}[\text{Byte}]): bl = bl')):$ transformers_ok?(pm' states, singleton(expr_2_super(access_count(a', d, cp') ## ($\lambda (bl: \text{list}[\text{Byte}]):$ read_rnd_val(a', bl, cp') ## ($\lambda (bl: \text{list}[\text{Byte}]):$ read_mem_cnt(a', bl, cp') ## ($\lambda (bl: \text{list}[\text{Byte}]):$ read_seed(a', bl, cp'))))))))
{2}	transformers_ok?(pm' states, singleton(expr_2_super((unaligned_access(a', bl', cp') ## ($\lambda (bl: \text{list}[\text{Byte}]):$ access_count(a', bl, cp') ## ($\lambda (bl: \text{list}[\text{Byte}]):$ read_rnd_val(a', bl, cp') ## ($\lambda (bl: \text{list}[\text{Byte}]):$ read_mem_cnt(a', bl, cp') ## (λ (bl: list[Byte]): read_seed (a', bl, cp'))))))))))))

Repeatedly Skolemizing and flattening,

Hiding formulas: (-1 2),

Rewriting using fexpr_composition_transformers_ok, matching in * where P gets LAMBDA (bl: list[Byte]): bl = bl!1,

we get 4 subgoals:

random_side_effects_transformers_ok.1.1.1.4.1:

{-1}	$d' = \text{bl}'$
{-2}	$\text{transformer_invariant?}(\text{pm}'\text{states}, \text{singleton}(\text{expr_2_super}(\text{ok_result}[\text{Random_device_memory}, \text{list}[\text{Byte}] (\text{bl}')])))$
{-3}	$\text{transformers_ok?}(\text{pm}'\text{states}, \text{singleton}(\text{expr_2_super}(\text{ok_result}[\text{Random_device_memory}, \text{list}[\text{Byte}] (\text{bl}')])))$
{-4}	$\text{cons?}(\text{bl}') \supset a' \neq \text{rnd_seed} \wedge a' \neq \text{mem_cnt} \wedge a' \neq \text{rnd_val}$
{-5}	$\text{disjoint?}(\text{address_block}(a', \text{length}(\text{bl}')), \text{address_block}(\text{base}, \text{size}))$
{-6}	$(\text{address_block}(a', \text{length}(\text{bl}')) \subseteq (\text{pm}'\text{ro_addr} \cup \text{pm}'\text{rw_addr}))$
{1}	$\text{transformers_ok?}(\text{pm}'\text{states}, \text{singleton}(\text{expr_2_super}(\text{access_count}(a', d', \text{cp}'))))$
{2}	$\text{transformers_ok?}(\text{pm}'\text{states}, \text{singleton}(\text{expr_2_super}(\text{access_count}(a', d', \text{cp}') \#\# (\lambda (\text{bl}: \text{list}[\text{Byte}]): \text{read_rnd_val}(a', \text{bl}, \text{cp}') \#\# (\lambda (\text{bl}: \text{list}[\text{Byte}]): \text{read_mem_cnt}(a', \text{bl}, \text{cp}') \#\# (\lambda (\text{bl}: \text{list}[\text{Byte}]): \text{read_seed}(a', \text{bl}, \text{cp}') \#\#))))))$

Expanding the definition of access_count,

Keeping (1) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of random_side_effects_transformers_ok.1.1.1.4.1.

random_side_effects_transformers_ok.1.1.1.4.2:

{-1}	$d' = \text{bl}'$
{-2}	$\text{transformer_invariant?}(\text{pm}'\text{states}, \text{singleton}(\text{expr_2_super}(\text{ok_result}[\text{Random_device_memory}, \text{list}[\text{Byte}] (\text{bl}')])))$
{-3}	$\text{transformers_ok?}(\text{pm}'\text{states}, \text{singleton}(\text{expr_2_super}(\text{ok_result}[\text{Random_device_memory}, \text{list}[\text{Byte}] (\text{bl}')])))$
{-4}	$\text{cons?}(\text{bl}') \supset a' \neq \text{rnd_seed} \wedge a' \neq \text{mem_cnt} \wedge a' \neq \text{rnd_val}$
{-5}	$\text{disjoint?}(\text{address_block}(a', \text{length}(\text{bl}')), \text{address_block}(\text{base}, \text{size}))$
{-6}	$(\text{address_block}(a', \text{length}(\text{bl}')) \subseteq (\text{pm}'\text{ro_addr} \cup \text{pm}'\text{rw_addr}))$
{1}	$\text{transformer_invariant?}(\text{pm}'\text{states}, \text{singleton}(\text{expr_2_super}(\text{access_count}(a', d', \text{cp}'))))$
{2}	$\text{transformers_ok?}(\text{pm}'\text{states}, \text{singleton}(\text{expr_2_super}(\text{access_count}(a', d', \text{cp}') \#\# (\lambda (\text{bl}: \text{list}[\text{Byte}]): \text{read_rnd_val}(a', \text{bl}, \text{cp}') \#\# (\lambda (\text{bl}: \text{list}[\text{Byte}]): \text{read_mem_cnt}(a', \text{bl}, \text{cp}') \#\# (\lambda (\text{bl}: \text{list}[\text{Byte}]): \text{read_seed}(a', \text{bl}, \text{cp}') \#\#))))))$

Keeping (1) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of random_side_effects_transformers_ok.1.1.1.4.2.

random_side_effects_transformers_ok.1.1.1.4.3:

<pre> {-1} d' = bl' {-2} transformer_invariant?(pm' states, singleton(expr_2_super(ok_result [Random_device_memory, list[Byte]] (bl')))) {-3} transformers_ok?(pm' states, singleton(expr_2_super(ok_result [Random_device_memory, list[Byte]] (bl')))) {-4} cons?(bl') \supset a' \neq rnd_seed \wedge a' \neq mem_cnt \wedge a' \neq rnd_val {-5} disjoint?(address_block(a', length(bl')), address_block(base, size)) {-6} (address_block(a', length(bl')) \subseteq (pm'ro_addr \cup pm'rw_addr)) </pre>	<pre> {1} \forall (s: (pm' states)): OK?(access_count(a', d', cp')(s) \supset data_access_count(a', d', cp')(s) = bl' {2} transformers_ok?(pm' states, singleton(expr_2_super(access_count(a', d', cp') ## (λ (bl: list[Byte]): read_rnd_val(a', bl, cp') ## (λ (bl: list[Byte]): read_mem_cnt(a', bl, cp') ## (λ (bl: list[Byte]): read_seed(a', bl, cp')))))))) </pre>
--	---

Repeatedly Skolemizing and flattening,
Expanding the definition of access_count,
which is trivially true.

This completes the proof of random_side_effects_transformers_ok.1.1.1.4.3.

random_side_effects_transformers_ok.1.1.1.4.4:

<pre> {-1} d' = bl' {-2} transformer_invariant?(pm' states, singleton(expr_2_super(ok_result [Random_device_memory, list[Byte]] (bl')))) {-3} transformers_ok?(pm' states, singleton(expr_2_super(ok_result [Random_device_memory, list[Byte]] (bl')))) {-4} cons?(bl') \supset a' \neq rnd_seed \wedge a' \neq mem_cnt \wedge a' \neq rnd_val {-5} disjoint?(address_block(a', length(bl')), address_block(base, size)) {-6} (address_block(a', length(bl')) \subseteq (pm'ro_addr \cup pm'rw_addr)) </pre>	<pre> {1} \forall (d: (λ (bl: list[Byte]): bl = bl')): transformers_ok?(pm' states, singleton(expr_2_super(read_rnd_val(a', d, cp') ## (λ (bl: list[Byte]): read_mem_cnt(a', bl, cp') ## (λ (bl: list[Byte]): read_seed(a', bl, cp')))))) {2} transformers_ok?(pm' states, singleton(expr_2_super(access_count(a', d', cp') ## (λ (bl: list[Byte]): read_rnd_val(a', bl, cp') ## (λ (bl: list[Byte]): read_mem_cnt(a', bl, cp') ## (λ (bl: list[Byte]): read_seed(a', bl, cp')))))))) </pre>
--	---

C Proof scripts

Repeatedly Skolemizing and flattening,

Hiding formulas: (-1 2),

Rewriting using `fexpr_composition_transformers_ok`, matching in `*` where `P` gets `LAMBDA (bl: list[Byte])`: `bl = bl!1`,

we get 4 subgoals:

`random_side_effects_transformers_ok.1.1.1.4.4.1:`

{-1}	$d'' = bl'$
{-2}	$d' = bl'$
{-3}	$\text{transformer_invariant?}(\text{pm}'\text{states}, \text{singleton}(\text{expr_2_super}(\text{ok_result}[\text{Random_device_memory}, \text{list}[\text{Byte}]](bl'))))$
{-4}	$\text{transformers_ok?}(\text{pm}'\text{states}, \text{singleton}(\text{expr_2_super}(\text{ok_result}[\text{Random_device_memory}, \text{list}[\text{Byte}]](bl'))))$
{-5}	$\text{cons?}(bl') \supset a' \neq \text{rnd_seed} \wedge a' \neq \text{mem_cnt} \wedge a' \neq \text{rnd_val}$
{-6}	$\text{disjoint?}(\text{address_block}(a', \text{length}(bl')), \text{address_block}(\text{base}, \text{size}))$
{-7}	$(\text{address_block}(a', \text{length}(bl')) \subseteq (\text{pm}'\text{ro_addr} \cup \text{pm}'\text{rw_addr}))$
{1}	$\text{transformers_ok?}(\text{pm}'\text{states}, \text{singleton}(\text{expr_2_super}(\text{read_rnd_val}(a', d'', cp'))))$
{2}	$\text{transformers_ok?}(\text{pm}'\text{states}, \text{singleton}(\text{expr_2_super}(\text{read_rnd_val}(a', d'', cp') \text{ ## } (\lambda (bl: \text{list}[\text{Byte}]): \text{read_mem_cnt}(a', bl, cp') \text{ ## } (\lambda (bl: \text{list}[\text{Byte}]): \text{read_seed}(a', bl, cp'))))))$

Hiding formulas: 2,

Expanding the definition of `read_rnd_val`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `random_side_effects_transformers_ok.1.1.1.4.4.1`.

`random_side_effects_transformers_ok.1.1.1.4.4.2:`

{-1}	$d'' = bl'$
{-2}	$d' = bl'$
{-3}	$\text{transformer_invariant?}(\text{pm}'\text{states}, \text{singleton}(\text{expr_2_super}(\text{ok_result}[\text{Random_device_memory}, \text{list}[\text{Byte}]](bl'))))$
{-4}	$\text{transformers_ok?}(\text{pm}'\text{states}, \text{singleton}(\text{expr_2_super}(\text{ok_result}[\text{Random_device_memory}, \text{list}[\text{Byte}]](bl'))))$
{-5}	$\text{cons?}(bl') \supset a' \neq \text{rnd_seed} \wedge a' \neq \text{mem_cnt} \wedge a' \neq \text{rnd_val}$
{-6}	$\text{disjoint?}(\text{address_block}(a', \text{length}(bl')), \text{address_block}(\text{base}, \text{size}))$
{-7}	$(\text{address_block}(a', \text{length}(bl')) \subseteq (\text{pm}'\text{ro_addr} \cup \text{pm}'\text{rw_addr}))$
{1}	$\text{transformer_invariant?}(\text{pm}'\text{states}, \text{singleton}(\text{expr_2_super}(\text{read_rnd_val}(a', d'', cp'))))$
{2}	$\text{transformers_ok?}(\text{pm}'\text{states}, \text{singleton}(\text{expr_2_super}(\text{read_rnd_val}(a', d'', cp') \text{ ## } (\lambda (bl: \text{list}[\text{Byte}]): \text{read_mem_cnt}(a', bl, cp') \text{ ## } (\lambda (bl: \text{list}[\text{Byte}]): \text{read_seed}(a', bl, cp'))))))$

Expanding the definition of `read_rnd_val`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `random_side_effects_transformers_ok.1.1.1.4.4.2`.

random_side_effects_transformers_ok.1.1.1.4.4.3:

{-1}	$d'' = bl'$
{-2}	$d' = bl'$
{-3}	transformer_invariant?(pm' states, singleton(expr_2_super(ok_result [Random_device_memory, list[Byte]] (bl'))))
{-4}	transformers_ok?(pm' states, singleton(expr_2_super(ok_result [Random_device_memory, list[Byte]] (bl'))))
{-5}	cons?(bl') \supset $a' \neq \text{rnd_seed} \wedge a' \neq \text{mem_cnt} \wedge a' \neq \text{rnd_val}$
{-6}	disjoint?(address_block(a' , length(bl')), address_block(base, size))
{-7}	(address_block(a' , length(bl')) \subseteq (pm'ro_addr \cup pm'rw_addr))
{1}	$\forall (s: (\text{pm}'\text{states})):$ OK?(read_rnd_val(a' , d'' , cp')(s) \supset data_block(read_rnd_val(a' , d'' , cp')(s)) = bl'
{2}	transformers_ok?(pm' states, singleton(expr_2_super(read_rnd_val(a' , d'' , cp') ## (λ (bl: list[Byte]): read_mem_cnt(a' , bl, cp') ## (λ (bl: list[Byte]): read_seed(a' , bl, cp'))))))

Repeatedly Skolemizing and flattening,

Expanding the definition of read_rnd_val,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of random_side_effects_transformers_ok.1.1.1.4.4.3.

random_side_effects_transformers_ok.1.1.1.4.4.4:

{-1}	$d'' = bl'$
{-2}	$d' = bl'$
{-3}	transformer_invariant?(pm' states, singleton(expr_2_super(ok_result [Random_device_memory, list[Byte]] (bl'))))
{-4}	transformers_ok?(pm' states, singleton(expr_2_super(ok_result [Random_device_memory, list[Byte]] (bl'))))
{-5}	cons?(bl') \supset $a' \neq \text{rnd_seed} \wedge a' \neq \text{mem_cnt} \wedge a' \neq \text{rnd_val}$
{-6}	disjoint?(address_block(a' , length(bl')), address_block(base, size))
{-7}	(address_block(a' , length(bl')) \subseteq (pm'ro_addr \cup pm'rw_addr))
{1}	$\forall (d: (\lambda$ (bl: list[Byte]): bl = bl')): transformers_ok?(pm' states, singleton(expr_2_super(read_mem_cnt(a' , d , cp') ## (λ (bl: list[Byte]): read_seed(a' , bl, cp'))))))
{2}	transformers_ok?(pm' states, singleton(expr_2_super(read_rnd_val(a' , d'' , cp') ## (λ (bl: list[Byte]): read_mem_cnt(a' , bl, cp') ## (λ (bl: list[Byte]): read_seed(a' , bl, cp'))))))

Repeatedly Skolemizing and flattening,

Hiding formulas: (-1 2),

Rewriting using fexpr_composition_transformers_ok, matching in * where P gets LAMBDA (bl:

C Proof scripts

list[Byte]): bl = bl!1,

we get 4 subgoals:

random_side_effects_transformers_ok.1.1.1.4.4.4.1:

{-1}	$d''' = bl'$
{-2}	$d'' = bl'$
{-3}	$d' = bl'$
{-4}	$transformer_invariant?(pm\ 'states,$ $\quad singleton(expr_2_super(ok_result[Random_device_memory, list[Byte]]$ $\quad\quad\quad (bl'))))$
{-5}	$transformers_ok?(pm\ 'states,$ $\quad singleton(expr_2_super(ok_result[Random_device_memory, list[Byte]]$ $\quad\quad\quad (bl'))))$
{-6}	$cons?(bl') \supset a' \neq rnd_seed \wedge a' \neq mem_cnt \wedge a' \neq rnd_val$
{-7}	$disjoint?(address_block(a', length(bl')), address_block(base, size))$
{-8}	$(address_block(a', length(bl')) \subseteq (pm\ 'ro_addr \cup pm\ 'rw_addr))$
{1}	$transformers_ok?(pm\ 'states, singleton(expr_2_super(read_mem_cnt(a', d''', cp'))))$
{2}	$transformers_ok?(pm\ 'states,$ $\quad singleton(expr_2_super(read_mem_cnt(a', d''', cp') \#\#$ $\quad\quad\quad (\lambda (bl: list[Byte]):$ $\quad\quad\quad\quad read_seed(a', bl, cp'))))$

Expanding the definition of read_mem_cnt,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of random_side_effects_transformers_ok.1.1.1.4.4.4.1.

random_side_effects_transformers_ok.1.1.1.4.4.4.2:

{-1}	$d''' = bl'$
{-2}	$d'' = bl'$
{-3}	$d' = bl'$
{-4}	$transformer_invariant?(pm\ 'states,$ $\quad singleton(expr_2_super(ok_result[Random_device_memory, list[Byte]]$ $\quad\quad\quad (bl'))))$
{-5}	$transformers_ok?(pm\ 'states,$ $\quad singleton(expr_2_super(ok_result[Random_device_memory, list[Byte]]$ $\quad\quad\quad (bl'))))$
{-6}	$cons?(bl') \supset a' \neq rnd_seed \wedge a' \neq mem_cnt \wedge a' \neq rnd_val$
{-7}	$disjoint?(address_block(a', length(bl')), address_block(base, size))$
{-8}	$(address_block(a', length(bl')) \subseteq (pm\ 'ro_addr \cup pm\ 'rw_addr))$
{1}	$transformer_invariant?(pm\ 'states, singleton(expr_2_super(read_mem_cnt(a', d''', cp'))))$
{2}	$transformers_ok?(pm\ 'states,$ $\quad singleton(expr_2_super(read_mem_cnt(a', d''', cp') \#\#$ $\quad\quad\quad (\lambda (bl: list[Byte]):$ $\quad\quad\quad\quad read_seed(a', bl, cp'))))$

Expanding the definition of read_mem_cnt,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of random_side_effects_transformers_ok.1.1.1.4.4.4.2.

random_side_effects_transformers_ok.1.1.1.4.4.4.3:

<pre> {-1} d''' = bl' {-2} d'' = bl' {-3} d' = bl' {-4} transformer_invariant?(pm' states, singleton(expr_2_super(ok_result [Random_device_memory, list[Byte]] (bl')))) {-5} transformers_ok?(pm' states, singleton(expr_2_super(ok_result [Random_device_memory, list[Byte]] (bl')))) {-6} cons?(bl') \supset a' \neq rnd_seed \wedge a' \neq mem_cnt \wedge a' \neq rnd_val {-7} disjoint?(address_block(a', length(bl')), address_block(base, size)) {-8} (address_block(a', length(bl')) \subseteq (pm'ro_addr \cup pm'rw_addr)) </pre>	<pre> {1} \forall (s: (pm' states)): OK?(read_mem_cnt(a', d''', cp')(s) \supset data(read_mem_cnt(a', d''', cp')(s)) = bl') {2} transformers_ok?(pm' states, singleton(expr_2_super(read_mem_cnt(a', d''', cp') ## (λ (bl: list[Byte]): read_seed(a', bl, cp'))))) </pre>
---	---

Expanding the definition of read_mem_cnt,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of random_side_effects_transformers_ok.1.1.1.4.4.4.3.

random_side_effects_transformers_ok.1.1.1.4.4.4.4:

<pre> {-1} d''' = bl' {-2} d'' = bl' {-3} d' = bl' {-4} transformer_invariant?(pm' states, singleton(expr_2_super(ok_result [Random_device_memory, list[Byte]] (bl')))) {-5} transformers_ok?(pm' states, singleton(expr_2_super(ok_result [Random_device_memory, list[Byte]] (bl')))) {-6} cons?(bl') \supset a' \neq rnd_seed \wedge a' \neq mem_cnt \wedge a' \neq rnd_val {-7} disjoint?(address_block(a', length(bl')), address_block(base, size)) {-8} (address_block(a', length(bl')) \subseteq (pm'ro_addr \cup pm'rw_addr)) </pre>	<pre> {1} \forall (d: (λ (bl: list[Byte]): bl = bl')): transformers_ok?(pm' states, singleton(expr_2_super(read_seed(a', d, cp')))) {2} transformers_ok?(pm' states, singleton(expr_2_super(read_mem_cnt(a', d''', cp') ## (λ (bl: list[Byte]): read_seed(a', bl, cp'))))) </pre>
---	---

Repeatedly Skolemizing and flattening,

Expanding the definition of read_seed,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of random_side_effects_transformers_ok.1.1.1.4.4.4.4.

random_side_effects_transformers_ok.1.1.2:

{-1}	transformers_ok?(pm' states, singleton(expr_2_super(ok_result [Random_device_memory, list[Byte]] (bl'))))
{-2}	cons?(bl') \supset $a' \neq \text{rnd_seed} \wedge a' \neq \text{mem_cnt} \wedge a' \neq \text{rnd_val}$
{-3}	disjoint?(address_block(a', length(bl')), address_block(base, size))
{-4}	(address_block(a', length(bl')) \subseteq (pm'ro_addr \cup pm'rw_addr))
{1}	transformer_invariant?(pm' states, singleton(expr_2_super(ok_result [Random_device_memory, list[Byte]] (bl'))))
{2}	transformers_ok?(pm' states, singleton(expr_2_super((unaligned_access(a', bl', cp') ## (λ (bl: list[Byte]): access_count(a', bl, cp') ## (λ (bl: list[Byte]): read_rnd_val(a', bl, cp') ## (λ (bl: list[Byte]): read_mem_cnt(a', bl, cp') ## (λ (bl: list[Byte]): read_seed (a', bl, cp'))))))))))))

Keeping (1) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of random_side_effects_transformers_ok.1.1.2.

random_side_effects_transformers_ok.1.2:

{-1}	cons?(bl') \supset $a' \neq \text{rnd_seed} \wedge a' \neq \text{mem_cnt} \wedge a' \neq \text{rnd_val}$
{-2}	disjoint?(address_block(a', length(bl')), address_block(base, size))
{-3}	(address_block(a', length(bl')) \subseteq (pm'ro_addr \cup pm'rw_addr))
{1}	transformers_ok?(pm' states, singleton(expr_2_super(ok_result [Random_device_memory, list[Byte]] (bl'))))
{2}	transformers_ok?(pm' states, singleton(expr_2_super((unaligned_access(a', bl', cp') ## (λ (bl: list[Byte]): access_count(a', bl, cp') ## (λ (bl: list[Byte]): read_rnd_val(a', bl, cp') ## (λ (bl: list[Byte]): read_mem_cnt(a', bl, cp') ## (λ (bl: list[Byte]): read_seed (a', bl, cp'))))))))))))

Keeping (1) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of random_side_effects_transformers_ok.1.2.

C.23 Proofs for Challenge_Random_Device (random_device.pvs)

random_side_effects_transformers_ok.2:

<div style="display: flex; flex-direction: column; align-items: center;"> <div style="border-bottom: 1px solid black; width: 100%;"></div> <div style="display: flex; align-items: center; width: 100%;"> {1} transformers_ok?(pm' states, dwse_super_transformers(pm' rw_addr, rnd_dev_write_side_effect)) </div> <div style="display: flex; align-items: center; width: 100%;"> {2} transformers_ok?(pm' states, (drse_super_transformers((pm' ro_addr ∪ pm' rw_addr), rnd_dev_read_side_effect) ∪ dwse_s </div> </div>	
---	--

Rewriting using transformers_ok_all_transformers, matching in *,

Expanding the definition of dwse_super_transformers,

Repeatedly Skolemizing and flattening,

Replacing using formula -3,

Keeping (-2 1) and hiding *,

Expanding the definition of rnd_dev_write_side_effect,

Using lemma in_blessed_memory_disjoint_device,

Using lemma in_blessed_memory_not_device_address,

Case splitting on subset?(address_block(a!1, length(bl!1)), union(pm' ro_addr, pm' rw_addr)),

we get 2 subgoals:

random_side_effects_transformers_ok.2.1:

<div style="display: flex; flex-direction: column; align-items: center;"> <div style="border-bottom: 1px solid black; width: 100%;"></div> <div style="display: flex; align-items: center; width: 100%;"> {-1} (address_block(a', length(bl')) ⊆ (pm' ro_addr ∪ pm' rw_addr)) </div> <div style="display: flex; align-items: center; width: 100%;"> {-2} (address_block(a', length(bl')) ⊆ (pm' ro_addr ∪ pm' rw_addr)) ∧ cons?(bl') ⊃ a' ≠ rnd_seed ∧ a' ≠ mem_cnt ∧ a' ≠ rnd_val </div> <div style="display: flex; align-items: center; width: 100%;"> {-3} (address_block(a', length(bl')) ⊆ (pm' ro_addr ∪ pm' rw_addr)) ⊃ disjoint?(address_block(a', length(bl')), address_block(base, size)) </div> <div style="display: flex; align-items: center; width: 100%;"> {-4} (address_block(a', length(bl')) ⊆ pm' rw_addr) </div> </div>	<div style="display: flex; flex-direction: column; align-items: center;"> <div style="border-bottom: 1px solid black; width: 100%;"></div> <div style="display: flex; align-items: center; width: 100%;"> {1} transformers_ok?(pm' states, singleton(expr_2_super((unaligned_access(a', bl', cp') ## (λ (bl: list[Byte]): access_count(a', bl, cp') ## (λ (bl: list[Byte]): write_rnd_dev(a', bl, cp') ## (λ (bl: list[Byte]): reset_count (a', bl, cp')))))))) </div> </div>
---	---

Case splitting on transformers_ok?(pm' states, singleton(expr_2_super(ok_result [Random_device_memory, list[Byte]] (bl!1)))),

we get 2 subgoals:

C Proof scripts

random_side_effects_transformers_ok.2.1.1.1:

<pre>{-1} transformers_ok?(pm' states, singleton(expr_2_super(ok_result [Random_device_memory, list[Byte]] (bl')))) {-2} (address_block(a', length(bl')) ⊆ (pm'ro_addr ∪ pm'rw_addr)) {-3} (address_block(a', length(bl')) ⊆ (pm'ro_addr ∪ pm'rw_addr)) ∧ cons?(bl') ⊃ a' ≠ rnd_seed ∧ a' ≠ mem_cnt ∧ a' ≠ rnd_val {-4} (address_block(a', length(bl')) ⊆ (pm'ro_addr ∪ pm'rw_addr)) ⊃ disjoint?(address_block(a', length(bl')), address_block(base, size)) {-5} (address_block(a', length(bl')) ⊆ pm'rw_addr)</pre>	<pre> singleton(expr_2_super((unaligned_access(a', bl', cp') ## (λ (bl: list[Byte]): access_count(a', bl, cp') ## (λ (bl: list[Byte]): write_rnd_dev(a', bl, cp') ## (λ (bl: list[Byte]): reset_count (a', bl, cp')))))))) </pre>
---	---

Case splitting on `transformer_invariant?(pm' states, singleton(expr_2_super (ok_result [Random_device_memory, list[Byte]] (bl!1))))`,

we get 2 subgoals:

random_side_effects_transformers_ok.2.1.1.1.1:

<pre>{-1} transformer_invariant?(pm' states, singleton(expr_2_super(ok_result [Random_device_memory, list[Byte]] (bl')))) {-2} transformers_ok?(pm' states, singleton(expr_2_super(ok_result [Random_device_memory, list[Byte]] (bl')))) {-3} (address_block(a', length(bl')) ⊆ (pm'ro_addr ∪ pm'rw_addr)) {-4} (address_block(a', length(bl')) ⊆ (pm'ro_addr ∪ pm'rw_addr)) ∧ cons?(bl') ⊃ a' ≠ rnd_seed ∧ a' ≠ mem_cnt ∧ a' ≠ rnd_val {-5} (address_block(a', length(bl')) ⊆ (pm'ro_addr ∪ pm'rw_addr)) ⊃ disjoint?(address_block(a', length(bl')), address_block(base, size)) {-6} (address_block(a', length(bl')) ⊆ pm'rw_addr)</pre>	<pre> singleton(expr_2_super((unaligned_access(a', bl', cp') ## (λ (bl: list[Byte]): access_count(a', bl, cp') ## (λ (bl: list[Byte]): write_rnd_dev(a', bl, cp') ## (λ (bl: list[Byte]): reset_count (a', bl, cp')))))))) </pre>
---	---

Rewriting using `fexpr_composition_transformers_ok`, matching in * where P gets LAMBDA (bl: list[Byte]): bl = bl!1,

we get 4 subgoals:

C.23 Proofs for Challenge_Random_Device (random_device.pvs)

random_side_effects_transformers_ok.2.1.1.1.1:

<pre>{-1} transformer_invariant?(pm' states, singleton(expr_2_super(ok_result [Random_device_memory, list[Byte]] (bl')))) {-2} transformers_ok?(pm' states, singleton(expr_2_super(ok_result [Random_device_memory, list[Byte]] (bl')))) {-3} (address_block(a', length(bl')) ⊆ (pm'ro_addr ∪ pm'rw_addr)) {-4} cons?(bl') ⊃ a' ≠ rnd_seed ∧ a' ≠ mem_cnt ∧ a' ≠ rnd_val {-5} TRUE ⊃ TRUE {-6} (address_block(a', length(bl')) ⊆ pm'rw_addr)</pre>	<pre> {1} transformers_ok?(pm' states, singleton(expr_2_super(unaligned_access(a', bl', cp')))) {2} transformers_ok?(pm' states, singleton(expr_2_super((unaligned_access(a', bl', cp') ## (λ (bl: list[Byte]): access_count(a', bl, cp') ## (λ (bl: list[Byte]): write_rnd_dev(a', bl, cp') ## (λ (bl: list[Byte]): reset_count (a', bl, cp')))))))))</pre>
--	--

Expanding the definition of unaligned_access,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of random_side_effects_transformers_ok.2.1.1.1.1.

random_side_effects_transformers_ok.2.1.1.1.2:

<pre>{-1} transformer_invariant?(pm' states, singleton(expr_2_super(ok_result [Random_device_memory, list[Byte]] (bl')))) {-2} transformers_ok?(pm' states, singleton(expr_2_super(ok_result [Random_device_memory, list[Byte]] (bl')))) {-3} (address_block(a', length(bl')) ⊆ (pm'ro_addr ∪ pm'rw_addr)) {-4} cons?(bl') ⊃ a' ≠ rnd_seed ∧ a' ≠ mem_cnt ∧ a' ≠ rnd_val {-5} TRUE ⊃ TRUE {-6} (address_block(a', length(bl')) ⊆ pm'rw_addr)</pre>	<pre> {1} transformer_invariant?(pm' states, singleton(expr_2_super(unaligned_access(a', bl', cp')))) {2} transformers_ok?(pm' states, singleton(expr_2_super((unaligned_access(a', bl', cp') ## (λ (bl: list[Byte]): access_count(a', bl, cp') ## (λ (bl: list[Byte]): write_rnd_dev(a', bl, cp') ## (λ (bl: list[Byte]): reset_count (a', bl, cp')))))))))</pre>
--	---

Expanding the definition of unaligned_access,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of random_side_effects_transformers_ok.2.1.1.1.2.

C Proof scripts

random_side_effects_transformers_ok.2.1.1.1.3:

```

{-1} transformer_invariant?(pm' states,
                             singleton(expr_2_super(ok_result [Random_device_memory, list[Byte]]
                                                    (bl'))))
{-2} transformers_ok?(pm' states,
                      singleton(expr_2_super(ok_result [Random_device_memory, list[Byte]]
                                                    (bl'))))
{-3} (address_block(a', length(bl')) ⊆ (pm'ro_addr ∪ pm'rw_addr))
{-4} cons?(bl') ⊃ a' ≠ rnd_seed ∧ a' ≠ mem_cnt ∧ a' ≠ rnd_val
{-5} TRUE ⊃ TRUE
{-6} (address_block(a', length(bl')) ⊆ pm'rw_addr)
-----
{1}  ∀ (s: (pm' states)):
      OK?(unaligned_access(a', bl', cp')(s) ⊃
           data(unaligned_access(a', bl', cp')(s)) = bl')
{2}  transformers_ok?(pm' states,
                      singleton(expr_2_super((unaligned_access(a', bl', cp') ##
                                                                (λ (bl: list[Byte]):
                                                                    access_count(a', bl, cp') ##
                                                                    (λ (bl: list[Byte]):
                                                                        write_rnd_dev(a', bl, cp') ##
                                                                        (λ (bl: list[Byte]):
                                                                            reset_count
                                                                              (a', bl, cp')))))))))

```

Expanding the definition of `unaligned_access`,

Repeatedly Skolemizing and flattening,

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `random_side_effects_transformers_ok.2.1.1.1.3`.

C.23 Proofs for Challenge_Random_Device (random_device.pvs)

random_side_effects_transformers_ok.2.1.1.1.4:

{-1}	transformer_invariant?(pm' states, singleton(expr_2_super(ok_result[Random_device_memory, list[Byte]] (bl'))))
{-2}	transformers_ok?(pm' states, singleton(expr_2_super(ok_result[Random_device_memory, list[Byte]] (bl'))))
{-3}	(address_block(a', length(bl')) ⊆ (pm'ro_addr ∪ pm'rw_addr))
{-4}	cons?(bl') ⊃ a' ≠ rnd_seed ∧ a' ≠ mem_cnt ∧ a' ≠ rnd_val
{-5}	TRUE ⊃ TRUE
{-6}	(address_block(a', length(bl')) ⊆ pm'rw_addr)
{1}	$\forall (d: (\lambda (bl: list[Byte]): bl = bl')):$ transformers_ok?(pm' states, singleton(expr_2_super(access_count(a', d, cp') ## ($\lambda (bl: list[Byte]):$ write_rnd_dev(a', bl, cp') ## ($\lambda (bl: list[Byte]):$ re- set_count(a', bl, cp'))))))))
{2}	transformers_ok?(pm' states, singleton(expr_2_super((unaligned_access(a', bl', cp') ## ($\lambda (bl: list[Byte]):$ access_count(a', bl, cp') ## ($\lambda (bl: list[Byte]):$ write_rnd_dev(a', bl, cp') ## ($\lambda (bl: list[Byte]):$ reset_count (a', bl, cp'))))))))

Repeatedly Skolemizing and flattening,

Hiding formulas: (-1 2),

Rewriting using fexpr_composition_transformers_ok, matching in * where P gets LAMBDA (bl: list[Byte]): bl = bl!1,

we get 4 subgoals:

C Proof scripts

random_side_effects_transformers_ok.2.1.1.1.4.1:

{-1}	$d' = \text{bl}'$
{-2}	transformer_invariant?(pm' states, singleton(expr_2_super(ok_result[Random_device_memory, list[Byte]] (bl'))))
{-3}	transformers_ok?(pm' states, singleton(expr_2_super(ok_result[Random_device_memory, list[Byte]] (bl'))))
{-4}	(address_block(a' , length(bl')) \subseteq (pm'ro_addr \cup pm'rw_addr))
{-5}	cons?(bl') $\supset a' \neq \text{rnd_seed} \wedge a' \neq \text{mem_cnt} \wedge a' \neq \text{rnd_val}$
{-6}	TRUE \supset TRUE
{-7}	(address_block(a' , length(bl')) \subseteq pm'rw_addr)
{1}	transformers_ok?(pm' states, singleton(expr_2_super(access_count(a' , d' , cp'))))
{2}	transformers_ok?(pm' states, singleton(expr_2_super(access_count(a' , d' , cp') ## (λ (bl: list[Byte]): write_rnd_dev(a' , bl, cp') ## (λ (bl: list[Byte]): re- set_count(a' , bl, cp'))))))

Expanding the definition of access_count,

Keeping (1) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of random_side_effects_transformers_ok.2.1.1.1.4.1.

random_side_effects_transformers_ok.2.1.1.1.4.2:

{-1}	$d' = \text{bl}'$
{-2}	transformer_invariant?(pm' states, singleton(expr_2_super(ok_result[Random_device_memory, list[Byte]] (bl'))))
{-3}	transformers_ok?(pm' states, singleton(expr_2_super(ok_result[Random_device_memory, list[Byte]] (bl'))))
{-4}	(address_block(a' , length(bl')) \subseteq (pm'ro_addr \cup pm'rw_addr))
{-5}	cons?(bl') $\supset a' \neq \text{rnd_seed} \wedge a' \neq \text{mem_cnt} \wedge a' \neq \text{rnd_val}$
{-6}	TRUE \supset TRUE
{-7}	(address_block(a' , length(bl')) \subseteq pm'rw_addr)
{1}	transformer_invariant?(pm' states, singleton(expr_2_super(access_count(a' , d' , cp'))))
{2}	transformers_ok?(pm' states, singleton(expr_2_super(access_count(a' , d' , cp') ## (λ (bl: list[Byte]): write_rnd_dev(a' , bl, cp') ## (λ (bl: list[Byte]): re- set_count(a' , bl, cp'))))))

Keeping (1) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of random_side_effects_transformers_ok.2.1.1.1.4.2.

random_side_effects_transformers_ok.2.1.1.1.4.3:

<pre> {-1} d' = bl' {-2} transformer_invariant?(pm' states, singleton(expr_2_super(ok_result [Random_device_memory, list[Byte]] (bl')))) {-3} transformers_ok?(pm' states, singleton(expr_2_super(ok_result [Random_device_memory, list[Byte]] (bl')))) {-4} (address_block(a', length(bl')) ⊆ (pm'ro_addr ∪ pm'rw_addr)) {-5} cons?(bl') ⊃ a' ≠ rnd_seed ∧ a' ≠ mem_cnt ∧ a' ≠ rnd_val {-6} TRUE ⊃ TRUE {-7} (address_block(a', length(bl')) ⊆ pm'rw_addr) </pre>	<hr/> <pre> {1} ∀ (s: (pm' states)): OK?(access_count(a', d', cp')(s) ⊃ data_access_count(a', d', cp')(s) = bl' {2} transformers_ok?(pm' states, singleton(expr_2_super(access_count(a', d', cp') ## (λ (bl: list[Byte]): write_rnd_dev(a', bl, cp') ## (λ (bl: list[Byte]): re- set_count(a', bl, cp')))))))) </pre>
---	---

Repeatedly Skolemizing and flattening,
Expanding the definition of access_count,
which is trivially true.

This completes the proof of random_side_effects_transformers_ok.2.1.1.1.4.3.

random_side_effects_transformers_ok.2.1.1.1.4.4:

<pre> {-1} d' = bl' {-2} transformer_invariant?(pm' states, singleton(expr_2_super(ok_result [Random_device_memory, list[Byte]] (bl')))) {-3} transformers_ok?(pm' states, singleton(expr_2_super(ok_result [Random_device_memory, list[Byte]] (bl')))) {-4} (address_block(a', length(bl')) ⊆ (pm'ro_addr ∪ pm'rw_addr)) {-5} cons?(bl') ⊃ a' ≠ rnd_seed ∧ a' ≠ mem_cnt ∧ a' ≠ rnd_val {-6} TRUE ⊃ TRUE {-7} (address_block(a', length(bl')) ⊆ pm'rw_addr) </pre>	<hr/> <pre> {1} ∀ (d: (λ (bl: list[Byte]): bl = bl')): transformers_ok?(pm' states, singleton(expr_2_super(write_rnd_dev(a', d, cp') ## (λ (bl: list[Byte]): reset_count(a', bl, cp')))))) {2} transformers_ok?(pm' states, singleton(expr_2_super(access_count(a', d', cp') ## (λ (bl: list[Byte]): write_rnd_dev(a', bl, cp') ## (λ (bl: list[Byte]): re- set_count(a', bl, cp')))))))) </pre>
---	--

Repeatedly Skolemizing and flattening,

C Proof scripts

Hiding formulas: (-1 2),

Rewriting using `fexpr_composition_transformers_ok`, matching in `*` where P gets LAMBDA (bl: list[Byte]): bl = bl!1,

we get 4 subgoals:

`random_side_effects_transformers_ok.2.1.1.1.4.4.1:`

<pre>{-1} d'' = bl' {-2} d' = bl' {-3} transformer_invariant?(pm' states, singleton(expr_2_super(ok_result [Random_device_memory, list[Byte]] (bl')))) {-4} transformers_ok?(pm' states, singleton(expr_2_super(ok_result [Random_device_memory, list[Byte]] (bl')))) {-5} (address_block(a', length(bl')) ⊆ (pm'ro_addr ∪ pm'rw_addr)) {-6} cons?(bl') ⊃ a' ≠ rnd_seed ∧ a' ≠ mem_cnt ∧ a' ≠ rnd_val {-7} TRUE ⊃ TRUE {-8} (address_block(a', length(bl')) ⊆ pm'rw_addr)</pre>	<pre> {1} transformers_ok?(pm' states, singleton(expr_2_super(write_rnd_dev(a', d'', cp')))) {2} transformers_ok?(pm' states, singleton(expr_2_super(write_rnd_dev(a', d'', cp') ## (λ (bl: list[Byte]): reset_count(a', bl, cp'))))) </pre>
---	---

Hiding formulas: 2,

Expanding the definition of `write_rnd_dev`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `random_side_effects_transformers_ok.2.1.1.1.4.4.1`.

`random_side_effects_transformers_ok.2.1.1.1.4.4.2:`

<pre>{-1} d'' = bl' {-2} d' = bl' {-3} transformer_invariant?(pm' states, singleton(expr_2_super(ok_result [Random_device_memory, list[Byte]] (bl')))) {-4} transformers_ok?(pm' states, singleton(expr_2_super(ok_result [Random_device_memory, list[Byte]] (bl')))) {-5} (address_block(a', length(bl')) ⊆ (pm'ro_addr ∪ pm'rw_addr)) {-6} cons?(bl') ⊃ a' ≠ rnd_seed ∧ a' ≠ mem_cnt ∧ a' ≠ rnd_val {-7} TRUE ⊃ TRUE {-8} (address_block(a', length(bl')) ⊆ pm'rw_addr)</pre>	<pre> {1} transformer_invariant?(pm' states, singleton(expr_2_super(write_rnd_dev(a', d'', cp')))) {2} transformers_ok?(pm' states, singleton(expr_2_super(write_rnd_dev(a', d'', cp') ## (λ (bl: list[Byte]): reset_count(a', bl, cp'))))) </pre>
---	---

Hiding formulas: 2,

Expanding the definition of `write_rnd_dev`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `random_side_effects_transformers_ok.2.1.1.1.4.4.2`.

random_side_effects_transformers_ok.2.1.1.1.4.4.3:

<pre> {-1} d'' = bl' {-2} d' = bl' {-3} transformer_invariant?(pm' states, singleton(expr_2_super(ok_result[Random_device_memory, list[Byte]] (bl')))) {-4} transformers_ok?(pm' states, singleton(expr_2_super(ok_result[Random_device_memory, list[Byte]] (bl')))) {-5} (address_block(a', length(bl')) ⊆ (pm'ro_addr ∪ pm'rw_addr)) {-6} cons?(bl') ⊃ a' ≠ rnd_seed ∧ a' ≠ mem_cnt ∧ a' ≠ rnd_val {-7} TRUE ⊃ TRUE {-8} (address_block(a', length(bl')) ⊆ pm'rw_addr) </pre>	<pre> {1} ∀ (s: (pm' states)): OK?(write_rnd_dev(a', d'', cp')(s) ⊃ data(write_rnd_dev(a', d'', cp')(s) = bl') {2} transformers_ok?(pm' states, singleton(expr_2_super(write_rnd_dev(a', d'', cp') ## (λ (bl: list[Byte]): reset_count(a', bl, cp'))))) </pre>
---	--

Repeatedly Skolemizing and flattening,

Expanding the definition of write_rnd_dev,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of random_side_effects_transformers_ok.2.1.1.1.4.4.3.

random_side_effects_transformers_ok.2.1.1.1.4.4.4:

<pre> {-1} d'' = bl' {-2} d' = bl' {-3} transformer_invariant?(pm' states, singleton(expr_2_super(ok_result[Random_device_memory, list[Byte]] (bl')))) {-4} transformers_ok?(pm' states, singleton(expr_2_super(ok_result[Random_device_memory, list[Byte]] (bl')))) {-5} (address_block(a', length(bl')) ⊆ (pm'ro_addr ∪ pm'rw_addr)) {-6} cons?(bl') ⊃ a' ≠ rnd_seed ∧ a' ≠ mem_cnt ∧ a' ≠ rnd_val {-7} TRUE ⊃ TRUE {-8} (address_block(a', length(bl')) ⊆ pm'rw_addr) </pre>	<pre> {1} ∀ (d: (λ (bl: list[Byte]): bl = bl')): transformers_ok?(pm' states, singleton(expr_2_super(reset_count(a', d, cp')))) {2} transformers_ok?(pm' states, singleton(expr_2_super(write_rnd_dev(a', d'', cp') ## (λ (bl: list[Byte]): reset_count(a', bl, cp'))))) </pre>
---	---

Repeatedly Skolemizing and flattening,

Expanding the definition of reset_count,

Hiding formulas: (-1 2 -9),

Replacing using formula -1,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of random_side_effects_transformers_ok.2.1.1.1.4.4.4.

random_side_effects_transformers_ok.2.1.1.2:

<pre>{-1} transformers_ok?(pm' states, singleton(expr_2_super(ok_result [Random_device_memory, list[Byte]] (bl')))) {-2} (address_block(a', length(bl')) ⊆ (pm'ro_addr ∪ pm'rw_addr)) {-3} (address_block(a', length(bl')) ⊆ (pm'ro_addr ∪ pm'rw_addr)) ∧ cons?(bl') ⊃ a' ≠ rnd_seed ∧ a' ≠ mem_cnt ∧ a' ≠ rnd_val {-4} (address_block(a', length(bl')) ⊆ (pm'ro_addr ∪ pm'rw_addr)) ⊃ disjoint?(address_block(a', length(bl')), address_block(base, size)) {-5} (address_block(a', length(bl')) ⊆ pm'rw_addr)</pre>	<pre>singleton(expr_2_super(ok_result [Random_device_memory, list[Byte]] (bl'))))</pre>
<pre>{1} transformer_invariant?(pm' states, singleton(expr_2_super(ok_result [Random_device_memory, list[Byte]] (bl')))) {2} transformers_ok?(pm' states, singleton(expr_2_super((unaligned_access(a', bl', cp') ## (λ (bl: list[Byte]): access_count(a', bl, cp') ## (λ (bl: list[Byte]): write_rnd_dev(a', bl, cp') ## (λ (bl: list[Byte]): reset_count (a', bl, cp'))))))))</pre>	<pre>singleton(expr_2_super((unaligned_access(a', bl', cp') ## (λ (bl: list[Byte]): access_count(a', bl, cp') ## (λ (bl: list[Byte]): write_rnd_dev(a', bl, cp') ## (λ (bl: list[Byte]): reset_count (a', bl, cp'))))))))</pre>

Keeping (1) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of random_side_effects_transformers_ok.2.1.1.2.

random_side_effects_transformers_ok.2.1.2:

<pre>{-1} (address_block(a', length(bl')) ⊆ (pm'ro_addr ∪ pm'rw_addr)) {-2} (address_block(a', length(bl')) ⊆ (pm'ro_addr ∪ pm'rw_addr)) ∧ cons?(bl') ⊃ a' ≠ rnd_seed ∧ a' ≠ mem_cnt ∧ a' ≠ rnd_val {-3} (address_block(a', length(bl')) ⊆ (pm'ro_addr ∪ pm'rw_addr)) ⊃ disjoint?(address_block(a', length(bl')), address_block(base, size)) {-4} (address_block(a', length(bl')) ⊆ pm'rw_addr)</pre>	<pre>singleton(expr_2_super(ok_result [Random_device_memory, list[Byte]] (bl'))))</pre>
<pre>{1} transformers_ok?(pm' states, singleton(expr_2_super((unaligned_access(a', bl', cp') ## (λ (bl: list[Byte]): access_count(a', bl, cp') ## (λ (bl: list[Byte]): write_rnd_dev(a', bl, cp') ## (λ (bl: list[Byte]): reset_count (a', bl, cp'))))))))</pre>	<pre>singleton(expr_2_super((unaligned_access(a', bl', cp') ## (λ (bl: list[Byte]): access_count(a', bl, cp') ## (λ (bl: list[Byte]): write_rnd_dev(a', bl, cp') ## (λ (bl: list[Byte]): reset_count (a', bl, cp'))))))))</pre>

Keeping (1) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of random_side_effects_transformers_ok.2.1.2.

random_side_effects_transformers_ok.2.2:

<pre>{-1} (address_block(a', length(bl')) ⊆ (pm'ro_addr ∪ pm'rw_addr)) ∧ cons?(bl') ⊃ a' ≠ rnd_seed ∧ a' ≠ mem_cnt ∧ a' ≠ rnd_val {-2} (address_block(a', length(bl')) ⊆ (pm'ro_addr ∪ pm'rw_addr)) ⊃ disjoint?(address_block(a', length(bl')), address_block(base, size)) {-3} (address_block(a', length(bl')) ⊆ pm'rw_addr)</pre>	<hr style="border: 0.5px solid black;"/> <pre>{1} (address_block(a', length(bl')) ⊆ (pm'ro_addr ∪ pm'rw_addr)) {2} transformers_ok?(pm'states, singleton(expr_2_super((unaligned_access(a', bl', cp') ## (λ (bl: list[Byte]): access_count(a', bl, cp') ## (λ (bl: list[Byte]): write_rnd_dev(a', bl, cp') ## (λ (bl: list[Byte]): reset_count (a', bl, cp'))))))))</pre>
---	---

Keeping (-3 1) and hiding *,

Rewriting using subset_bigger_union_right, matching in *,

This completes the proof of random_side_effects_transformers_ok.2.2.

Q.E.D.

C.23.6 Challenge_Random_Device.random_side_effects_side_effect_content_unchanged_read

Terse proof for random_side_effects_side_effect_content_unchanged_read.

random_side_effects_side_effect_content_unchanged_read:

<pre>{1} side_effect_content_unchanged((pm'ro_addr ∪ pm'rw_addr), pm'states, rnd_dev_read_side_effect)</pre>	
--	--

Case splitting on size(uidt(dt_uint)) > 0,

we get 2 subgoals:

random_side_effects_side_effect_content_unchanged_read.1:

<pre>{-1} size(uidt(dt_uint)) > 0</pre>	<hr style="border: 0.5px solid black;"/> <pre>{1} side_effect_content_unchanged((pm'ro_addr ∪ pm'rw_addr), pm'states, rnd_dev_read_side_effect)</pre>
--	---

Installing automatic rewrites from: ok_expr_2_super has_next_state_expr_2_super fatal_result state_expr_2_super ok_result_state ok_result_ok expr_2_super_ok_result expr_2_super_fatal_result has_next_state_ok_result has_next_state_fatal_result ok_result_data ertimes_ax

Expanding the definition of rnd_dev_read_side_effect,

Rewriting using side_effect_content_unchanged_composition, matching in *,

we get 3 subgoals:

random_side_effects_side_effect_content_unchanged_read.1.1:

{-1}	size(uidt(dt_uint)) > 0
{1}	transformer_invariant?(pm'states, se_super_transformers((pm'ro_addr ∪ pm'rw_addr), un- aligned_access))
{2}	side_effect_content_unchanged((pm'ro_addr ∪ pm'rw_addr), pm'states, λ (a: Address, bl: list[Byte], cp: bool): (unaligned_access(a, bl, cp) ## (λ (bl: list[Byte]): access_count(a, bl, cp) ## (λ (bl: list[Byte]): read_rnd_val(a, bl, cp) ## (λ (bl: list[Byte]): read_mem_cnt(a, bl, cp) ## (λ (bl: list[Byte]): read_seed(a, bl, cp))))))))))

Rewriting using transformer_invariant_all_transformers, matching in *,

Expanding the definition of se_super_transformers,

Repeatedly Skolemizing and flattening,

Replacing using formula -3,

Keeping (-2 1) and hiding *,

Expanding the definition of unaligned_access,

Rewriting using in_blessed_memory_disjoint_device, matching in *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of random_side_effects_side_effect_content_unchanged_read.1.1.

random_side_effects_side_effect_content_unchanged_read.1.2:

{-1}	size(uidt(dt_uint)) > 0
{1}	side_effect_content_unchanged((pm'ro_addr ∪ pm'rw_addr), pm'states, unaligned_access)
{2}	side_effect_content_unchanged((pm'ro_addr ∪ pm'rw_addr), pm'states, λ (a: Address, bl: list[Byte], cp: bool): (unaligned_access(a, bl, cp) ## (λ (bl: list[Byte]): access_count(a, bl, cp) ## (λ (bl: list[Byte]): read_rnd_val(a, bl, cp) ## (λ (bl: list[Byte]): read_mem_cnt(a, bl, cp) ## (λ (bl: list[Byte]): read_seed(a, bl, cp))))))))))

Hiding formulas: 2,

Expanding the definition of side_effect_content_unchanged,

Repeatedly Skolemizing and flattening,

Expanding the definition of unaligned_access,

Rewriting using in_blessed_memory_disjoint_device, matching in *,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of random_side_effects_side_effect_content_unchanged_read.1.2.

random_side_effects_side_effect_content_unchanged_read.1.3:

{-1}	size(uidt(dt_uint)) > 0
{1}	side_effect_content_unchanged((pm'ro_addr ∪ pm'rw_addr), pm'states, λ (a: Address, bl: list[Byte], cp: bool): access_count(a, bl, cp) ## (λ (bl: list[Byte]): read_rnd_val(a, bl, cp) ## (λ (bl: list[Byte]): read_mem_cnt(a, bl, cp) ## (λ (bl: list[Byte]): read_seed(a, bl, cp))))))
{2}	side_effect_content_unchanged((pm'ro_addr ∪ pm'rw_addr), pm'states, λ (a: Address, bl: list[Byte], cp: bool): unaligned_access(a, bl, cp) ## (λ (bl: list[Byte]): access_count(a, bl, cp) ## (λ (bl: list[Byte]): read_rnd_val(a, bl, cp) ## (λ (bl: list[Byte]): read_mem_cnt(a, bl, cp) ## (λ (bl: list[Byte]): read_seed(a, bl, cp))))))

Hiding formulas: 2,

Rewriting using side_effect_content_unchanged_composition, matching in *,

we get 3 subgoals:

random_side_effects_side_effect_content_unchanged_read.1.3.1:

{-1}	size(uidt(dt_uint)) > 0
{1}	transformer_invariant?(pm'states, se_super_transformers((pm'ro_addr ∪ pm'rw_addr), ac- cess_count))
{2}	side_effect_content_unchanged((pm'ro_addr ∪ pm'rw_addr), pm'states, λ (a: Address, bl: list[Byte], cp: bool): access_count(a, bl, cp) ## (λ (bl: list[Byte]): read_rnd_val(a, bl, cp) ## (λ (bl: list[Byte]): read_mem_cnt(a, bl, cp) ## (λ (bl: list[Byte]): read_seed(a, bl, cp))))

Rewriting using transformer_invariant_all_transformers, matching in *,

Expanding the definition of se_super_transformers,

Repeatedly Skolemizing and flattening,

Replacing using formula -3,

Keeping (-2 1) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of random_side_effects_side_effect_content_unchanged_read.1.3.1.

random_side_effects_side_effect_content_unchanged_read.1.3.2:

{-1}	size(uidt(dt_uint)) > 0
{1}	side_effect_content_unchanged((pm'ro_addr ∪ pm'rw_addr), pm'states, access_count)
{2}	side_effect_content_unchanged((pm'ro_addr ∪ pm'rw_addr), pm'states, λ (a: Address, bl: list[Byte], cp: bool): access_count(a, bl, cp) ## (λ (bl: list[Byte]): read_rnd_val(a, bl, cp) ## (λ (bl: list[Byte]): read_mem_cnt(a, bl, cp) ## (λ (bl: list[Byte]): read_seed(a, bl, cp))))))

Hiding formulas: 2,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of random_side_effects_side_effect_content_unchanged_read.1.3.2.

random_side_effects_side_effect_content_unchanged_read.1.3.3:

{-1}	size(uidt(dt_uint)) > 0
{1}	side_effect_content_unchanged((pm'ro_addr ∪ pm'rw_addr), pm'states, λ (a: Address, bl: list[Byte], cp: bool): read_rnd_val(a, bl, cp) ## (λ (bl: list[Byte]): read_mem_cnt(a, bl, cp) ## (λ (bl: list[Byte]): read_seed(a, bl, cp))))
{2}	side_effect_content_unchanged((pm'ro_addr ∪ pm'rw_addr), pm'states, λ (a: Address, bl: list[Byte], cp: bool): access_count(a, bl, cp) ## (λ (bl: list[Byte]): read_rnd_val(a, bl, cp) ## (λ (bl: list[Byte]): read_mem_cnt(a, bl, cp) ## (λ (bl: list[Byte]): read_seed(a, bl, cp))))))

Hiding formulas: 2,

Rewriting using side_effect_content_unchanged_composition, matching in *,

we get 3 subgoals:

random_side_effects_side_effect_content_unchanged_read.1.3.3.1:

{-1}	size(uidt(dt_uint)) > 0
{1}	transformer_invariant?(pm'states, se_super_transformers((pm'ro_addr ∪ pm'rw_addr), read_rnd_val))
{2}	side_effect_content_unchanged((pm'ro_addr ∪ pm'rw_addr), pm'states, λ (a: Address, bl: list[Byte], cp: bool): read_rnd_val(a, bl, cp) ## (λ (bl: list[Byte]): read_mem_cnt(a, bl, cp) ## (λ (bl: list[Byte]): read_seed(a, bl, cp))))

Rewriting using transformer_invariant_all_transformers, matching in *,

Expanding the definition of se_super_transformers,

Repeatedly Skolemizing and flattening,

Replacing using formula -3,

Keeping (-2 1) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of random_side_effects_side_effect_content_unchanged_read.1.3.3.1.

random_side_effects_side_effect_content_unchanged_read.1.3.3.2:

{-1}	size(uidt(dt_uint)) > 0
{1}	side_effect_content_unchanged((pm'ro_addr ∪ pm'rw_addr), pm'states, read_rnd_val)
{2}	side_effect_content_unchanged((pm'ro_addr ∪ pm'rw_addr), pm'states, $\lambda (a: \text{Address}, \text{bl}: \text{list}[\text{Byte}], \text{cp}: \text{bool}):$ $\text{read_rnd_val}(a, \text{bl}, \text{cp}) \#\#$ $(\lambda (\text{bl}: \text{list}[\text{Byte}]):$ $\text{read_mem_cnt}(a, \text{bl}, \text{cp}) \#\#$ $(\lambda (\text{bl}: \text{list}[\text{Byte}]): \text{read_seed}(a, \text{bl}, \text{cp})))$

Hiding formulas: 2,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of random_side_effects_side_effect_content_unchanged_read.1.3.3.2.

random_side_effects_side_effect_content_unchanged_read.1.3.3.3:

{-1}	size(uidt(dt_uint)) > 0
{1}	side_effect_content_unchanged((pm'ro_addr ∪ pm'rw_addr), pm'states, $\lambda (a: \text{Address}, \text{bl}: \text{list}[\text{Byte}], \text{cp}: \text{bool}):$ $\text{read_mem_cnt}(a, \text{bl}, \text{cp}) \#\#$ $(\lambda (\text{bl}: \text{list}[\text{Byte}]): \text{read_seed}(a, \text{bl}, \text{cp})))$
{2}	side_effect_content_unchanged((pm'ro_addr ∪ pm'rw_addr), pm'states, $\lambda (a: \text{Address}, \text{bl}: \text{list}[\text{Byte}], \text{cp}: \text{bool}):$ $\text{read_rnd_val}(a, \text{bl}, \text{cp}) \#\#$ $(\lambda (\text{bl}: \text{list}[\text{Byte}]):$ $\text{read_mem_cnt}(a, \text{bl}, \text{cp}) \#\#$ $(\lambda (\text{bl}: \text{list}[\text{Byte}]): \text{read_seed}(a, \text{bl}, \text{cp})))$

Hiding formulas: 2,

Rewriting using side_effect_content_unchanged_composition, matching in *,

we get 3 subgoals:

random_side_effects_side_effect_content_unchanged_read.1.3.3.3.1:

{-1}	size(uidt(dt_uint)) > 0
{1}	transformer_invariant?(pm'states, $\text{se_super_transformers}((\text{pm'ro_addr} \cup \text{pm'rw_addr}), \text{read_mem_cnt}))$
{2}	side_effect_content_unchanged((pm'ro_addr ∪ pm'rw_addr), pm'states, $\lambda (a: \text{Address}, \text{bl}: \text{list}[\text{Byte}], \text{cp}: \text{bool}):$ $\text{read_mem_cnt}(a, \text{bl}, \text{cp}) \#\#$ $(\lambda (\text{bl}: \text{list}[\text{Byte}]): \text{read_seed}(a, \text{bl}, \text{cp})))$

Rewriting using transformer_invariant_all_transformers, matching in *,

Expanding the definition of se_super_transformers,

Repeatedly Skolemizing and flattening,

Replacing using formula -3,

Keeping (-2 1) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of random_side_effects_side_effect_content_unchanged_read.1.3.3.3.1.

random_side_effects_side_effect_content_unchanged_read.1.3.3.3.2:

{-1}	size(uidt(dt_uint)) > 0
{1}	side_effect_content_unchanged((pm'ro_addr ∪ pm'rw_addr), pm'states, read_mem_cnt)
{2}	side_effect_content_unchanged((pm'ro_addr ∪ pm'rw_addr), pm'states, λ (a: Address, bl: list[Byte], cp: bool): read_mem_cnt(a, bl, cp) ## (λ (bl: list[Byte]): read_seed(a, bl, cp)))

Hiding formulas: 2,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of random_side_effects_side_effect_content_unchanged_read.1.3.3.3.2.

random_side_effects_side_effect_content_unchanged_read.1.3.3.3.3:

{-1}	size(uidt(dt_uint)) > 0
{1}	side_effect_content_unchanged((pm'ro_addr ∪ pm'rw_addr), pm'states, read_seed)
{2}	side_effect_content_unchanged((pm'ro_addr ∪ pm'rw_addr), pm'states, λ (a: Address, bl: list[Byte], cp: bool): read_mem_cnt(a, bl, cp) ## (λ (bl: list[Byte]): read_seed(a, bl, cp)))

Hiding formulas: 2,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of random_side_effects_side_effect_content_unchanged_read.1.3.3.3.3.

random_side_effects_side_effect_content_unchanged_read.2:

{1}	size(uidt(dt_uint)) > 0
{2}	side_effect_content_unchanged((pm'ro_addr ∪ pm'rw_addr), pm'states, rnd_dev_read_side_effect)

Using lemma dt_uint_size,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of random_side_effects_side_effect_content_unchanged_read.2.

Q.E.D.

C.23.7 Chal-

lenge_Random_Device.random_side_effects_side_effect_content_unchanged_w

Terse proof for random_side_effects_side_effect_content_unchanged_write.

random_side_effects_side_effect_content_unchanged_write:

{1}	side_effect_content_unchanged(pm'rw_addr, pm'states, rnd_dev_write_side_effect)
-----	---

Case splitting on size(uidt(dt_uint)) > 0,

we get 2 subgoals:

random_side_effects_side_effect_content_unchanged_write.1:

{-1}	size(uidt(dt_uint)) > 0
{1}	side_effect_content_unchanged(pm'rw_addr, pm'states, rnd_dev_write_side_effect)

Installing automatic rewrites from: ok_expr_2_super has_next_state_expr_2_super fatal_result state_expr_2_super ok_result_state ok_result_ok expr_2_super_ok_result expr_2_super_fatal_result has_next_state_ok_result has_next_state_fatal_result ok_result_data ertimes_ax

Expanding the definition of rnd_dev_write_side_effect,

Rewriting using side_effect_content_unchanged_composition, matching in *,

we get 3 subgoals:

random_side_effects_side_effect_content_unchanged_write.1.1.1:

{-1}	size(uidt(dt_uint)) > 0
{1}	transformer_invariant?(pm'states, se_super_transformers(pm'rw_addr, unaligned_access))
{2}	side_effect_content_unchanged(pm'rw_addr, pm'states, $\lambda (a: \text{Address}, \text{bl}: \text{list}[\text{Byte}], \text{cp}: \text{bool}):$ $(\text{unaligned_access}(a, \text{bl}, \text{cp}) \#\#$ $(\lambda (\text{bl}: \text{list}[\text{Byte}]):$ $\text{access_count}(a, \text{bl}, \text{cp}) \#\#$ $(\lambda (\text{bl}: \text{list}[\text{Byte}]):$ $\text{write_rnd_dev}(a, \text{bl}, \text{cp}) \#\#$ $(\lambda (\text{bl}: \text{list}[\text{Byte}]):$ $\text{reset_count}(a, \text{bl}, \text{cp}))))))$

Rewriting using transformer_invariant_all_transformers, matching in *,

Expanding the definition of se_super_transformers,

Repeatedly Skolemizing and flattening,

Replacing using formula -3,

Keeping (-2 1) and hiding *,

Expanding the definition of unaligned_access,

Rewriting using in_blessed_memory_disjoint_device, matching in *,

we get 2 subgoals:

random_side_effects_side_effect_content_unchanged_write.1.1.1.1:

{-1}	(address_block(a', length(bl')) \subseteq pm'rw_addr)
{1}	transformer_invariant?(pm'states, singleton(expr_2_super(ok_result(bl'))))

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of random_side_effects_side_effect_content_unchanged_write.1.1.1.

random_side_effects_side_effect_content_unchanged_write.1.1.2:

{-1}	(address_block(a', length(bl')) \subseteq pm'rw_addr)
{1}	(address_block(a', length(bl')) \subseteq (pm'ro_addr \cup pm'rw_addr))
{2}	transformer_invariant?(pm'states, $\text{IF } \text{null?}(\text{bl}') \vee$ $\text{disjoint?}(\text{address_block}(a', \text{length}(\text{bl}')),$ $\text{address_block}(\text{base}, \text{size}))$ \vee $(\neg \text{cp}' \wedge$ $\text{length}(\text{bl}') = \text{size}(\text{uidt}(\text{dt_uint})) \wedge$ $(a' = \text{rnd_seed} \vee a' = \text{mem_cnt} \vee a' = \text{rnd_val}))$ $\text{THEN } \text{singleton}(\text{expr_2_super}(\text{ok_result}(\text{bl}')))$ $\text{ELSE } \text{singleton}(\text{expr_2_super}(\text{fatal_result}))$ $\text{ENDIF})$

Rewriting using subset_bigger_union_right, matching in *,

This completes the proof of random_side_effects_side_effect_content_unchanged_write.1.1.2.

C Proof scripts

`random_side_effects_side_effect_content_unchanged_write.1.2:`

{-1}	<code>size(uidt(dt_uint)) > 0</code>
{1}	<code>side_effect_content_unchanged(pm'rw_addr, pm'states, unaligned_access)</code>
{2}	<code>side_effect_content_unchanged(pm'rw_addr, pm'states,</code> $\lambda (a: \text{Address}, bl: \text{list}[\text{Byte}], cp: \text{bool}):$ $(\text{unaligned_access}(a, bl, cp) \#\#$ $(\lambda (bl: \text{list}[\text{Byte}]):$ $\text{access_count}(a, bl, cp) \#\#$ $(\lambda (bl: \text{list}[\text{Byte}]):$ $\text{write_rnd_dev}(a, bl, cp) \#\#$ $(\lambda (bl: \text{list}[\text{Byte}]):$ $\text{reset_count}(a, bl, cp))))))$

Hiding formulas: 2,

Expanding the definition of `side_effect_content_unchanged`,

Repeatedly Skolemizing and flattening,

Expanding the definition of `unaligned_access`,

Rewriting using `in_blessed_memory_disjoint_device`, matching in *,

we get 2 subgoals:

`random_side_effects_side_effect_content_unchanged_write.1.2.1:`

{-1}	<code>every($\lambda (x: \text{number}):$</code> $\text{number_field_pred}(x) \wedge$ $\text{real_pred}(x) \wedge$ $\text{rational_pred}(x) \wedge \text{integer_pred}(x) \wedge x \geq 0 \wedge x < \text{max_byte}$
	(bl')
{-2}	<code>pm'states(s')</code>
{-3}	<code>(address_block(a', length(bl')) \subseteq pm'rw_addr)</code>
{-4}	<code>OK?(ok_result(bl')(s'))</code>
{-5}	<code>size(uidt(dt_uint)) > 0</code>
{1}	<code>data(ok_result(bl')(s')) = bl'</code>

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `random_side_effects_side_effect_content_unchanged_write.1.2.1`.

random_side_effects_side_effect_content_unchanged_write.1.2.2:

{-1}	<pre> every(λ (x: number): number_field_pred(x) \wedge real_pred(x) \wedge rational_pred(x) \wedge integer_pred(x) \wedge $x \geq 0 \wedge x < \text{max_byte}$) (bl') </pre>
{-2}	pm'states(s')
{-3}	(address_block(a' , length(bl')) \subseteq pm'rw_addr)
{-4}	<pre> OK?(IF null?(bl') \vee disjoint?(address_block(a', length(bl')), address_block(base, size)) \vee (\neg cp' \wedge length(bl') = size(uidt(dt_uint)) \wedge ($a' = \text{rnd_seed} \vee a' = \text{mem_cnt} \vee a' = \text{rnd_val}$)) THEN ok_result(bl')(s') ELSE Fatal ENDIF) </pre>
{-5}	size(uidt(dt_uint)) > 0
{1}	(address_block(a' , length(bl')) \subseteq (pm'ro_addr \cup pm'rw_addr))
{2}	<pre> IF null?(bl') \vee disjoint?(address_block(a', length(bl')), address_block(base, size)) \vee (\neg cp' \wedge length(bl') = size(uidt(dt_uint)) \wedge ($a' = \text{rnd_seed} \vee a' = \text{mem_cnt} \vee a' = \text{rnd_val}$)) THEN bl' ELSE data(Fatal) ENDIF = bl' </pre>

Rewriting using subset_bigger_union_right, matching in *,

This completes the proof of random_side_effects_side_effect_content_unchanged_write.1.2.2.

random_side_effects_side_effect_content_unchanged_write.1.3:

{-1}	size(uidt(dt_uint)) > 0
{1}	<pre> side_effect_content_unchanged(pm'rw_addr, pm'states, λ (a: Address, bl: list[Byte], cp: bool): access_count(a, bl, cp) ## (λ (bl: list[Byte]): write_rnd_dev(a, bl, cp) ## (λ (bl: list[Byte]): re- </pre>
{2}	<pre> set_count(a, bl, cp))) side_effect_content_unchanged(pm'rw_addr, pm'states, λ (a: Address, bl: list[Byte], cp: bool): (unaligned_access(a, bl, cp) ## (λ (bl: list[Byte]): access_count(a, bl, cp) ## (λ (bl: list[Byte]): write_rnd_dev(a, bl, cp) ## (λ (bl: list[Byte]): reset_count(a, bl, cp)))))) </pre>

Hiding formulas: 2,

Rewriting using side_effect_content_unchanged_composition, matching in *,

we get 3 subgoals:

random_side_effects_side_effect_content_unchanged_write.1.3.1:

{-1}	size(uidt(dt_uint)) > 0
{1}	transformer_invariant?(pm'rw_addr, pm'states, se_super_transformers(pm'rw_addr, access_count))
{2}	side_effect_content_unchanged(pm'rw_addr, pm'states, λ (a: Address, bl: list[Byte], cp: bool): access_count(a, bl, cp) ## (λ (bl: list[Byte]): write_rnd_dev(a, bl, cp) ## (λ (bl: list[Byte]): re- set_count(a, bl, cp))))

Rewriting using transformer_invariant_all_transformers, matching in *,

Expanding the definition of se_super_transformers,

Repeatedly Skolemizing and flattening,

Replacing using formula -3,

Keeping (-2 1) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of random_side_effects_side_effect_content_unchanged_write.1.3.1.

random_side_effects_side_effect_content_unchanged_write.1.3.2:

{-1}	size(uidt(dt_uint)) > 0
{1}	side_effect_content_unchanged(pm'rw_addr, pm'states, access_count)
{2}	side_effect_content_unchanged(pm'rw_addr, pm'states, λ (a: Address, bl: list[Byte], cp: bool): access_count(a, bl, cp) ## (λ (bl: list[Byte]): write_rnd_dev(a, bl, cp) ## (λ (bl: list[Byte]): re- set_count(a, bl, cp))))

Hiding formulas: 2,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of random_side_effects_side_effect_content_unchanged_write.1.3.2.

random_side_effects_side_effect_content_unchanged_write.1.3.3:

{-1}	size(uidt(dt_uint)) > 0
{1}	side_effect_content_unchanged(pm'rw_addr, pm'states, λ (a: Address, bl: list[Byte], cp: bool): write_rnd_dev(a, bl, cp) ## (λ (bl: list[Byte]): reset_count(a, bl, cp)))
{2}	side_effect_content_unchanged(pm'rw_addr, pm'states, λ (a: Address, bl: list[Byte], cp: bool): access_count(a, bl, cp) ## (λ (bl: list[Byte]): write_rnd_dev(a, bl, cp) ## (λ (bl: list[Byte]): re- set_count(a, bl, cp))))

Hiding formulas: 2,

Rewriting using side_effect_content_unchanged_composition, matching in *,

we get 3 subgoals:

random_side_effects_side_effect_content_unchanged_write.1.3.3.1:

{-1}	size(uidt(dt_uint)) > 0
{1}	transformer_invariant?(pm' states, se_super_transformers(pm' rw_addr, write_rnd_dev))
{2}	side_effect_content_unchanged(pm' rw_addr, pm' states, $\lambda (a: \text{Address}, bl: \text{list}[\text{Byte}], cp: \text{bool}):$ $\text{write_rnd_dev}(a, bl, cp) \#\#$ $(\lambda (bl: \text{list}[\text{Byte}]): \text{reset_count}(a, bl, cp))$)

Rewriting using transformer_invariant_all_transformers, matching in *

Expanding the definition of se_super_transformers,

Repeatedly Skolemizing and flattening,

Replacing using formula -3,

Keeping (-2 1) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of random_side_effects_side_effect_content_unchanged_write.1.3.3.1.

random_side_effects_side_effect_content_unchanged_write.1.3.3.2:

{-1}	size(uidt(dt_uint)) > 0
{1}	side_effect_content_unchanged(pm' rw_addr, pm' states, write_rnd_dev)
{2}	side_effect_content_unchanged(pm' rw_addr, pm' states, $\lambda (a: \text{Address}, bl: \text{list}[\text{Byte}], cp: \text{bool}):$ $\text{write_rnd_dev}(a, bl, cp) \#\#$ $(\lambda (bl: \text{list}[\text{Byte}]): \text{reset_count}(a, bl, cp))$)

Hiding formulas: 2,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of random_side_effects_side_effect_content_unchanged_write.1.3.3.2.

random_side_effects_side_effect_content_unchanged_write.1.3.3.3:

{-1}	size(uidt(dt_uint)) > 0
{1}	side_effect_content_unchanged(pm' rw_addr, pm' states, reset_count)
{2}	side_effect_content_unchanged(pm' rw_addr, pm' states, $\lambda (a: \text{Address}, bl: \text{list}[\text{Byte}], cp: \text{bool}):$ $\text{write_rnd_dev}(a, bl, cp) \#\#$ $(\lambda (bl: \text{list}[\text{Byte}]): \text{reset_count}(a, bl, cp))$)

Hiding formulas: 2,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of random_side_effects_side_effect_content_unchanged_write.1.3.3.3.

random_side_effects_side_effect_content_unchanged_write.2:

{1}	size(uidt(dt_uint)) > 0
{2}	side_effect_content_unchanged(pm' rw_addr, pm' states, rnd_dev_write_side_effect)

Using lemma dt_uint_size,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of random_side_effects_side_effect_content_unchanged_write.2.

Q.E.D.

C.23.8 Challenge_Random_Device.random_device_plain_memory

Terse proof for random_device_plain_memory.

random_device_plain_memory:

{1} plain_memory?(pm_phy) \supset plain_memory?(pm)

Using lemma device_memory_plain_memory,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 Expanding the definition of is_device_plain_memory?,
 Keeping (1) and hiding *,
 Rewriting using random_side_effects_transformers_ok, matching in *,
 Rewriting using random_side_effects_unchanged_invariant, matching in *,
 Rewriting using random_side_effects_side_effect_content_unchanged_read, matching in *,
 Rewriting using random_side_effects_side_effect_content_unchanged_write, matching in *,
 Expanding the definition of pm,
 Expanding the definition of random_device_pm,
 Installing automatic rewrites from: subset_reflexive difference_subset
 Expanding the definition of random_device_pm_mem,
 Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of random_device_plain_memory.
 Q.E.D.

C.23.9 Challenge_Random_Device.unaligned_fails_write

Terse proof for unaligned_fails_write.

unaligned_fails_write:

{1} $\forall (s: \text{Random_device_memory}, a: \text{Address}, \text{bl}: (\text{cons?} [\text{Byte}])):$
 $\neg \text{disjoint?}(\text{address_block}(a, \text{length}(\text{bl})), \text{address_block}(\text{base}, \text{size})) \wedge$
 $\neg (\text{length}(\text{bl}) = \text{size}(\text{uidt}(\text{dt_uint})) \wedge$
 $(a = \text{rnd_seed} \vee a = \text{mem_cnt} \vee a = \text{rnd_val}))$
 \wedge
 $\text{OK?}(\text{memory_write_side_effect}(\text{pm_phy}'\text{mem})(a, \text{bl}, \text{FALSE})(s'\text{state})) \wedge$
 $\text{data}(\text{memory_write_side_effect}(\text{pm_phy}'\text{mem})(a, \text{bl}, \text{FALSE})(s'\text{state})) = \text{bl}$
 $\supset \text{Fatal?}(\text{memory_write_list}(\text{pm}'\text{mem})(a, \text{bl})(s))$

Repeatedly Skolemizing and flattening,
 Expanding the definition of memory_write_list,
 Expanding the definition of pm,
 Expanding the definition of random_device_pm,
 Expanding the definition of random_device_pm_mem,
 Expanding the definition of device_pm,
 Expanding the definition of ##,
 Expanding the definition of em_lift,
 Simplifying, rewriting, and recording with decision procedures,
 Expanding the definition of rnd_dev_write_side_effect,
 Expanding the definition of unaligned_access,
 Expanding the definition of fatal_result,
 Expanding the definition of ##,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 This completes the proof of unaligned_fails_write.
 Q.E.D.

C.23.10 Challenge_Random_Device.unaligned_fails_read

Terse proof for unaligned_fails_read.

unaligned_fails_read:

<pre> {1} ∀ (s: Random_device_memory, a: Address, bl: (cons?[Byte])): ¬ disjoint?(address_block(a, length(bl)), address_block(base, size)) ∧ ¬ (length(bl) = size(uidt(dt_uint)) ∧ (a = rnd_seed ∨ a = mem_cnt ∨ a = rnd_val)) ∧ OK?(memory_read_list_nse(pm' mem)(a, length(bl))(s)) ∧ (LET ures = (memory_read_list_nse(pm' mem)(a, length(bl)) ## (λ (bl1: list[Byte]): em_lift [Physical_memory, Random_device_state[Physical_memory, pm_phy], list[Byte]] (memory_read_side_effect(pm_phy' mem)(a, bl1, FALSE)))) (s) IN OK?(ures) ∧ length(data(ures)) = length(bl)) ⊃ Fatal?(memory_read_list(pm' mem)(a, length(bl))(s) </pre>

Repeatedly Skolemizing and flattening,

Simplifying, rewriting, and recording with decision procedures,

Expanding the definition of memory_read_list,

Expanding the definition of pm,

Expanding the definition of random_device_pm,

Expanding the definition of random_device_pm_mem,

Expanding the definition of device_pm,

Expanding the definition of rnd_dev_read_side_effect,

Expanding the definition of unaligned_access,

Expanding the definition of ##,

Applying disjunctive simplification to flatten sequent,

Simplifying, rewriting, and recording with decision procedures,

Case splitting on $\text{cons?}(\text{data}(\text{em_lift} \text{ [Physical_memory, Random_device_state[Physical_memory, pm_phy], list[Byte]]} (\text{memory_read_side_effect}(\text{pm_phy' mem}) (\text{a!1}, \text{data}(\text{memory_read_list_nse}(\text{pm' mem}) (\text{a!1}, \text{length}(\text{bl!1})) (\text{s!1})), \text{FALSE})) (\text{state}(\text{memory_read_list_nse}(\text{pm' mem}) (\text{a!1}, \text{length}(\text{bl!1})) (\text{s!1}))))))$,

we get 2 subgoals:

unaligned_fails_read.1:

```

{-1} cons?(data(em_lift
              [Physical_memory, Random_device_state[Physical_memory, pm_phy], list[Byte]
              (memory_read_side_effect(pm_phy 'mem)
                (a', data(memory_read_list_nse(pm 'mem)(a', length(bl'))(s')),
                  FALSE))
              (state(memory_read_list_nse(pm 'mem)(a', length(bl'))(s')))))
{-2} every(λ (x: number):
           number_field_pred(x) ∧
           real_pred(x) ∧
           rational_pred(x) ∧ integer_pred(x) ∧ x ≥ 0 ∧ x < max_byte)
           (bl')
{-3} cons?[Byte](bl')
{-4} OK?(memory_read_list_nse(pm 'mem)(a', length(bl'))(s'))
{-5} OK?(em_lift[Physical_memory, Random_device_state[Physical_memory, pm_phy], list[Byte]]
        (memory_read_side_effect(pm_phy 'mem)
          (a', data(memory_read_list_nse(pm 'mem)(a', length(bl'))(s')),
            FALSE))
        (state(memory_read_list_nse(pm 'mem)(a', length(bl'))(s')))))
{-6} length(data(em_lift
                 [Physical_memory, Random_device_state[Physical_memory, pm_phy], list[Byte]
                 (memory_read_side_effect(pm_phy 'mem)
                   (a',
                     data(memory_read_list_nse(pm 'mem)(a', length(bl'))(s')),
                     FALSE))
                 (state(memory_read_list_nse(pm 'mem)(a', length(bl'))(s')))))
        = length(bl')
-----
{1} disjoint?(address_block(a', length(bl')), address_block(base, size))
{2} (length(bl') = size(uidt(dt_uint)) ∧
     (a' = rnd_seed ∨ a' = mem_cnt ∨ a' = rnd_val))
{3} Fatal?(CASES IF null?(data(em_lift
                               [Physical_memory,
                               Random_device_state[Physical_memory, pm_phy], list[Byte]
                               (memory_read_side_effect(pm_phy 'mem)
                                 (a',
                                   data(memory_read_list_nse(pm 'mem)
                                     (a', length(bl'))
                                     (s')),
                                   FALSE))
                               (state(memory_read_list_nse(pm 'mem)
                                     (a', length(bl'))
                                     (s')))))
                               ∨
                               length(data(em_lift
                                             [Physical_memory,
                                             Random_device_state[Physical_memory, pm_phy], list[Byte]
                                             (memory_read_side_effect(pm_phy 'mem)
                                               (a',
                                                 data(memory_read_list_nse(pm 'mem)
                                                   (a', length(bl'))
                                                   (s')),
                                                 FALSE))
                                             (state(memory_read_list_nse(pm 'mem)
                                                   (a', length(bl'))
                                                   (s')))))
                                             = size(uidt(dt_uint))
                                             ∧ (a' = rnd_seed ∨ a' = mem_cnt ∨ a' = rnd_val)
                               THEN ok_result(data(em_lift
                                                    [Physical_memory,
                                                    Random_device_state[Physical_memory, pm_phy], list[Byte]
                                                    (memory_read_side_effect(pm_phy 'mem)
                                                      (a',
                                                        data(memory_read_list_nse(pm 'mem)
                                                          (a', length(bl'))
                                                          (s')),
                                                        FALSE))
                                                    (state(memory_read_list_nse(pm 'mem)
                                                          (a', length(bl'))
                                                          (s')))))

```

Expanding the definition of `fatal_result`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `unaligned_fails_read.1`.

unaligned_fails_read.2:

```

{-1} every(λ (x: number):
      number_field_pred(x) ∧
      real_pred(x) ∧
      rational_pred(x) ∧ integer_pred(x) ∧ x ≥ 0 ∧ x < max_byte)
      (bl')
{-2} cons?[Byte](bl')
{-3} OK?(memory_read_list_nse(pm' mem)(a', length(bl'))(s'))
{-4} OK?(em_lift[Physical_memory, Random_device_state[Physical_memory, pm_phy], list[Byte]]
      (memory_read_side_effect(pm_phy' mem)
       (a', data(memory_read_list_nse(pm' mem)(a', length(bl'))(s')),
        FALSE))
      (state(memory_read_list_nse(pm' mem)(a', length(bl'))(s'))))
{-5} length(data(em_lift
      [Physical_memory, Random_device_state[Physical_memory, pm_phy], list[Byte]]
      (memory_read_side_effect(pm_phy' mem)
       (a',
        data(memory_read_list_nse(pm' mem)(a', length(bl'))(s')),
        FALSE))
      (state(memory_read_list_nse(pm' mem)(a', length(bl'))(s')))))
      = length(bl')
-----
{1} cons?(data(em_lift
      [Physical_memory, Random_device_state[Physical_memory, pm_phy], list[Byte]]
      (memory_read_side_effect(pm_phy' mem)
       (a', data(memory_read_list_nse(pm' mem)(a', length(bl'))(s')),
        FALSE))
      (state(memory_read_list_nse(pm' mem)(a', length(bl'))(s'))))
      disjoint?(address_block(a', length(bl')), address_block(base, size))
{2} (length(bl') = size(uidt(dt_uint)) ∧
{3} (a' = rnd_seed ∨ a' = mem_cnt ∨ a' = rnd_val))
{4} Fatal?(CASES IF null?(data(em_lift
      [Physical_memory,
      Random_device_state[Physical_memory, pm_phy], list[Byte]]
      (memory_read_side_effect(pm_phy' mem)
       (a',
        data(memory_read_list_nse(pm' mem)
          (a', length(bl'))
          (s')),
        FALSE))
      (state(memory_read_list_nse(pm' mem)
          (a', length(bl'))
          (s')))))
      ∨
      length(data(em_lift
      [Physical_memory,
      Random_device_state[Physical_memory, pm_phy], list[Byte]]
      (memory_read_side_effect(pm_phy' mem)
       (a',
        data(memory_read_list_nse(pm' mem)
          (a', length(bl'))
          (s')),
        FALSE))
      (state(memory_read_list_nse(pm' mem)
          (a', length(bl'))
          (s')))))
      = size(uidt(dt_uint))
      ∧ (a' = rnd_seed ∨ a' = mem_cnt ∨ a' = rnd_val)
      THEN ok_result(data(em_lift
      [Physical_memory,
      Random_device_state[Physical_memory, pm_phy], list[Byte]]
      (memory_read_side_effect(pm_phy' mem)
       (a',
        data(memory_read_list_nse(pm' mem)
          (a', length(bl'))
          (s')),
        FALSE))
      (state(memory_read_list_nse(pm' mem)
          (a', length(bl'))
          (s')))))

```

Keeping (-2 -5 1) and hiding *,
Expanding the definition of length,
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `unaligned_fails_read.2`.
Q.E.D.

C.24 Proofs for CommaExpressions (expressions.pvs)

This theory contains no provable formal statements.

C.25 Proofs for Composition_Lifted_Statement_Lifted_Expression (state-transformer.pvs)

This theory contains no provable formal statements.

C.26 Proofs for Composition_Lifted_Statement_Lifted_Statement (state-transformer.pvs)

This theory contains no provable formal statements.

C.27 Proofs for Composition_Lifted_Statement_Simple_Expression (state-transformer.pvs)

This theory contains no provable formal statements.

C.28 Proofs for Composition_Lifted_Statement_Simple_Statement (state-transformer.pvs)

This theory contains no provable formal statements.

C.29 Proofs for Composition_Simple_Expression_Data_Expression (state-transformer.pvs)

C.29.1 Composition_Simple_Expression_Data_Expression.super_comp_expr_expr_TCC1

Terse proof for `super_comp_expr_expr_TCC1`.

`super_comp_expr_expr_TCC1`:

```
{1}  ∃ (expr: [State → ExprResult[State, Data1]], s, state: State):
      expr_2_super(expr)(s) = OK(state) ⊃
      OK?[State](expr_2_super[State, Data1](expr)(s)) ∨
      abnormal?[State](expr_2_super[State, Data1](expr)(s))
```

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `super_comp_expr_expr_TCC1`.
Q.E.D.

C.29.2 Composition_Simple_Expression_Data_Expression.super_comp_expr_expr_TCC2

Terse proof for `super_comp_expr_expr_TCC2`.

`super_comp_expr_expr_TCC2`:

```
{1}  ∃ (expr: [State → ExprResult[State, Data1]], s, state: State):
      expr_2_super(expr)(s) = OK(state) ⊃ OK?[State, Data1](expr(s))
```

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `super_comp_expr_expr_TCC2`.
Q.E.D.

C.29.3 Composition_Simple_Expression_Data_Expression.super_comp_expr_expr

Terse proof for `super_comp_expr_expr`.

`super_comp_expr_expr`:

```
{1}  ∃ (expr: [State → ExprResult[State, Data1]],
      fexpr: [Data1 → [State → ExprResult[State, Data2]]], s: State):
      expr_2_super(expr ## fexpr)(s) =
      CASES expr_2_super(expr)(s) OF
      OK(state):
        expr_2_super(fexpr(data(expr(s))))(state(expr_2_super(expr)(s)))
      ELSE expr_2_super(expr)(s)
      ENDCASES
```

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `super_comp_expr_expr`.

Q.E.D.

C.30 Proofs for Composition_Simple_Expression_Data_Statement (state-transformer.pvs)

This theory contains no provable formal statements.

C.31 Proofs for Composition_Simple_Expression_Simple_Expression (state-transformer.pvs)

C.31.1 Composi- tion_Simple_Expression_Simple_Expression.comp_expr_forget_expr

Terse proof for comp_expr_forget_expr.

comp_expr_forget_expr:

```
{1}  ∃ (expr1: [State → ExprResult[State, Data1]],
      expr2: [State → ExprResult[State, Data2]], s: State):
      (expr1 ## expr2)(s) =
      CASES expr1(s) OF
        OK(state, data): expr2(state),
        Exception(ex_type, state): Exception(ex_type, state),
        Fatal: Fatal,
        Hang: Hang
      ENDCASES
```

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of comp_expr_forget_expr.

Q.E.D.

C.31.2 Composi- tion_Simple_Expression_Simple_Expression.super_comp_expr_forget_expr

Terse proof for super_comp_expr_forget_expr.

super_comp_expr_forget_expr:

```
{1}  ∃ (expr1: [State → ExprResult[State, Data1]],
      expr2: [State → ExprResult[State, Data2]], s: State):
      expr_2_super(expr1 ## expr2)(s) =
      CASES expr_2_super(expr1)(s) OF OK(state): expr_2_super(expr2)(state)
      ELSE expr_2_super(expr1)(s)
      ENDCASES
```

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `super_comp_expr_forget_expr`.
Q.E.D.

C.31.3 Composition_Simple_Expression_Simple_Expression.comp_simple_expr_simple_expr_TCC1

Terse proof for `comp_simple_expr_simple_expr_TCC1`.

`comp_simple_expr_simple_expr_TCC1`:

$$\frac{\{1\} \quad \forall (\text{expr1}: [\text{State} \rightarrow \text{ExprResult}[\text{State}, \text{Data1}]], s: \text{State}): \quad \text{OK?}(\text{expr1}(s)) \supset \quad \text{OK?}[\text{State}, \text{Data1}](\text{expr1}(s)) \vee \text{Exception?}[\text{State}, \text{Data1}](\text{expr1}(s))}{}$$

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `comp_simple_expr_simple_expr_TCC1`.
Q.E.D.

C.31.4 Composition_Simple_Expression_Simple_Expression.comp_simple_expr_simple_expr

Terse proof for `comp_simple_expr_simple_expr`.

`comp_simple_expr_simple_expr`:

$$\frac{\{1\} \quad \forall (\text{expr1}: [\text{State} \rightarrow \text{ExprResult}[\text{State}, \text{Data1}]], \quad \text{expr2}: [\text{State} \rightarrow \text{ExprResult}[\text{State}, \text{Data2}]], s: \text{State}): \quad \text{OK?}(\text{expr1}(s)) \supset \text{expr2}(\text{state}(\text{expr1}(s))) = (\text{expr1} \#\# \text{expr2})(s)}{}$$

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `comp_simple_expr_simple_expr`.
Q.E.D.

C.31.5 Composition_Simple_Expression_Simple_Expression.comp_simple_expr_simple_expr_ok_TCC1

Terse proof for `comp_simple_expr_simple_expr_ok_TCC1`.

`comp_simple_expr_simple_expr_ok_TCC1`:

$$\frac{\{1\} \quad \forall (\text{expr1}: [\text{State} \rightarrow \text{ExprResult}[\text{State}, \text{Data1}]], \quad \text{expr2}: [\text{State} \rightarrow \text{ExprResult}[\text{State}, \text{Data2}]], s: \text{State}): \quad \text{OK?}((\text{expr1} \#\# \text{expr2})(s)) \supset \quad \text{OK?}[\text{State}, \text{Data1}](\text{expr1}(s)) \vee \text{Exception?}[\text{State}, \text{Data1}](\text{expr1}(s))}{}$$

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `comp_simple_expr_simple_expr_ok_TCC1`.
Q.E.D.

C.31.6 Composition_Simple_Expression_Simple_Expression.comp_simple_expr_simple_expr_ok

Terse proof for comp_simple_expr_simple_expr_ok.

comp_simple_expr_simple_expr_ok:

$$\frac{\{1\} \quad \forall (\text{expr1}: [\text{State} \rightarrow \text{ExprResult}[\text{State}, \text{Data1}]], \text{expr2}: [\text{State} \rightarrow \text{ExprResult}[\text{State}, \text{Data2}]], s: \text{State}): \text{OK?}((\text{expr1} \#\# \text{expr2})(s)) \supset \text{OK?}(\text{expr2}(\text{state}(\text{expr1}(s))))}{}$$

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of comp_simple_expr_simple_expr_ok.
Q.E.D.

C.31.7 Composition_Simple_Expression_Simple_Expression.comp_simple_expr_simple_expr_data_TCC1

Terse proof for comp_simple_expr_simple_expr_data_TCC1.

comp_simple_expr_simple_expr_data_TCC1:

$$\frac{\{1\} \quad \forall (\text{expr1}: [\text{State} \rightarrow \text{ExprResult}[\text{State}, \text{Data1}]], \text{expr2}: [\text{State} \rightarrow \text{ExprResult}[\text{State}, \text{Data2}]], s: \text{State}): \text{OK?}((\text{expr1} \#\# \text{expr2})(s)) \supset \text{OK?}[\text{State}, \text{Data2}](\text{expr2}(\text{state}[\text{State}, \text{Data1}](\text{expr1}(s))))}{}$$

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of comp_simple_expr_simple_expr_data_TCC1.
Q.E.D.

C.31.8 Composition_Simple_Expression_Simple_Expression.comp_simple_expr_simple_expr_data

Terse proof for comp_simple_expr_simple_expr_data.

comp_simple_expr_simple_expr_data:

$$\frac{\{1\} \quad \forall (\text{expr1}: [\text{State} \rightarrow \text{ExprResult}[\text{State}, \text{Data1}]], \text{expr2}: [\text{State} \rightarrow \text{ExprResult}[\text{State}, \text{Data2}]], s: \text{State}): \text{OK?}((\text{expr1} \#\# \text{expr2})(s)) \supset \text{data}(\text{expr2}(\text{state}(\text{expr1}(s)))) = \text{data}((\text{expr1} \#\# \text{expr2})(s))}{}$$

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of comp_simple_expr_simple_expr_data.
Q.E.D.

C.31.9 Composition_Simple_Expression_Simple_Expression.comp_simple_expr_simple_expr_state_TCC1

Terse proof for comp_simple_expr_simple_expr_state_TCC1.

comp_simple_expr_simple_expr_state_TCC1:

$$\{1\} \quad \forall (\text{expr1}: [\text{State} \rightarrow \text{ExprResult}[\text{State}, \text{Data1}]], \\ \text{expr2}: [\text{State} \rightarrow \text{ExprResult}[\text{State}, \text{Data2}]], s: \text{State}): \\ \text{OK}((\text{expr1} \## \text{expr2})(s)) \supset \\ \text{OK}[\text{State}, \text{Data2}]((\text{expr1} \## \text{expr2})(s)) \vee \\ \text{Exception}[\text{State}, \text{Data2}]((\text{expr1} \## \text{expr2})(s))$$

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of comp_simple_expr_simple_expr_state_TCC1.

Q.E.D.

C.31.10 Composition_Simple_Expression_Simple_Expression.comp_simple_expr_simple_expr_state

Terse proof for comp_simple_expr_simple_expr_state.

comp_simple_expr_simple_expr_state:

$$\{1\} \quad \forall (\text{expr1}: [\text{State} \rightarrow \text{ExprResult}[\text{State}, \text{Data1}]], \\ \text{expr2}: [\text{State} \rightarrow \text{ExprResult}[\text{State}, \text{Data2}]], s: \text{State}): \\ \text{OK}((\text{expr1} \## \text{expr2})(s)) \supset \\ \text{state}(\text{expr2}(\text{state}(\text{expr1}(s)))) = \text{state}((\text{expr1} \## \text{expr2})(s))$$

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of comp_simple_expr_simple_expr_state.

Q.E.D.

C.32 Proofs for Composition_Simple_Statement_Lifted_Expression (state-transformer.pvs)

This theory contains no provable formal statements.

C.33 Proofs for Composition_Simple_Statement_Lifted_Statement (state-transformer.pvs)

This theory contains no provable formal statements.

C.34 Proofs for Composition_Simple_Statement_Simple_Expression (state-transformer.pvs)

C.34.1 Composi- tion_Simple_Statement_Simple_Expression.comp_simple_stmt_simple_expr_TCC1

Terse proof for comp_simple_stmt_simple_expr_TCC1.

comp_simple_stmt_simple_expr_TCC1:

$\{1\} \quad \forall (s: \text{State}, \text{stmt}: [\text{State} \rightarrow \text{StmtResult}[\text{State}]]):$ $\text{OK?}(\text{stmt}(s)) \supset$ $\text{OK?}[\text{State}](\text{stmt}(s)) \vee$ $\text{Break?}[\text{State}](\text{stmt}(s)) \vee$ $\text{Continue?}[\text{State}](\text{stmt}(s)) \vee$ $\text{Return?}[\text{State}](\text{stmt}(s)) \vee$ $\text{Switch?}[\text{State}](\text{stmt}(s)) \vee$ $\text{Default?}[\text{State}](\text{stmt}(s)) \vee \text{Exception?}[\text{State}](\text{stmt}(s))$
--

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of comp_simple_stmt_simple_expr_TCC1.

Q.E.D.

C.34.2 Composi- tion_Simple_Statement_Simple_Expression.comp_simple_stmt_simple_expr

Terse proof for comp_simple_stmt_simple_expr.

comp_simple_stmt_simple_expr:

$\{1\} \quad \forall (\text{expr}: [\text{State} \rightarrow \text{ExprResult}[\text{State}, \text{Data}]], s: \text{State},$ $\text{stmt}: [\text{State} \rightarrow \text{StmtResult}[\text{State}]]):$ $\text{OK?}(\text{stmt}(s)) \supset \text{expr}(\text{state}(\text{stmt}(s))) = (\text{stmt} \#\# \text{expr})(s)$
--

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of comp_simple_stmt_simple_expr.

Q.E.D.

C.34.3 Composi- tion_Simple_Statement_Simple_Expression.comp_simple_stmt_simple_expr_ok_TCC1

Terse proof for comp_simple_stmt_simple_expr_ok_TCC1.

comp_simple_stmt_simple_expr_ok_TCC1:

$$\{1\} \quad \forall (\text{expr}: [\text{State} \rightarrow \text{ExprResult}[\text{State}, \text{Data}]], s: \text{State}, \\ \text{stmt}: [\text{State} \rightarrow \text{StmtResult}[\text{State}]]) : \\ \text{OK}((\text{stmt} \ \#\# \ \text{expr})(s)) \supset \\ \text{OK}[\text{State}](\text{stmt}(s)) \vee \\ \text{Break}[\text{State}](\text{stmt}(s)) \vee \\ \text{Continue}[\text{State}](\text{stmt}(s)) \vee \\ \text{Return}[\text{State}](\text{stmt}(s)) \vee \\ \text{Switch}[\text{State}](\text{stmt}(s)) \vee \\ \text{Default}[\text{State}](\text{stmt}(s)) \vee \text{Exception}[\text{State}](\text{stmt}(s)))$$

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of comp_simple_stmt_simple_expr_ok_TCC1.
Q.E.D.

C.34.4 Composition_Simple_Statement_Simple_Expression.comp_simple_stmt_simple_expr_ok

Terse proof for comp_simple_stmt_simple_expr_ok.

comp_simple_stmt_simple_expr_ok:

$$\{1\} \quad \forall (\text{expr}: [\text{State} \rightarrow \text{ExprResult}[\text{State}, \text{Data}]], s: \text{State}, \\ \text{stmt}: [\text{State} \rightarrow \text{StmtResult}[\text{State}]]) : \\ \text{OK}((\text{stmt} \ \#\# \ \text{expr})(s)) \supset \text{OK}(\text{expr}(\text{state}(\text{stmt}(s))))$$

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of comp_simple_stmt_simple_expr_ok.
Q.E.D.

C.34.5 Composition_Simple_Statement_Simple_Expression.comp_simple_stmt_simple_expr_data

Terse proof for comp_simple_stmt_simple_expr_data_TCC1.

comp_simple_stmt_simple_expr_data_TCC1:

$$\{1\} \quad \forall (\text{expr}: [\text{State} \rightarrow \text{ExprResult}[\text{State}, \text{Data}]], s: \text{State}, \\ \text{stmt}: [\text{State} \rightarrow \text{StmtResult}[\text{State}]]) : \\ \text{OK}((\text{stmt} \ \#\# \ \text{expr})(s)) \supset \text{OK}[\text{State}, \text{Data}](\text{expr}(\text{state}[\text{State}](\text{stmt}(s))))$$

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of comp_simple_stmt_simple_expr_data_TCC1.
Q.E.D.

C.34.6 Composition_Simple_Statement_Simple_Expression.comp_simple_stmt_simple_expr_data

Terse proof for comp_simple_stmt_simple_expr_data.

comp_simple_stmt_simple_expr_data:

$\{1\} \quad \forall (\text{expr}: [\text{State} \rightarrow \text{ExprResult}[\text{State}, \text{Data}]], s: \text{State}, \\ \text{stmt}: [\text{State} \rightarrow \text{StmtResult}[\text{State}]]):$ $\text{OK?}((\text{stmt} \ \#\# \ \text{expr})(s)) \supset \\ \text{data}(\text{expr}(\text{state}(\text{stmt}(s)))) = \text{data}((\text{stmt} \ \#\# \ \text{expr})(s))$

Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of comp_simple_stmt_simple_expr_data.
 Q.E.D.

C.34.7 Composition_Simple_Statement_Simple_Expression.comp_simple_stmt_simple_expr_state_TCC1

Terse proof for comp_simple_stmt_simple_expr_state_TCC1.

comp_simple_stmt_simple_expr_state_TCC1:

$\{1\} \quad \forall (\text{expr}: [\text{State} \rightarrow \text{ExprResult}[\text{State}, \text{Data}]], s: \text{State}, \\ \text{stmt}: [\text{State} \rightarrow \text{StmtResult}[\text{State}]]):$ $\text{OK?}((\text{stmt} \ \#\# \ \text{expr})(s)) \supset \\ \text{OK?}[\text{State}, \text{Data}]((\text{stmt} \ \#\# \ \text{expr})(s)) \vee \\ \text{Exception?}[\text{State}, \text{Data}]((\text{stmt} \ \#\# \ \text{expr})(s))$
--

Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of comp_simple_stmt_simple_expr_state_TCC1.
 Q.E.D.

C.34.8 Composition_Simple_Statement_Simple_Expression.comp_simple_stmt_simple_expr_state

Terse proof for comp_simple_stmt_simple_expr_state.

comp_simple_stmt_simple_expr_state:

$\{1\} \quad \forall (\text{expr}: [\text{State} \rightarrow \text{ExprResult}[\text{State}, \text{Data}]], s: \text{State}, \\ \text{stmt}: [\text{State} \rightarrow \text{StmtResult}[\text{State}]]):$ $\text{OK?}((\text{stmt} \ \#\# \ \text{expr})(s)) \supset \\ \text{state}(\text{expr}(\text{state}(\text{stmt}(s)))) = \text{state}((\text{stmt} \ \#\# \ \text{expr})(s))$

Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of comp_simple_stmt_simple_expr_state.
 Q.E.D.

C.35 Proofs for Composition_Simple_Statement_Simple_Statement (state-transformer.pvs)

This theory contains no provable formal statements.

C.36 Proofs for ConditionalExpression (expressions.pvs)

This theory contains no provable formal statements.

C.37 Proofs for Control_Register_Datatype (paging-data.pvs)

C.37.1 Control_Register_Datatype.cr0_data_type_TCC1

Terse proof for cr0_data_type_TCC1.

cr0_data_type_TCC1:

$$\{1\} \quad \exists (x: (\text{pod_data_type?}[CR0_type])): \text{TRUE}$$

Rewriting using cr0_data_type_exists, matching in *,

This completes the proof of cr0_data_type_TCC1.

Q.E.D.

C.37.2 Control_Register_Datatype.cr2_data_type_TCC1

Terse proof for cr2_data_type_TCC1.

cr2_data_type_TCC1:

$$\{1\} \quad \exists (x_1: (\text{pod_data_type?}[CR2_type])): \text{TRUE}$$

Rewriting using cr2_data_type_exists, matching in *,

This completes the proof of cr2_data_type_TCC1.

Q.E.D.

C.37.3 Control_Register_Datatype.cr4_data_type_TCC1

Terse proof for cr4_data_type_TCC1.

cr4_data_type_TCC1:

$$\{1\} \quad \exists (x: (\text{pod_data_type?}[CR4_type])): \text{TRUE}$$

Rewriting using cr4_data_type_exists, matching in *,

This completes the proof of cr4_data_type_TCC1.

Q.E.D.

C.38 Proofs for Control_Register_Types (paging-data.pvs)

This theory contains no provable formal statements.

C.39 Proofs for Conversions (conversions.pvs)

This theory contains no provable formal statements.

C.40 Proofs for Cpp_Deep_Type_Wellformedness (types.pvs)

C.40.1

Cpp_Deep_Type_Wellformedness.no_pointers_to_bitfield?_TCC1

Terse proof for no_pointers_to_bitfield?_TCC1.

no_pointers_to_bitfield?_TCC1:

<pre>{1} ∀ (t: Cpp_Type_): pointer?(t) ⊃ array?(t) ∨ pointer?(t) ∨ reference?(t) ∨ bitfield?(t) ∨ enum?(t) ∨ pointer_to_member?(t) ∨ const?(t) ∨ volatile?(t)</pre>
--

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of no_pointers_to_bitfield?_TCC1.
Q.E.D.

C.40.2 Cpp_Deep_Type_Wellformedness.no_cv_references?_TCC1

Terse proof for no_cv_references?_TCC1.

no_cv_references?_TCC1:

<pre>{1} ∀ (t: Cpp_Type_): (const?(t) ∨ volatile?(t)) ⊃ array?(t) ∨ pointer?(t) ∨ reference?(t) ∨ bitfield?(t) ∨ enum?(t) ∨ pointer_to_member?(t) ∨ const?(t) ∨ volatile?(t)</pre>

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of no_cv_references?_TCC1.
Q.E.D.

C.40.3

Cpp_Deep_Type_Wellformedness.no_reference_to_reference?_TCC1

Terse proof for no_reference_to_reference?_TCC1.

no_reference_to_reference?_TCC1:

```
{1}  ∀ (t: Cpp_Type_):
      reference?(t) ⊃
      array?(t) ∨
      pointer?(t) ∨
      reference?(t) ∨
      bitfield?(t) ∨ enum?(t) ∨ pointer_to_member?(t) ∨ const?(t) ∨ volatile?(t)
```

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of no_reference_to_reference?_TCC1.
Q.E.D.

C.40.4

Cpp_Deep_Type_Wellformedness.no_array_of_references?_TCC1

Terse proof for no_array_of_references?_TCC1.

no_array_of_references?_TCC1:

```
{1}  ∀ (t: Cpp_Type_):
      array?(t) ⊃
      array?(t) ∨
      pointer?(t) ∨
      reference?(t) ∨
      bitfield?(t) ∨ enum?(t) ∨ pointer_to_member?(t) ∨ const?(t) ∨ volatile?(t)
```

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of no_array_of_references?_TCC1.
Q.E.D.

C.40.5

Cpp_Deep_Type_Wellformedness.no_pointer_or_ref_to_incomplete_array_parameter?_TCC1

Terse proof for no_pointer_or_ref_to_incomplete_array_parameter?_TCC1.

no_pointer_or_ref_to_incomplete_array_parameter?_TCC1:

```
{1}  ∀ (t: Cpp_Type_):
      function?(t) ⊃
      (∀ (p: Cpp_Type_):
        (pointer?(p) ∨ reference?(p)) ⊃
        array?(p) ∨
        pointer?(p) ∨
        reference?(p) ∨
        bitfield?(p) ∨
        enum?(p) ∨ pointer_to_member?(p) ∨ const?(p) ∨ volatile?(p))
```

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of no_pointer_or_ref_to_incomplete_array_parameter?_TCC1.
Q.E.D.

C.40.6**Cpp_Deep_Type_Wellformedness.no_pointer_or_ref_to_incomplete_array_parameter?**

Terse proof for no_pointer_or_ref_to_incomplete_array_parameter?_TCC2.

no_pointer_or_ref_to_incomplete_array_parameter?_TCC2:

```
{1}  ∀ (t: Cpp_Type_):
      function?(t) ⊃
      (∀ (p: Cpp_Type_):
        (const?(p) ∨ volatile?(p)) ∧
        ¬ ((pointer?(p) ∨ reference?(p)) ∧
           (array?(typ(p)) ∧ size(typ(p)) > 0))
        ⊃
        array?(p) ∨
        pointer?(p) ∨
        reference?(p) ∨
        bitfield?(p) ∨
        enum?(p) ∨ pointer_to_member?(p) ∨ const?(p) ∨ volatile?(p))
```

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of no_pointer_or_ref_to_incomplete_array_parameter?_TCC2.

Q.E.D.

C.40.7**Cpp_Deep_Type_Wellformedness.no_pointer_or_ref_to_incomplete_array_parameter?**

Terse proof for no_pointer_or_ref_to_incomplete_array_parameter?_TCC3.

no_pointer_or_ref_to_incomplete_array_parameter?_TCC3:

```
{1}  ∀ (t: Cpp_Type_):
      function?(t) ⊃
      (∀ (p: Cpp_Type_):
        pointer?(typ(p)) ∧
        (const?(p) ∨ volatile?(p)) ∧
        ¬ ((pointer?(p) ∨ reference?(p)) ∧
           (array?(typ(p)) ∧ size(typ(p)) > 0))
        ⊃
        array?(typ(p)) ∨
        pointer?(typ(p)) ∨
        reference?(typ(p)) ∨
        bitfield?(typ(p)) ∨
        enum?(typ(p)) ∨
        pointer_to_member?(typ(p)) ∨ const?(typ(p)) ∨ volatile?(typ(p)))
```

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of no_pointer_or_ref_to_incomplete_array_parameter?_TCC3.

Q.E.D.

C.40.8**Cpp_Deep_Type_Wellformedness.no_pointer_or_ref_to_incomplete_array_param**

Terse proof for `no_pointer_or_ref_to_incomplete_array_parameter?_TCC4`.

`no_pointer_or_ref_to_incomplete_array_parameter?_TCC4`:

```
{1}  ∀ (t: Cpp_Type_):
      function?(t) ⊃
      (∀ (p: Cpp_Type_):
        const?(p) ∧
        ¬ ((const?(p) ∨ volatile?(p)) ∧
           pointer?(typ(p)) ∧
           (array?(typ(typ(p))) ∧ size(typ(typ(p))) > 0))
        ∧
        ¬ ((pointer?(p) ∨ reference?(p)) ∧
           (array?(typ(p)) ∧ size(typ(p)) > 0))
        ⊃
        array?(p) ∨
        pointer?(p) ∨
        reference?(p) ∨
        bitfield?(p) ∨
        enum?(p) ∨ pointer_to_member?(p) ∨ const?(p) ∨ volatile?(p))
```

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `no_pointer_or_ref_to_incomplete_array_parameter?_TCC4`.

Q.E.D.

C.40.9**Cpp_Deep_Type_Wellformedness.no_pointer_or_ref_to_incomplete_array_param**

Terse proof for `no_pointer_or_ref_to_incomplete_array_parameter?_TCC5`.

`no_pointer_or_ref_to_incomplete_array_parameter?_TCC5`:

```
{1}  ∀ (t: Cpp_Type_):
      function?(t) ⊃
      (∀ (p: Cpp_Type_):
        volatile?(typ(p)) ∧
        const?(p) ∧
        ¬ ((const?(p) ∨ volatile?(p)) ∧
           pointer?(typ(p)) ∧
           (array?(typ(typ(p))) ∧ size(typ(typ(p))) > 0))
        ∧
        ¬ ((pointer?(p) ∨ reference?(p)) ∧
           (array?(typ(p)) ∧ size(typ(p)) > 0))
        ⊃
        array?(typ(p)) ∨
        pointer?(typ(p)) ∨
        reference?(typ(p)) ∨
        bitfield?(typ(p)) ∨
        enum?(typ(p)) ∨
        pointer_to_member?(typ(p)) ∨ const?(typ(p)) ∨ volatile?(typ(p))
```

Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of `no_pointer_or_ref_to_incomplete_array_parameter?_TCC5`.
 Q.E.D.

C.40.10

Cpp_Deep_Type_Wellformedness.no_pointer_or_ref_to_incomplete_array_parameter?

Terse proof for `no_pointer_or_ref_to_incomplete_array_parameter?_TCC6`.
`no_pointer_or_ref_to_incomplete_array_parameter?_TCC6`:

```
{1}  ∀ (t: Cpp_Type_):
      function?(t) ⊃
      (∀ (p: Cpp_Type_):
        pointer?(typ(typ(p))) ∧
        volatile?(typ(p)) ∧
        const?(p) ∧
        ¬ ((const?(p) ∨ volatile?(p)) ∧
           pointer?(typ(p)) ∧
           (array?(typ(typ(p))) ∧ size(typ(typ(p))) > 0))
        ∧
        ¬ ((pointer?(p) ∨ reference?(p)) ∧
           (array?(typ(p)) ∧ size(typ(p)) > 0))
      ) ⊃
      array?(typ(typ(p))) ∨
      pointer?(typ(typ(p))) ∨
      reference?(typ(typ(p))) ∨
      bitfield?(typ(typ(p))) ∨
      enum?(typ(typ(p))) ∨
      pointer_to_member?(typ(typ(p))) ∨
      const?(typ(typ(p))) ∨ volatile?(typ(typ(p)))
```

Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of `no_pointer_or_ref_to_incomplete_array_parameter?_TCC6`.
 Q.E.D.

C.40.11

Cpp_Deep_Type_Wellformedness.bitfield_underlying_integral_or_enum_type?_TCC1

Terse proof for `bitfield_underlying_integral_or_enum_type?_TCC1`.
`bitfield_underlying_integral_or_enum_type?_TCC1`:

```
{1}  ∀ (t: Cpp_Type_):
      bitfield?(t) ⊃
      array?(t) ∨
      pointer?(t) ∨
      reference?(t) ∨
      bitfield?(t) ∨ enum?(t) ∨ pointer_to_member?(t) ∨ const?(t) ∨ volatile?(t)
```

Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of `bitfield_underlying_integral_or_enum_type?_TCC1`.
 Q.E.D.

C.40.12

Cpp_Deep_Type_Wellformedness.enum_underlying_integral?_TCC1

Terse proof for enum_underlying_integral?_TCC1.

enum_underlying_integral?_TCC1:

```
{1}  ∀ (t: Cpp_Type_):
      enum?(t) ⊃
      array?(t) ∨
      pointer?(t) ∨
      reference?(t) ∨
      bitfield?(t) ∨ enum?(t) ∨ pointer_to_member?(t) ∨ const?(t) ∨ volatile?(t)
```

Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of enum_underlying_integral?_TCC1.
 Q.E.D.

C.40.13 Cpp_Deep_Type_Wellformedness.enum_constants?_TCC1

Terse proof for enum_constants?_TCC1.

enum_constants?_TCC1:

```
{1}  ∀ (e: (enum_underlying_integral?)):
      enum?(e) ⊃
      (∀ (m: below[members(e)]):
        array?(e) ∨
        pointer?(e) ∨
        reference?(e) ∨
        bitfield?(e) ∨
        enum?(e) ∨ pointer_to_member?(e) ∨ const?(e) ∨ volatile?(e))
```

Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of enum_constants?_TCC1.
 Q.E.D.

C.40.14 Cpp_Deep_Type_Wellformedness.enum_constants?_TCC2

Terse proof for enum_constants?_TCC2.

enum_constants?_TCC2:

```
{1}  ∀ (e: (enum_underlying_integral?)):
      enum?(e) ⊃ (∀ (m: below[members(e)]): non_boolIntegral?(typ(e)))
```

Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of enum_constants?_TCC2.
 Q.E.D.

C.40.15 Cpp_Deep_Type_Wellformedness.const_volatile?_TCC1

Terse proof for const_volatile?_TCC1.

const_volatile?_TCC1:

```
{1}  ∀ (t: Cpp_Type_):
      volatile?(t) ⊃
      array?(t) ∨
      pointer?(t) ∨
      reference?(t) ∨
      bitfield?(t) ∨ enum?(t) ∨ pointer_to_member?(t) ∨ const?(t) ∨ volatile?(t)
```

Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of const_volatile?_TCC1.
 Q.E.D.

C.40.16 Cpp_Deep_Type_Wellformedness.const_stutter?_TCC1

Terse proof for const_stutter?_TCC1.

const_stutter?_TCC1:

```
{1}  ∀ (t: Cpp_Type_):
      const?(t) ⊃
      array?(t) ∨
      pointer?(t) ∨
      reference?(t) ∨
      bitfield?(t) ∨ enum?(t) ∨ pointer_to_member?(t) ∨ const?(t) ∨ volatile?(t)
```

Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of const_stutter?_TCC1.
 Q.E.D.

C.40.17 Cpp_Deep_Type_Wellformedness.cv_array?_TCC1

Terse proof for cv_array?_TCC1.

cv_array?_TCC1:

```
{1}  ∀ (t: Cpp_Type_):
      (volatile?(t) ∨ const?(t)) ⊃
      array?(t) ∨
      pointer?(t) ∨
      reference?(t) ∨
      bitfield?(t) ∨ enum?(t) ∨ pointer_to_member?(t) ∨ const?(t) ∨ volatile?(t)
```

Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of cv_array?_TCC1.
 Q.E.D.

C.40.18 Cpp_Deep_Type_Wellformedness.cv_base_TCC1

Terse proof for cv_base_TCC1.

cv_base_TCC1:

{1} well_founded?($\lambda (x: \text{Cpp_Type}, y: \text{Cpp_Type}):$
 $\text{restrict}[[\text{Cpp_Type}_-, \text{Cpp_Type}_-], [\text{Cpp_Type}, \text{Cpp_Type}], \text{boolean}]$
 $(\ll)(x, y)$)

Using lemma well_founded_restrict[Cpp_Type_, Cpp_Type],
 Expanding the definition of restrict,
 which is trivially true.
 This completes the proof of cv_base_TCC1.
 Q.E.D.

C.40.19 Cpp_Deep_Type_Wellformedness.cv_base_TCC2

Terse proof for cv_base_TCC2.

cv_base_TCC2:

{1} $\forall (\text{typ}: \text{Cpp_Type}, t: \text{Cpp_Type}_-): \text{typ} = \text{const}(t) \supset \text{Cpp_Type?}(t)$

Repeatedly Skolemizing and flattening,
 Replacing using formula -2,
 Expanding the definition of Cpp_Type?,
 Repeatedly Skolemizing and flattening,
 Instantiating quantified variables,
 Expanding the definition of subterm,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 This completes the proof of cv_base_TCC2.
 Q.E.D.

C.40.20 Cpp_Deep_Type_Wellformedness.cv_base_TCC3

Terse proof for cv_base_TCC3.

cv_base_TCC3:

{1} $\forall (\text{typ}: \text{Cpp_Type}, t: \text{Cpp_Type}_-):$
 $\text{typ} = \text{const}(t) \supset$
 $\text{restrict}[[\text{Cpp_Type}_-, \text{Cpp_Type}_-], [\text{Cpp_Type}, \text{Cpp_Type}], \text{boolean}](\ll)(t, \text{typ})$

Expanding the definition of restrict,
 Repeatedly Skolemizing and flattening,
 Replacing using formula -2,
 Expanding the definition of ■,
 which is trivially true.
 This completes the proof of cv_base_TCC3.
 Q.E.D.

C.40.21 Cpp_Deep_Type_Wellformedness.cv_base_TCC4

Terse proof for cv_base_TCC4.

cv_base_TCC4:

{1} $\forall (\text{typ}: \text{Cpp_Type}, t: \text{Cpp_Type}_-): \text{typ} = \text{volatile}(t) \supset \text{Cpp_Type?}(t)$

Repeatedly Skolemizing and flattening,
 Replacing using formula -2,
 Expanding the definition of Cpp_Type?,
 Repeatedly Skolemizing and flattening,
 Instantiating quantified variables,
 Expanding the definition of subterm,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 This completes the proof of cv_base_TCC4.
 Q.E.D.

C.40.22 Cpp_Deep_Type_Wellformedness.cv_base_TCC5

Terse proof for cv_base_TCC5.

cv_base_TCC5:

$\{1\} \quad \forall (\text{typ}: \text{Cpp_Type}, t: \text{Cpp_Type_}):$ $\text{typ} = \text{volatile}(t) \supset$ $\text{restrict}[[\text{Cpp_Type_}, \text{Cpp_Type_}], [\text{Cpp_Type}, \text{Cpp_Type}], \text{boolean}](\ll)(t, \text{typ})$
--

Repeatedly Skolemizing and flattening,
 Expanding the definition of restrict,
 Replacing using formula -2,
 Expanding the definition of ■,
 which is trivially true.
 This completes the proof of cv_base_TCC5.
 Q.E.D.

C.40.23 Cpp_Deep_Type_Wellformedness.subterm_cv_base

Terse proof for subterm_cv_base.

subterm_cv_base:

$\{1\} \quad \forall (\text{typ}: \text{Cpp_Type}): \text{subterm}(\text{cv_base}(\text{typ}), \text{typ})$

Repeatedly Skolemizing and flattening,
 Expanding the definition of cv_base,
 Expanding the definition of cv_base,
 Lifting IF-conditions to the top level,
 Expanding the definition of subterm,
 Expanding the definition of subterm,
 Simplifying, rewriting, and recording with decision procedures,
 Expanding the definition of Cpp_Type?,
 Expanding the definition of const_volatile?,
 Expanding the definition of const_stutter?,
 Expanding the definition of volatile_stutter?,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 we get 5 subgoals:

subterm_cv_base.1:

{-1}	$\forall (t: \text{Cpp_Type_}):$ $\text{subterm}(t, \text{typ}') \supset$ $\text{no_pointers_to_bitfield?}(t) \wedge$ $\text{no_pointers_to_references?}(t) \wedge$ $\text{no_cv_references?}(t) \wedge$ $\text{no_reference_to_reference?}(t) \wedge$ $\text{no_reference_to_bitfields?}(t) \wedge$ $\text{no_pointer_to_member_to_reference?}(t) \wedge$ $\text{no_pointer_to_member_to_cv_void?}(t) \wedge$ $\text{no_cv_void_parameter?}(t) \wedge$ $\text{no_array_of_references?}(t) \wedge$ $\text{no_array_of_cv_void?}(t) \wedge$ $\text{no_array_of_function?}(t) \wedge$ $\text{no_array_of_abstract_class?}(t) \wedge$ $\text{no_pointer_or_ref_to_incomplete_array_parameter?}(t) \wedge$ $\text{no_array_return_type?}(t) \wedge$ $\text{no_function_return_type?}(t) \wedge$ $\text{bitfield_underlying_integral_or_enum_type?}(t) \wedge$ $\text{cv_array?}(t) \wedge$ $\text{enum_underlying_integral?}(t) \wedge$ $\text{enum_constants?}(t) \wedge$ $(\text{volatile?}(t) \supset \neg \text{const?}(\text{typ}(t))) \wedge$ $(\text{const?}(t) \supset \neg \text{const?}(\text{typ}(t))) \wedge$ $(\text{volatile?}(t) \supset \neg \text{volatile?}(\text{typ}(t))) \wedge$ $\text{no_cv_class?}(t) \wedge$ $\text{no_cv_union?}(t) \wedge$ $\text{no_cv_function?}(t) \wedge$ $\text{no_reference_to_void?}(t) \wedge$ $\text{no_cv_void?}(t) \wedge \text{no_array_of_bitfields?}(t)$
{-2}	$\text{const?}(\text{typ}')$
{-3}	$\text{const?}(\text{typ}(\text{typ}'))$
{1}	$\text{subterm}(\text{cv_base}(\text{typ}(\text{typ}(\text{typ}'))), \text{typ}')$

Instantiating the top quantifier in -1 with the terms: (typ!1),

Expanding the definition of subterm,

which is trivially true.

This completes the proof of subterm_cv_base.1.

subterm_cv_base.2:

<div style="display: flex; flex-direction: column; gap: 5px;"> <div style="display: flex; align-items: flex-start;"> {-1} <div style="flex-grow: 1;"> $\forall (t: \text{Cpp_Type_}):$ $\text{subterm}(t, \text{typ}') \supset$ $\text{no_pointers_to_bitfield?}(t) \wedge$ $\text{no_pointers_to_references?}(t) \wedge$ $\text{no_cv_references?}(t) \wedge$ $\text{no_reference_to_reference?}(t) \wedge$ $\text{no_reference_to_bitfields?}(t) \wedge$ $\text{no_pointer_to_member_to_reference?}(t) \wedge$ $\text{no_pointer_to_member_to_cv_void?}(t) \wedge$ $\text{no_cv_void_parameter?}(t) \wedge$ $\text{no_array_of_references?}(t) \wedge$ $\text{no_array_of_cv_void?}(t) \wedge$ $\text{no_array_of_function?}(t) \wedge$ $\text{no_array_of_abstract_class?}(t) \wedge$ $\text{no_pointer_or_ref_to_incomplete_array_parameter?}(t) \wedge$ $\text{no_array_return_type?}(t) \wedge$ $\text{no_function_return_type?}(t) \wedge$ $\text{bitfield_underlying_integral_or_enum_type?}(t) \wedge$ $\text{cv_array?}(t) \wedge$ $\text{enum_underlying_integral?}(t) \wedge$ $\text{enum_constants?}(t) \wedge$ $(\text{volatile?}(t) \supset \neg \text{const?}(\text{typ}(t))) \wedge$ $(\text{const?}(t) \supset \neg \text{const?}(\text{typ}(t))) \wedge$ $(\text{volatile?}(t) \supset \neg \text{volatile?}(\text{typ}(t))) \wedge$ $\text{no_cv_class?}(t) \wedge$ $\text{no_cv_union?}(t) \wedge$ $\text{no_cv_function?}(t) \wedge$ $\text{no_reference_to_void?}(t) \wedge$ $\text{no_cv_void?}(t) \wedge \text{no_array_of_bitfields?}(t)$ </div> </div> <div style="display: flex; align-items: flex-start; margin-top: 5px;"> {-2} $\text{const?}(\text{typ}')$ </div> <div style="display: flex; align-items: flex-start; margin-top: 5px;"> {-3} $\text{volatile?}(\text{typ}(\text{typ}'))$ </div> <hr style="border: 1px solid black; margin: 5px 0;"/> <div style="display: flex; align-items: flex-start;"> {1} $\text{subterm}(\text{cv_base}(\text{typ}(\text{typ}(\text{typ}'))), \text{typ}')$ </div> </div>	
--	--

Expanding the definition of cv_base,

Instantiating the top quantifier in -1 with the terms: (typ(typ!1)),

Expanding the definition of subterm,

Expanding the definition of subterm,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Expanding the definition of subterm,

which is trivially true.

This completes the proof of subterm_cv_base.2.

subterm_cv_base.3:

<div style="display: flex; align-items: flex-start;"> <div style="margin-right: 10px;">{-1}</div> <div> $\forall (t: \text{Cpp_Type_}):$ $\text{subterm}(t, \text{typ}') \supset$ $\text{no_pointers_to_bitfield?}(t) \wedge$ $\text{no_pointers_to_references?}(t) \wedge$ $\text{no_cv_references?}(t) \wedge$ $\text{no_reference_to_reference?}(t) \wedge$ $\text{no_reference_to_bitfields?}(t) \wedge$ $\text{no_pointer_to_member_to_reference?}(t) \wedge$ $\text{no_pointer_to_member_to_cv_void?}(t) \wedge$ $\text{no_cv_void_parameter?}(t) \wedge$ $\text{no_array_of_references?}(t) \wedge$ $\text{no_array_of_cv_void?}(t) \wedge$ $\text{no_array_of_function?}(t) \wedge$ $\text{no_array_of_abstract_class?}(t) \wedge$ $\text{no_pointer_or_ref_to_incomplete_array_parameter?}(t) \wedge$ $\text{no_array_return_type?}(t) \wedge$ $\text{no_function_return_type?}(t) \wedge$ $\text{bitfield_underlying_integral_or_enum_type?}(t) \wedge$ $\text{cv_array?}(t) \wedge$ $\text{enum_underlying_integral?}(t) \wedge$ $\text{enum_constants?}(t) \wedge$ $(\text{volatile?}(t) \supset \neg \text{const?}(\text{typ}(t))) \wedge$ $(\text{const?}(t) \supset \neg \text{const?}(\text{typ}(t))) \wedge$ $(\text{volatile?}(t) \supset \neg \text{volatile?}(\text{typ}(t))) \wedge$ $\text{no_cv_class?}(t) \wedge$ $\text{no_cv_union?}(t) \wedge$ $\text{no_cv_function?}(t) \wedge$ $\text{no_reference_to_void?}(t) \wedge$ $\text{no_cv_void?}(t) \wedge \text{no_array_of_bitfields?}(t)$ </div> </div>	<div style="display: flex; align-items: flex-start;"> <div style="margin-right: 10px;">{1}</div> <div>$\text{const?}(\text{typ}')$</div> </div> <div style="display: flex; align-items: flex-start; margin-top: 5px;"> <div style="margin-right: 10px;">{2}</div> <div>$\text{volatile?}(\text{typ}')$</div> </div> <div style="display: flex; align-items: flex-start; margin-top: 5px;"> <div style="margin-right: 10px;">{3}</div> <div>$\text{subterm}(\text{typ}', \text{typ}')$</div> </div>
--	---

Expanding the definition of subterm,

which is trivially true.

This completes the proof of subterm_cv_base.3.

subterm_cv_base.4:

{-1}	$\forall (t: \text{Cpp_Type_}):$ $\text{subterm}(t, \text{typ}') \supset$ $\text{no_pointers_to_bitfield?}(t) \wedge$ $\text{no_pointers_to_references?}(t) \wedge$ $\text{no_cv_references?}(t) \wedge$ $\text{no_reference_to_reference?}(t) \wedge$ $\text{no_reference_to_bitfields?}(t) \wedge$ $\text{no_pointer_to_member_to_reference?}(t) \wedge$ $\text{no_pointer_to_member_to_cv_void?}(t) \wedge$ $\text{no_cv_void_parameter?}(t) \wedge$ $\text{no_array_of_references?}(t) \wedge$ $\text{no_array_of_cv_void?}(t) \wedge$ $\text{no_array_of_function?}(t) \wedge$ $\text{no_array_of_abstract_class?}(t) \wedge$ $\text{no_pointer_or_ref_to_incomplete_array_parameter?}(t) \wedge$ $\text{no_array_return_type?}(t) \wedge$ $\text{no_function_return_type?}(t) \wedge$ $\text{bitfield_underlying_integral_or_enum_type?}(t) \wedge$ $\text{cv_array?}(t) \wedge$ $\text{enum_underlying_integral?}(t) \wedge$ $\text{enum_constants?}(t) \wedge$ $(\text{volatile?}(t) \supset \neg \text{const?}(\text{typ}(t))) \wedge$ $(\text{const?}(t) \supset \neg \text{const?}(\text{typ}(t))) \wedge$ $(\text{volatile?}(t) \supset \neg \text{volatile?}(\text{typ}(t))) \wedge$ $\text{no_cv_class?}(t) \wedge$ $\text{no_cv_union?}(t) \wedge$ $\text{no_cv_function?}(t) \wedge$ $\text{no_reference_to_void?}(t) \wedge$ $\text{no_cv_void?}(t) \wedge \text{no_array_of_bitfields?}(t)$
{-2}	$\text{const?}(\text{typ}(\text{typ}'))$
{-3}	$\text{volatile?}(\text{typ}')$
{1}	$\text{subterm}(\text{cv_base}(\text{typ}(\text{typ}(\text{typ}'))), \text{typ}')$

Instantiating the top quantifier in -1 with the terms: (typ!1),

Expanding the definition of subterm,

which is trivially true.

This completes the proof of subterm_cv_base.4.

subterm_cv_base.5:

<div style="display: flex; flex-direction: column; align-items: flex-start;"> <div style="margin-bottom: 10px;"> $\{-1\} \quad \forall (t: \text{Cpp_Type_}):$ </div> <div style="margin-bottom: 10px;"> $\text{subterm}(t, \text{typ}') \supset$ </div> <div style="margin-bottom: 10px;"> $\text{no_pointers_to_bitfield?}(t) \wedge$ </div> <div style="margin-bottom: 10px;"> $\text{no_pointers_to_references?}(t) \wedge$ </div> <div style="margin-bottom: 10px;"> $\text{no_cv_references?}(t) \wedge$ </div> <div style="margin-bottom: 10px;"> $\text{no_reference_to_reference?}(t) \wedge$ </div> <div style="margin-bottom: 10px;"> $\text{no_reference_to_bitfields?}(t) \wedge$ </div> <div style="margin-bottom: 10px;"> $\text{no_pointer_to_member_to_reference?}(t) \wedge$ </div> <div style="margin-bottom: 10px;"> $\text{no_pointer_to_member_to_cv_void?}(t) \wedge$ </div> <div style="margin-bottom: 10px;"> $\text{no_cv_void_parameter?}(t) \wedge$ </div> <div style="margin-bottom: 10px;"> $\text{no_array_of_references?}(t) \wedge$ </div> <div style="margin-bottom: 10px;"> $\text{no_array_of_cv_void?}(t) \wedge$ </div> <div style="margin-bottom: 10px;"> $\text{no_array_of_function?}(t) \wedge$ </div> <div style="margin-bottom: 10px;"> $\text{no_array_of_abstract_class?}(t) \wedge$ </div> <div style="margin-bottom: 10px;"> $\text{no_pointer_or_ref_to_incomplete_array_parameter?}(t) \wedge$ </div> <div style="margin-bottom: 10px;"> $\text{no_array_return_type?}(t) \wedge$ </div> <div style="margin-bottom: 10px;"> $\text{no_function_return_type?}(t) \wedge$ </div> <div style="margin-bottom: 10px;"> $\text{bitfield_underlying_integral_or_enum_type?}(t) \wedge$ </div> <div style="margin-bottom: 10px;"> $\text{cv_array?}(t) \wedge$ </div> <div style="margin-bottom: 10px;"> $\text{enum_underlying_integral?}(t) \wedge$ </div> <div style="margin-bottom: 10px;"> $\text{enum_constants?}(t) \wedge$ </div> <div style="margin-bottom: 10px;"> $(\text{volatile?}(t) \supset \neg \text{const?}(\text{typ}(t))) \wedge$ </div> <div style="margin-bottom: 10px;"> $(\text{const?}(t) \supset \neg \text{const?}(\text{typ}(t))) \wedge$ </div> <div style="margin-bottom: 10px;"> $(\text{volatile?}(t) \supset \neg \text{volatile?}(\text{typ}(t))) \wedge$ </div> <div style="margin-bottom: 10px;"> $\text{no_cv_class?}(t) \wedge$ </div> <div style="margin-bottom: 10px;"> $\text{no_cv_union?}(t) \wedge$ </div> <div style="margin-bottom: 10px;"> $\text{no_cv_function?}(t) \wedge$ </div> <div style="margin-bottom: 10px;"> $\text{no_reference_to_void?}(t) \wedge$ </div> <div style="margin-bottom: 10px;"> $\text{no_cv_void?}(t) \wedge \text{no_array_of_bitfields?}(t)$ </div> </div> <div style="margin-bottom: 10px;"> $\{-2\} \quad \text{volatile?}(\text{typ}(\text{typ}'))$ </div> <div style="margin-bottom: 10px;"> $\{-3\} \quad \text{volatile?}(\text{typ}')$ </div> <hr style="border: 0.5px solid black; margin: 5px 0;"/> <div style="margin-bottom: 10px;"> $\{1\} \quad \text{subterm}(\text{cv_base}(\text{typ}(\text{typ}(\text{typ}'))), \text{typ}')$ </div>

Instantiating the top quantifier in -1 with the terms: (typ!1),
Expanding the definition of subterm,
which is trivially true.
This completes the proof of subterm_cv_base.5.
Q.E.D.

C.40.24 Cpp_Deep_Type_Wellformedness.cv_base_result

Terse proof for cv_base_result.

cv_base_result:

$\{1\} \quad \forall (P: \{Q: \text{PRED}[\text{Cpp_Type_}] \mid (Q \subseteq \text{interpreted?})\}, \text{typ}: \text{Cpp_Subtype}(\text{cv}(P))):$ $P(\text{cv_base}(\text{typ}))$	
--	--

Repeatedly Skolemizing and flattening,
Expanding the definition of cv,
Expanding the definition of c,
Expanding the definition of v,

Installing automatic rewrites from: cv_base

Case splitting on FORALL (typ: (P1)): NOT const?(typ) AND NOT volatile?(typ),
we get 2 subgoals:

cv_base_result.1:

{-1}	$\forall (\text{typ}: (P')): \neg \text{const?}(\text{typ}) \wedge \neg \text{volatile?}(\text{typ})$
{-2}	$(P' \subseteq \text{interpreted?})$
{-3}	$\text{Cpp_Type?}(\text{typ}')$
{-4}	$(P'(\text{typ}') \vee (\text{const?}(\text{typ}') \wedge P'(\text{typ}(\text{typ}')))) \vee$ $(P'(\text{typ}') \vee (\text{volatile?}(\text{typ}') \wedge P'(\text{typ}(\text{typ}')))) \vee$ $(\text{const?}(\text{typ}') \wedge \text{volatile?}(\text{typ}(\text{typ}')) \wedge P'(\text{typ}(\text{typ}'))))$
{1} $P'(\text{cv_base}(\text{typ}'))$	

Hiding formulas: (-2 -3),

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
we get 4 subgoals:

cv_base_result.1.1:

{-1}	$\forall (\text{typ}: (P')): \neg \text{const?}(\text{typ}) \wedge \neg \text{volatile?}(\text{typ})$
{-2}	$P'(\text{typ}')$
{1} $P'(\text{cv_base}(\text{typ}'))$	

Instantiating quantified variables,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of cv_base_result.1.1.

cv_base_result.1.2:

{-1}	$\forall (\text{typ}: (P')): \neg \text{const?}(\text{typ}) \wedge \neg \text{volatile?}(\text{typ})$
{-2}	$\text{const?}(\text{typ}')$
{-3}	$P'(\text{typ}(\text{typ}'))$
{1} $P'(\text{cv_base}(\text{typ}(\text{typ}')))$	

Instantiating quantified variables,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of cv_base_result.1.2.

cv_base_result.1.3:

{-1}	$\forall (\text{typ}: (P')): \neg \text{const?}(\text{typ}) \wedge \neg \text{volatile?}(\text{typ})$
{-2}	$\text{const?}(\text{typ}')$
{-3}	$\text{volatile?}(\text{typ}(\text{typ}'))$
{-4}	$P'(\text{typ}(\text{typ}(\text{typ}')))$
{1} $P'(\text{cv_base}(\text{typ}(\text{typ}(\text{typ}'))))$	

Instantiating quantified variables,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of cv_base_result.1.3.

cv_base_result.1.4:

{-1}	$\forall (\text{typ}: (P')): \neg \text{const?}(\text{typ}) \wedge \neg \text{volatile?}(\text{typ})$
{-2}	$P'(\text{typ}(\text{typ}'))$
{-3}	$\text{volatile?}(\text{typ}')$
{1} $P'(\text{cv_base}(\text{typ}(\text{typ}')))$	

Instantiating quantified variables,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of cv_base_result.1.4.

cv_base_result.2:

<div style="display: flex; flex-direction: column; gap: 5px;"> <div style="border-bottom: 1px solid black; padding-bottom: 5px;"> <p>{-1} $(P' \subseteq \text{interpreted?})$</p> <p>{-2} $\text{Cpp_Type?}(\text{typ}')$</p> <p>{-3} $(P'(\text{typ}') \vee (\text{const?}(\text{typ}') \wedge P'(\text{typ}(\text{typ}')))) \vee$ $(P'(\text{typ}') \vee (\text{volatile?}(\text{typ}') \wedge P'(\text{typ}(\text{typ}')))) \vee$ $(\text{const?}(\text{typ}') \wedge \text{volatile?}(\text{typ}(\text{typ}')) \wedge P'(\text{typ}(\text{typ}(\text{typ}'))))$</p> </div> <p>{1} $\forall (\text{typ}: (P')): \neg \text{const?}(\text{typ}) \wedge \neg \text{volatile?}(\text{typ})$</p> <p>{2} $P'(\text{cv_base}(\text{typ}'))$</p> </div>	
---	--

Keeping (-1 1) and hiding *,
 Repeatedly Skolemizing and flattening,
 Expanding the definition of subset?,
 Expanding the definition of member,
 Instantiating quantified variables,
 Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of cv_base_result.2.
 Q.E.D.

C.40.25 Cpp_Deep_Type_Wellformedness.cv_base_result2

Terse proof for cv_base_result2.

cv_base_result2:

<p>{1} $\forall (\text{typ}: \text{Cpp_Type}): \neg \text{const?}(\text{typ}) \wedge \neg \text{volatile?}(\text{typ}) \supset \text{cv_base}(\text{typ}) = \text{typ}$</p>	
--	--

Repeatedly Skolemizing and flattening,
 Hiding formulas: -1,
 Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of cv_base_result2.
 Q.E.D.

C.40.26 Cpp_Deep_Type_Wellformedness.cv_base_not_const

Terse proof for cv_base_not_const.

cv_base_not_const:

<p>{1} $\forall (\text{typ}: \text{Cpp_Type}): \neg \text{const?}(\text{cv_base}(\text{typ}))$</p>	
---	--

Repeatedly Skolemizing and flattening,
 Expanding the definition of cv_base,
 Expanding the definition of cv_base,
 Expanding the definition of Cpp_Type?,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 we get 4 subgoals:

cv_base_not_const.1:

<pre> {-1} ∀ (t: Cpp_Type_): subterm(t, typ') ⊃ no_pointers_to_bitfield?(t) ∧ no_pointers_to_references?(t) ∧ no_cv_references?(t) ∧ no_reference_to_reference?(t) ∧ no_reference_to_bitfields?(t) ∧ no_pointer_to_member_to_reference?(t) ∧ no_pointer_to_member_to_cv_void?(t) ∧ no_cv_void_parameter?(t) ∧ no_array_of_references?(t) ∧ no_array_of_cv_void?(t) ∧ no_array_of_function?(t) ∧ no_array_of_abstract_class?(t) ∧ no_pointer_or_ref_to_incomplete_array_parameter?(t) ∧ no_array_return_type?(t) ∧ no_function_return_type?(t) ∧ bitfield_underlying_integral_or_enum_type?(t) ∧ cv_array?(t) ∧ enum_underlying_integral?(t) ∧ enum_constants?(t) ∧ const_volatile?(t) ∧ const_stutter?(t) ∧ volatile_stutter?(t) ∧ no_cv_class?(t) ∧ no_cv_union?(t) ∧ no_cv_function?(t) ∧ no_reference_to_void?(t) ∧ no_cv_void?(t) ∧ no_array_of_bitfields?(t) {-2} const?(typ') {-3} const?(typ(typ')) {-4} const?(cv_base(typ(typ(typ')))) </pre>	
--	--

Instantiating quantified variables,

Expanding the definition of subterm,

Expanding the definition of const_stutter?,

which is trivially true.

This completes the proof of cv_base_not_const.1.

cv_base_not_const.2:

```

{-1}  ∀ (t: Cpp_Type_):
      subterm(t, typ') ⊃
      no_pointers_to_bitfield?(t) ∧
      no_pointers_to_references?(t) ∧
      no_cv_references?(t) ∧
      no_reference_to_reference?(t) ∧
      no_reference_to_bitfields?(t) ∧
      no_pointer_to_member_to_reference?(t) ∧
      no_pointer_to_member_to_cv_void?(t) ∧
      no_cv_void_parameter?(t) ∧
      no_array_of_references?(t) ∧
      no_array_of_cv_void?(t) ∧
      no_array_of_function?(t) ∧
      no_array_of_abstract_class?(t) ∧
      no_pointer_or_ref_to_incomplete_array_parameter?(t) ∧
      no_array_return_type?(t) ∧
      no_function_return_type?(t) ∧
      bitfield_underlying_integral_or_enum_type?(t) ∧
      cv_array?(t) ∧
      enum_underlying_integral?(t) ∧
      enum_constants?(t) ∧
      const_volatile?(t) ∧
      const_stutter?(t) ∧
      volatile_stutter?(t) ∧
      no_cv_class?(t) ∧
      no_cv_union?(t) ∧
      no_cv_function?(t) ∧
      no_reference_to_void?(t) ∧
      no_cv_void?(t) ∧ no_array_of_bitfields?(t)
{-2}  const?(typ')
{-3}  const?(cv_base(typ(typ(typ'))))
{-4}  volatile?(typ(typ'))

```

Expanding the definition of cv_base,

Instantiating quantified variables,

Expanding the definition of subterm,

Expanding the definition of subterm,

Expanding the definition of volatile_stutter?,

Expanding the definition of const_volatile?,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of cv_base_not_const.2.

cv_base_not_const.3:

<div style="display: flex; flex-direction: column; gap: 5px;"> <div style="margin-bottom: 10px;">{-1} $\forall (t: \text{Cpp_Type_}):$</div> <div style="margin-bottom: 5px;">subterm(t, typ') \supset</div> <div style="margin-bottom: 5px;">no_pointers_to_bitfield?(t) \wedge</div> <div style="margin-bottom: 5px;">no_pointers_to_references?(t) \wedge</div> <div style="margin-bottom: 5px;">no_cv_references?(t) \wedge</div> <div style="margin-bottom: 5px;">no_reference_to_reference?(t) \wedge</div> <div style="margin-bottom: 5px;">no_reference_to_bitfields?(t) \wedge</div> <div style="margin-bottom: 5px;">no_pointer_to_member_to_reference?(t) \wedge</div> <div style="margin-bottom: 5px;">no_pointer_to_member_to_cv_void?(t) \wedge</div> <div style="margin-bottom: 5px;">no_cv_void_parameter?(t) \wedge</div> <div style="margin-bottom: 5px;">no_array_of_references?(t) \wedge</div> <div style="margin-bottom: 5px;">no_array_of_cv_void?(t) \wedge</div> <div style="margin-bottom: 5px;">no_array_of_function?(t) \wedge</div> <div style="margin-bottom: 5px;">no_array_of_abstract_class?(t) \wedge</div> <div style="margin-bottom: 5px;">no_pointer_or_ref_to_incomplete_array_parameter?(t) \wedge</div> <div style="margin-bottom: 5px;">no_array_return_type?(t) \wedge</div> <div style="margin-bottom: 5px;">no_function_return_type?(t) \wedge</div> <div style="margin-bottom: 5px;">bitfield_underlying_integral_or_enum_type?(t) \wedge</div> <div style="margin-bottom: 5px;">cv_array?(t) \wedge</div> <div style="margin-bottom: 5px;">enum_underlying_integral?(t) \wedge</div> <div style="margin-bottom: 5px;">enum_constants?(t) \wedge</div> <div style="margin-bottom: 5px;">const_volatile?(t) \wedge</div> <div style="margin-bottom: 5px;">const_stutter?(t) \wedge</div> <div style="margin-bottom: 5px;">volatile_stutter?(t) \wedge</div> <div style="margin-bottom: 5px;">no_cv_class?(t) \wedge</div> <div style="margin-bottom: 5px;">no_cv_union?(t) \wedge</div> <div style="margin-bottom: 5px;">no_cv_function?(t) \wedge</div> <div style="margin-bottom: 5px;">no_reference_to_void?(t) \wedge</div> <div style="margin-bottom: 5px;">no_cv_void?(t) \wedge no_array_of_bitfields?(t)</div> </div>	<div style="margin-bottom: 10px;">{-2} const?(typ(typ'))</div> <div style="margin-bottom: 5px;">{-3} const?(cv_base(typ(typ(typ'))))</div> <div style="margin-bottom: 5px;">{-4} volatile?(typ')</div>
--	--

Instantiating quantified variables,

Expanding the definition of const_volatile?,

Expanding the definition of subterm,

which is trivially true.

This completes the proof of cv_base_not_const.3.

cv_base_not_const.4:

```

{-1}  ∀ (t: Cpp_Type_):
      subterm(t, typ') ⊃
      no_pointers_to_bitfield?(t) ∧
      no_pointers_to_references?(t) ∧
      no_cv_references?(t) ∧
      no_reference_to_reference?(t) ∧
      no_reference_to_bitfields?(t) ∧
      no_pointer_to_member_to_reference?(t) ∧
      no_pointer_to_member_to_cv_void?(t) ∧
      no_cv_void_parameter?(t) ∧
      no_array_of_references?(t) ∧
      no_array_of_cv_void?(t) ∧
      no_array_of_function?(t) ∧
      no_array_of_abstract_class?(t) ∧
      no_pointer_or_ref_to_incomplete_array_parameter?(t) ∧
      no_array_return_type?(t) ∧
      no_function_return_type?(t) ∧
      bitfield_underlying_integral_or_enum_type?(t) ∧
      cv_array?(t) ∧
      enum_underlying_integral?(t) ∧
      enum_constants?(t) ∧
      const_volatile?(t) ∧
      const_stutter?(t) ∧
      volatile_stutter?(t) ∧
      no_cv_class?(t) ∧
      no_cv_union?(t) ∧
      no_cv_function?(t) ∧
      no_reference_to_void?(t) ∧
      no_cv_void?(t) ∧ no_array_of_bitfields?(t)
{-2}  const?(cv_base(typ(typ(typ'))))
{-3}  volatile?(typ(typ'))
{-4}  volatile?(typ')

```

Instantiating quantified variables,
Expanding the definition of volatile_stutter?,
Expanding the definition of subterm,
which is trivially true.
This completes the proof of cv_base_not_const.4.
Q.E.D.

C.40.27 Cpp_Deep_Type_Wellformedness.cv_base_not_volatile

Terse proof for cv_base_not_volatile.

cv_base_not_volatile:

```

{1}  ∀ (typ: Cpp_Type): ¬ volatile?(cv_base(typ))

```

Repeatedly Skolemizing and flattening,
Expanding the definition of cv_base,
Expanding the definition of cv_base,

Expanding the definition of Cpp_Type?,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 4 subgoals:

cv_base_not_volatile.1:

{-1}	$\forall (t: \text{Cpp_Type_}):$ $\text{subterm}(t, \text{typ}') \supset$ $\text{no_pointers_to_bitfield?}(t) \wedge$ $\text{no_pointers_to_references?}(t) \wedge$ $\text{no_cv_references?}(t) \wedge$ $\text{no_reference_to_reference?}(t) \wedge$ $\text{no_reference_to_bitfields?}(t) \wedge$ $\text{no_pointer_to_member_to_reference?}(t) \wedge$ $\text{no_pointer_to_member_to_cv_void?}(t) \wedge$ $\text{no_cv_void_parameter?}(t) \wedge$ $\text{no_array_of_references?}(t) \wedge$ $\text{no_array_of_cv_void?}(t) \wedge$ $\text{no_array_of_function?}(t) \wedge$ $\text{no_array_of_abstract_class?}(t) \wedge$ $\text{no_pointer_or_ref_to_incomplete_array_parameter?}(t) \wedge$ $\text{no_array_return_type?}(t) \wedge$ $\text{no_function_return_type?}(t) \wedge$ $\text{bitfield_underlying_integral_or_enum_type?}(t) \wedge$ $\text{cv_array?}(t) \wedge$ $\text{enum_underlying_integral?}(t) \wedge$ $\text{enum_constants?}(t) \wedge$ $\text{const_volatile?}(t) \wedge$ $\text{const_stutter?}(t) \wedge$ $\text{volatile_stutter?}(t) \wedge$ $\text{no_cv_class?}(t) \wedge$ $\text{no_cv_union?}(t) \wedge$ $\text{no_cv_function?}(t) \wedge$ $\text{no_reference_to_void?}(t) \wedge$ $\text{no_cv_void?}(t) \wedge \text{no_array_of_bitfields?}(t)$
{-2}	const?(typ')
{-3}	const?(typ(typ'))
{-4}	volatile?(cv_base(typ(typ(typ'))))

Instantiating quantified variables,

Expanding the definition of subterm,

Expanding the definition of const_stutter?,

which is trivially true.

This completes the proof of cv_base_not_volatile.1.

cv_base_not_volatile.2:

```

{-1}  ∀ (t: Cpp_Type_):
      subterm(t, typ') ⊃
      no_pointers_to_bitfield?(t) ∧
      no_pointers_to_references?(t) ∧
      no_cv_references?(t) ∧
      no_reference_to_reference?(t) ∧
      no_reference_to_bitfields?(t) ∧
      no_pointer_to_member_to_reference?(t) ∧
      no_pointer_to_member_to_cv_void?(t) ∧
      no_cv_void_parameter?(t) ∧
      no_array_of_references?(t) ∧
      no_array_of_cv_void?(t) ∧
      no_array_of_function?(t) ∧
      no_array_of_abstract_class?(t) ∧
      no_pointer_or_ref_to_incomplete_array_parameter?(t) ∧
      no_array_return_type?(t) ∧
      no_function_return_type?(t) ∧
      bitfield_underlying_integral_or_enum_type?(t) ∧
      cv_array?(t) ∧
      enum_underlying_integral?(t) ∧
      enum_constants?(t) ∧
      const_volatile?(t) ∧
      const_stutter?(t) ∧
      volatile_stutter?(t) ∧
      no_cv_class?(t) ∧
      no_cv_union?(t) ∧
      no_cv_function?(t) ∧
      no_reference_to_void?(t) ∧
      no_cv_void?(t) ∧ no_array_of_bitfields?(t)
{-2}  const?(typ')
{-3}  volatile?(cv_base(typ(typ(typ'))))
{-4}  volatile?(typ(typ'))

```

Expanding the definition of cv_base,

Instantiating quantified variables,

Expanding the definition of subterm,

Expanding the definition of subterm,

Expanding the definition of volatile_stutter?,

Expanding the definition of const_volatile?,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of cv_base_not_volatile.2.

cv_base_not_volatile.3:

<pre> {-1} ∀ (t: Cpp_Type_): subterm(t, typ') ⊃ no_pointers_to_bitfield?(t) ∧ no_pointers_to_references?(t) ∧ no_cv_references?(t) ∧ no_reference_to_reference?(t) ∧ no_reference_to_bitfields?(t) ∧ no_pointer_to_member_to_reference?(t) ∧ no_pointer_to_member_to_cv_void?(t) ∧ no_cv_void_parameter?(t) ∧ no_array_of_references?(t) ∧ no_array_of_cv_void?(t) ∧ no_array_of_function?(t) ∧ no_array_of_abstract_class?(t) ∧ no_pointer_or_ref_to_incomplete_array_parameter?(t) ∧ no_array_return_type?(t) ∧ no_function_return_type?(t) ∧ bitfield_underlying_integral_or_enum_type?(t) ∧ cv_array?(t) ∧ enum_underlying_integral?(t) ∧ enum_constants?(t) ∧ const_volatile?(t) ∧ const_stutter?(t) ∧ volatile_stutter?(t) ∧ no_cv_class?(t) ∧ no_cv_union?(t) ∧ no_cv_function?(t) ∧ no_reference_to_void?(t) ∧ no_cv_void?(t) ∧ no_array_of_bitfields?(t) {-2} const?(typ(typ')) {-3} volatile?(cv_base(typ(typ(typ')))) {-4} volatile?(typ') </pre>	
--	--

Instantiating quantified variables,

Expanding the definition of const_volatile?,

Expanding the definition of subterm,

which is trivially true.

This completes the proof of cv_base_not_volatile.3.

cv_base_not_volatile.4:

<div style="display: flex; flex-direction: column; gap: 10px;"> <div style="margin-bottom: 10px;">{-1}</div> <div style="margin-bottom: 10px;">{-2}</div> <div style="margin-bottom: 10px;">{-3}</div> <div style="margin-bottom: 10px;">{-4}</div> </div>	$\forall (t: \text{Cpp_Type_}):$ $\text{subterm}(t, \text{typ}') \supset$ $\text{no_pointers_to_bitfield?}(t) \wedge$ $\text{no_pointers_to_references?}(t) \wedge$ $\text{no_cv_references?}(t) \wedge$ $\text{no_reference_to_reference?}(t) \wedge$ $\text{no_reference_to_bitfields?}(t) \wedge$ $\text{no_pointer_to_member_to_reference?}(t) \wedge$ $\text{no_pointer_to_member_to_cv_void?}(t) \wedge$ $\text{no_cv_void_parameter?}(t) \wedge$ $\text{no_array_of_references?}(t) \wedge$ $\text{no_array_of_cv_void?}(t) \wedge$ $\text{no_array_of_function?}(t) \wedge$ $\text{no_array_of_abstract_class?}(t) \wedge$ $\text{no_pointer_or_ref_to_incomplete_array_parameter?}(t) \wedge$ $\text{no_array_return_type?}(t) \wedge$ $\text{no_function_return_type?}(t) \wedge$ $\text{bitfield_underlying_integral_or_enum_type?}(t) \wedge$ $\text{cv_array?}(t) \wedge$ $\text{enum_underlying_integral?}(t) \wedge$ $\text{enum_constants?}(t) \wedge$ $\text{const_volatile?}(t) \wedge$ $\text{const_stutter?}(t) \wedge$ $\text{volatile_stutter?}(t) \wedge$ $\text{no_cv_class?}(t) \wedge$ $\text{no_cv_union?}(t) \wedge$ $\text{no_cv_function?}(t) \wedge$ $\text{no_reference_to_void?}(t) \wedge$ $\text{no_cv_void?}(t) \wedge \text{no_array_of_bitfields?}(t)$
--	---

Instantiating quantified variables,
Expanding the definition of volatile_stutter?,
Expanding the definition of subterm,
which is trivially true.
This completes the proof of cv_base_not_volatile.4.
Q.E.D.

C.40.28 Cpp_Deep_Type_Wellformedness.cv_base_cv

Terse proof for cv_base_cv.

cv_base_cv:

<div style="display: flex; flex-direction: column; gap: 10px;"> <div style="margin-bottom: 10px;">{1}</div> </div>	$\forall (P: \text{PRED}[\text{Cpp_Type_}], \text{typ}: \text{Cpp_Type}):$ $(P \subseteq \text{interpreted?}) \supset (P(\text{cv_base}(\text{typ})) \equiv \text{cv}(P)(\text{typ}))$
--	--

Repeatedly Skolemizing and flattening,
Applying propositional simplification,

we get 2 subgoals:

cv_base_cv.1:

{-1}	$P'(cv_base(typ'))$
{-2}	$Cpp_Type?(typ')$
{-3}	$(P' \subseteq interpreted?)$
{1}	$cv(P')(typ')$

Expanding the definition of cv,

Expanding the definition of c,

Expanding the definition of v,

Applying disjunctive simplification to flatten sequent,

Expanding the definition of cv_base,

Installing automatic rewrites from: subterm const_stutter? volatile_stutter? const_volatile?

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 5 subgoals:

cv_base_cv.1.1:

{-1}	$const?(typ')$
{-2}	$P'(cv_base(typ(typ')))$
{-3}	$Cpp_Type?(typ')$
{-4}	$(P' \subseteq interpreted?)$
{1}	$P'(typ')$
{2}	$P'(typ(typ'))$
{3}	$P'(typ(typ(typ')))$

Expanding the definition of cv_base,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 2 subgoals:

cv_base_cv.1.1.1:

{-1}	$const?(typ')$
{-2}	$const?(typ(typ'))$
{-3}	$P'(cv_base(typ(typ(typ'))))$
{-4}	$Cpp_Type?(typ')$
{-5}	$(P' \subseteq interpreted?)$
{1}	$P'(typ(typ'))$
{2}	$P'(typ')$
{3}	$P'(typ(typ(typ')))$

Expanding the definition of Cpp_Type?,

Instantiating quantified variables,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of cv_base_cv.1.1.1.

cv_base_cv.1.1.2:

{-1}	$const?(typ')$
{-2}	$P'(cv_base(typ(typ(typ'))))$
{-3}	$volatile?(typ(typ'))$
{-4}	$Cpp_Type?(typ')$
{-5}	$(P' \subseteq interpreted?)$
{1}	$P'(typ(typ'))$
{2}	$P'(typ')$
{3}	$P'(typ(typ(typ')))$

Expanding the definition of cv_base,

C Proof scripts

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
we get 2 subgoals:

`cv_base_cv.1.1.2.1:`

{-1}	<code>const?(typ')</code>
{-2}	<code>const?(typ(typ(typ')))</code>
{-3}	<code>P'(cv_base(typ(typ(typ'))))</code>
{-4}	<code>volatile?(typ(typ'))</code>
{-5}	<code>Cpp_Type?(typ')</code>
{-6}	<code>(P' \subseteq interpreted?)</code>
<hr/>	
{1}	<code>P'(typ(typ(typ')))</code>
{2}	<code>P'(typ(typ'))</code>
{3}	<code>P'(typ')</code>

Expanding the definition of `Cpp_Type?`,

Instantiating quantified variables,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `cv_base_cv.1.1.2.1`.

`cv_base_cv.1.1.2.2:`

{-1}	<code>const?(typ')</code>
{-2}	<code>P'(cv_base(typ(typ(typ'))))</code>
{-3}	<code>volatile?(typ(typ'))</code>
{-4}	<code>volatile?(typ(typ'))</code>
{-5}	<code>Cpp_Type?(typ')</code>
{-6}	<code>(P' \subseteq interpreted?)</code>
<hr/>	
{1}	<code>P'(typ(typ(typ')))</code>
{2}	<code>P'(typ(typ'))</code>
{3}	<code>P'(typ')</code>

Expanding the definition of `Cpp_Type?`,

Instantiating quantified variables,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `cv_base_cv.1.1.2.2`.

`cv_base_cv.1.2:`

{-1}	<code>P'(cv_base(typ(typ')))</code>
{-2}	<code>volatile?(typ')</code>
{-3}	<code>Cpp_Type?(typ')</code>
{-4}	<code>(P' \subseteq interpreted?)</code>
<hr/>	
{1}	<code>P'(typ')</code>
{2}	<code>P'(typ(typ'))</code>
{3}	<code>P'(typ(typ'))</code>

Expanding the definition of `cv_base`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 2 subgoals:

cv_base_cv.1.2.1:

{-1}	const?(typ(typ'))
{-2}	$P'(cv_base(yp(yp(typ'))))$
{-3}	volatile?(typ')
{-4}	Cpp_Type?(typ')
{-5}	$(P' \subseteq interpreted?)$
{1}	$P'(yp(yp'))$
{2}	$P'(yp')$
{3}	$P'(yp(yp'))$

Expanding the definition of Cpp_Type?,

Instantiating quantified variables,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of cv_base_cv.1.2.1.

cv_base_cv.1.2.2:

{-1}	$P'(cv_base(yp(yp(typ'))))$
{-2}	volatile?(yp(yp'))
{-3}	volatile?(yp')
{-4}	Cpp_Type?(yp')
{-5}	$(P' \subseteq interpreted?)$
{1}	$P'(yp(yp'))$
{2}	$P'(yp')$
{3}	$P'(yp(yp'))$

Expanding the definition of Cpp_Type?,

Instantiating quantified variables,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of cv_base_cv.1.2.2.

cv_base_cv.1.3:

{-1}	const?(yp')
{-2}	$P'(cv_base(yp(yp')))$
{-3}	Cpp_Type?(yp')
{-4}	$(P' \subseteq interpreted?)$
{1}	$P'(yp')$
{2}	$P'(yp(yp'))$
{3}	volatile?(yp(yp'))

Expanding the definition of cv_base,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Expanding the definition of Cpp_Type?,

Instantiating quantified variables,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of cv_base_cv.1.3.

cv_base_cv.1.4:

{-1}	$P'(cv_base(yp(yp')))$
{-2}	volatile?(yp')
{-3}	Cpp_Type?(yp')
{-4}	$(P' \subseteq interpreted?)$
{1}	$P'(yp')$
{2}	$P'(yp(yp'))$
{3}	volatile?(yp(yp'))

C Proof scripts

Expanding the definition of `cv_base`,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 Expanding the definition of `Cpp_Type?`,
 Instantiating quantified variables,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 This completes the proof of `cv_base_cv.1.4`.

`cv_base_cv.1.5`:

{-1}	$P'(\text{cv_base}(\text{typ}(\text{typ}'))))$
{-2}	$\text{volatile?}(\text{typ}')$
{-3}	$\text{Cpp_Type?}(\text{typ}')$
{-4}	$(P' \subseteq \text{interpreted?})$
{1}	$P'(\text{typ}')$
{2}	$P'(\text{typ}(\text{typ}'))$

Expanding the definition of `cv_base`,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 we get 2 subgoals:

`cv_base_cv.1.5.1`:

{-1}	$\text{const?}(\text{typ}(\text{typ}'))$
{-2}	$P'(\text{cv_base}(\text{typ}(\text{typ}(\text{typ}'))))$
{-3}	$\text{volatile?}(\text{typ}')$
{-4}	$\text{Cpp_Type?}(\text{typ}')$
{-5}	$(P' \subseteq \text{interpreted?})$
{1}	$P'(\text{typ}(\text{typ}'))$
{2}	$P'(\text{typ}')$

Expanding the definition of `Cpp_Type?`,
 Instantiating quantified variables,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 This completes the proof of `cv_base_cv.1.5.1`.

`cv_base_cv.1.5.2`:

{-1}	$P'(\text{cv_base}(\text{typ}(\text{typ}(\text{typ}'))))$
{-2}	$\text{volatile?}(\text{typ}(\text{typ}'))$
{-3}	$\text{volatile?}(\text{typ}')$
{-4}	$\text{Cpp_Type?}(\text{typ}')$
{-5}	$(P' \subseteq \text{interpreted?})$
{1}	$P'(\text{typ}(\text{typ}'))$
{2}	$P'(\text{typ}')$

Expanding the definition of `Cpp_Type?`,
 Instantiating quantified variables,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 This completes the proof of `cv_base_cv.1.5.2`.

`cv_base_cv.2`:

{-1}	$\text{cv}(P')(\text{typ}')$
{-2}	$\text{Cpp_Type?}(\text{typ}')$
{-3}	$(P' \subseteq \text{interpreted?})$
{1}	$P'(\text{cv_base}(\text{typ}'))$

Rewriting using `cv_base_result`, matching in `*` where `P` gets `P!1`,
 This completes the proof of `cv_base_cv.2`.
 Q.E.D.

C.40.29 Cpp_Deep_Type_Wellformedness.layout_compatible?_TCC1

Terse proof for layout_compatible?_TCC1.

layout_compatible?_TCC1:

{1}	$\exists (x: \text{equivalence}[\text{Cpp_Type}]): \text{TRUE}$
-----	--

Instantiating the top quantifier in 1 with the terms: (=),
we get 2 subgoals:

layout_compatible?_TCC1.1:

{1}	TRUE
-----	------

which is trivially true.

This completes the proof of layout_compatible?_TCC1.1.

layout_compatible?_TCC1.2:

{1}	$\text{equivalence?}[\text{Cpp_Type}]$ $(\text{restrict}[[\text{Cpp_Type}_-, \text{Cpp_Type}_-], [\text{Cpp_Type}, \text{Cpp_Type}], \text{boolean}] (=))$
-----	--

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of layout_compatible?_TCC1.2.
Q.E.D.

C.40.30**Cpp_Deep_Type_Wellformedness.layout_compatible_enums_TCC1**

Terse proof for layout_compatible_enums_TCC1.

layout_compatible_enums_TCC1:

{1}	$\forall (e_1, e_2: \text{Cpp_Subtype}(\text{enum?})):$ $\text{array?}(e_1) \vee$ $\text{pointer?}(e_1) \vee$ $\text{reference?}(e_1) \vee$ $\text{bitfield?}(e_1) \vee \text{enum?}(e_1) \vee \text{pointer_to_member?}(e_1) \vee \text{const?}(e_1) \vee \text{volatile?}(e_1)$
-----	--

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of layout_compatible_enums_TCC1.
Q.E.D.

C.40.31**Cpp_Deep_Type_Wellformedness.layout_compatible_enums_TCC2**

Terse proof for layout_compatible_enums_TCC2.

layout_compatible_enums_TCC2:

{1}	$\forall (e_1, e_2: \text{Cpp_Subtype}(\text{enum?})):$ $\text{array?}(e_2) \vee$ $\text{pointer?}(e_2) \vee$ $\text{reference?}(e_2) \vee$ $\text{bitfield?}(e_2) \vee \text{enum?}(e_2) \vee \text{pointer_to_member?}(e_2) \vee \text{const?}(e_2) \vee \text{volatile?}(e_2)$
-----	--

Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of `layout_compatible_enums_TCC2`.
 Q.E.D.

C.41 Proofs for `Cpp_Deep_Types` (`types.pvs`)

C.41.1 `Cpp_Deep_Types.Cpp_Type__pointer_to_member_eta_TCC1`

Terse proof for `Cpp_Type__pointer_to_member_eta_TCC1`.

`Cpp_Type__pointer_to_member_eta_TCC1`:

<pre>{1} ∃ (pointer_to_member?_var: (pointer_to_member?): array?(pointer_to_member?_var) ∨ pointer?(pointer_to_member?_var) ∨ reference?(pointer_to_member?_var) ∨ bitfield?(pointer_to_member?_var) ∨ enum?(pointer_to_member?_var) ∨ pointer_to_member?(pointer_to_member?_var) ∨ const?(pointer_to_member?_var) ∨ volatile?(pointer_to_member?_var)</pre>

Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of `Cpp_Type__pointer_to_member_eta_TCC1`.
 Q.E.D.

C.41.2 `Cpp_Deep_Types.Cpp_Type__pointer_to_member_eta_TCC2`

Terse proof for `Cpp_Type__pointer_to_member_eta_TCC2`.

`Cpp_Type__pointer_to_member_eta_TCC2`:

<pre>{1} ∃ (pointer_to_member?_var: (pointer_to_member?): class?(typ(pointer_to_member?_var))</pre>
--

Repeatedly Skolemizing and flattening,
 Adding type constraints for `typ(pointer_to_member?_var!1)`,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 This completes the proof of `Cpp_Type__pointer_to_member_eta_TCC2`.
 Q.E.D.

C.41.3 `Cpp_Deep_Types.Cpp_Type__typ_pointer_to_member_TCC1`

Terse proof for `Cpp_Type__typ_pointer_to_member_TCC1`.

`Cpp_Type__typ_pointer_to_member_TCC1`:

<pre>{1} ∃ (pointer_to_member1_var: (class?), pointer_to_member2_var: Cpp_Type_): array?(pointer_to_member(pointer_to_member1_var, pointer_to_member2_var)) ∨ pointer?(pointer_to_member(pointer_to_member1_var, pointer_to_member2_var)) ∨ reference?(pointer_to_member(pointer_to_member1_var, pointer_to_member2_var)) ∨ bitfield?(pointer_to_member(pointer_to_member1_var, pointer_to_member2_var)) ∨ enum?(pointer_to_member(pointer_to_member1_var, pointer_to_member2_var)) ∨ pointer_to_member?(pointer_to_member(pointer_to_member1_var, pointer_to_member2_var)) ∨ const?(pointer_to_member(pointer_to_member1_var, pointer_to_member2_var)) ∨ volatile?(pointer_to_member(pointer_to_member1_var, pointer_to_member2_var))</pre>

Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of `Cpp_Type__typ_pointer_to_member_TCC1`.
 Q.E.D.

C.41.4 Cpp_Deep_Types.subterm_less

Terse proof for `subterm_less`.

`subterm_less`:

$$\frac{}{\{1\} \quad \forall (t_1, t_2: \text{Cpp_Type_}): \text{subterm}(t_1, t_2) \wedge t_1 \neq t_2 \supset t_1 \ll t_2}$$

Inducting on t_2 on formula 1,
 we get 28 subgoals:

`subterm_less.1`:

$$\frac{}{\{1\} \quad \forall (t_1: \text{Cpp_Type_}): \text{subterm}(t_1, \text{uchar}) \wedge t_1 \neq \text{uchar} \supset t_1 \ll \text{uchar}}$$

Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of `subterm_less.1`.

`subterm_less.2`:

$$\frac{}{\{1\} \quad \forall (t_1: \text{Cpp_Type_}): \text{subterm}(t_1, \text{schar}) \wedge t_1 \neq \text{schar} \supset t_1 \ll \text{schar}}$$

Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of `subterm_less.2`.

`subterm_less.3`:

$$\frac{}{\{1\} \quad \forall (t_1: \text{Cpp_Type_}): \text{subterm}(t_1, \text{char}) \wedge t_1 \neq \text{char} \supset t_1 \ll \text{char}}$$

Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of `subterm_less.3`.

`subterm_less.4`:

$$\frac{}{\{1\} \quad \forall (t_1: \text{Cpp_Type_}): \text{subterm}(t_1, \text{short}) \wedge t_1 \neq \text{short} \supset t_1 \ll \text{short}}$$

Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of `subterm_less.4`.

`subterm_less.5`:

$$\frac{}{\{1\} \quad \forall (t_1: \text{Cpp_Type_}): \text{subterm}(t_1, \text{int}) \wedge t_1 \neq \text{int} \supset t_1 \ll \text{int}}$$

Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of `subterm_less.5`.

`subterm_less.6`:

$$\frac{}{\{1\} \quad \forall (t_1: \text{Cpp_Type_}): \text{subterm}(t_1, \text{long}) \wedge t_1 \neq \text{long} \supset t_1 \ll \text{long}}$$

Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of `subterm_less.6`.

`subterm_less.7`:

$$\frac{}{\{1\} \quad \forall (t_1: \text{Cpp_Type_}): \text{subterm}(t_1, \text{longlong}) \wedge t_1 \neq \text{longlong} \supset t_1 \ll \text{longlong}}$$

C Proof scripts

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `subterm_less.7`.

`subterm_less.8`:

$$\{1\} \quad \forall (t_1 : \text{Cpp_Type_}) : \text{subterm}(t_1, \text{ushort}) \wedge t_1 \neq \text{ushort} \supset t_1 \ll \text{ushort}$$

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `subterm_less.8`.

`subterm_less.9`:

$$\{1\} \quad \forall (t_1 : \text{Cpp_Type_}) : \text{subterm}(t_1, \text{uint}) \wedge t_1 \neq \text{uint} \supset t_1 \ll \text{uint}$$

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `subterm_less.9`.

`subterm_less.10`:

$$\{1\} \quad \forall (t_1 : \text{Cpp_Type_}) : \text{subterm}(t_1, \text{ulong}) \wedge t_1 \neq \text{ulong} \supset t_1 \ll \text{ulong}$$

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `subterm_less.10`.

`subterm_less.11`:

$$\{1\} \quad \forall (t_1 : \text{Cpp_Type_}) : \text{subterm}(t_1, \text{ulonglong}) \wedge t_1 \neq \text{ulonglong} \supset t_1 \ll \text{ulonglong}$$

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `subterm_less.11`.

`subterm_less.12`:

$$\{1\} \quad \forall (t_1 : \text{Cpp_Type_}) : \text{subterm}(t_1, \text{wchar_t}) \wedge t_1 \neq \text{wchar_t} \supset t_1 \ll \text{wchar_t}$$

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `subterm_less.12`.

`subterm_less.13`:

$$\{1\} \quad \forall (t_1 : \text{Cpp_Type_}) : \text{subterm}(t_1, \text{bool}) \wedge t_1 \neq \text{bool} \supset t_1 \ll \text{bool}$$

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `subterm_less.13`.

`subterm_less.14`:

$$\{1\} \quad \forall (t_1 : \text{Cpp_Type_}) : \text{subterm}(t_1, \text{float}) \wedge t_1 \neq \text{float} \supset t_1 \ll \text{float}$$

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `subterm_less.14`.

`subterm_less.15`:

$$\{1\} \quad \forall (t_1 : \text{Cpp_Type_}) : \text{subterm}(t_1, \text{double}) \wedge t_1 \neq \text{double} \supset t_1 \ll \text{double}$$

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `subterm_less.15`.

subterm_less.16:

$$\{1\} \quad \forall (t_1: \text{Cpp_Type_}): \text{subterm}(t_1, \text{longdouble}) \wedge t_1 \neq \text{longdouble} \supset t_1 \ll \text{longdouble}$$

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of subterm_less.16.

subterm_less.17:

$$\{1\} \quad \forall (t_1: \text{Cpp_Type_}): \text{subterm}(t_1, \text{void}) \wedge t_1 \neq \text{void} \supset t_1 \ll \text{void}$$

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of subterm_less.17.

subterm_less.18:

$$\{1\} \quad \forall (\text{array_type1_var}: \text{Cpp_Type_}, \text{array_type2_var}: \text{nat}):$$

$$\quad (\forall (t_1: \text{Cpp_Type_}):$$

$$\quad \quad \text{subterm}(t_1, \text{array_type1_var}) \wedge t_1 \neq \text{array_type1_var} \supset t_1 \ll \text{array_type1_var})$$

$$\quad \supset$$

$$\quad (\forall (t_1: \text{Cpp_Type_}):$$

$$\quad \quad \text{subterm}(t_1, \text{array_type}(\text{array_type1_var}, \text{array_type2_var})) \wedge$$

$$\quad \quad t_1 \neq \text{array_type}(\text{array_type1_var}, \text{array_type2_var})$$

$$\quad \quad \supset t_1 \ll \text{array_type}(\text{array_type1_var}, \text{array_type2_var}))$$

Repeatedly Skolemizing and flattening,

Expanding the definition of ■,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Instantiating quantified variables,

Expanding the definition of subterm,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of subterm_less.18.

subterm_less.19:

$$\{1\} \quad \forall (\text{function_type1_var}: \text{Cpp_Type_}, \text{function_type2_var}: \text{list}[\text{Cpp_Type_}],$$

$$\quad \text{function_type3_var}: \text{bool}):$$

$$\quad (\forall (t_1: \text{Cpp_Type_}):$$

$$\quad \quad \text{subterm}(t_1, \text{function_type1_var}) \wedge t_1 \neq \text{function_type1_var} \supset t_1 \ll \text{function_type1_var})$$

$$\quad \wedge$$

$$\quad \text{every}(\lambda (t_2: \text{Cpp_Type_}): \forall (t_1: \text{Cpp_Type_}): \text{subterm}(t_1, t_2) \wedge t_1 \neq t_2 \supset t_1 \ll t_2)$$

$$\quad \quad (\text{function_type2_var})$$

$$\quad \supset$$

$$\quad (\forall (t_1: \text{Cpp_Type_}):$$

$$\quad \quad \text{subterm}(t_1, \text{function_type}(\text{function_type1_var}, \text{function_type2_var}, \text{function_type3_var}))$$

$$\quad \quad \wedge t_1 \neq \text{function_type}(\text{function_type1_var}, \text{function_type2_var}, \text{function_type3_var})$$

$$\quad \quad \supset t_1 \ll \text{function_type}(\text{function_type1_var}, \text{function_type2_var}, \text{function_type3_var}))$$

Repeatedly Skolemizing and flattening,

Expanding the definition of ■,

Expanding the definition of subterm,

C Proof scripts

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
we get 2 subgoals:

subterm_less.19.1:

{-1}	$\forall (t_1: \text{Cpp_Type_}):$
{-2}	$\text{subterm}(t_1, \text{function_type1_var}') \wedge t_1 \neq \text{function_type1_var}' \supset t_1 \ll \text{function_type1_var}'$
{-3}	$\text{every}(\lambda (t_2: \text{Cpp_Type_}): \forall (t_1: \text{Cpp_Type_}): \text{subterm}(t_1, t_2) \wedge t_1 \neq t_2 \supset t_1 \ll t_2)$ $(\text{function_type2_var}')$
{-3}	$\text{subterm}(t'_1, \text{function_type1_var}')$
{1}	$t'_1 = \text{function_type}(\text{function_type1_var}', \text{function_type2_var}', \text{function_type3_var}')$
{2}	$t'_1 = \text{function_type1_var}'$
{3}	$t'_1 \ll \text{function_type1_var}'$
{4}	$\text{some}[\text{Cpp_Type_}](\lambda (z: \text{Cpp_Type_}): t'_1 = z \vee t'_1 \ll z)(\text{function_type2_var}')$

Instantiating the top quantifier in -1 with the terms: (t!1),

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of **subterm_less.19.1**.

subterm_less.19.2:

{-1}	$\forall (t_1: \text{Cpp_Type_}):$
{-2}	$\text{subterm}(t_1, \text{function_type1_var}') \wedge t_1 \neq \text{function_type1_var}' \supset t_1 \ll \text{function_type1_var}'$
{-2}	$\text{every}(\lambda (t_2: \text{Cpp_Type_}): \forall (t_1: \text{Cpp_Type_}): \text{subterm}(t_1, t_2) \wedge t_1 \neq t_2 \supset t_1 \ll t_2)$ $(\text{function_type2_var}')$
{-3}	$\text{some}[\text{Cpp_Type_}](\lambda (z: \text{Cpp_Type_}): \text{subterm}(t'_1, z))(\text{function_type2_var}')$
{1}	$t'_1 = \text{function_type}(\text{function_type1_var}', \text{function_type2_var}', \text{function_type3_var}')$
{2}	$t'_1 = \text{function_type1_var}'$
{3}	$t'_1 \ll \text{function_type1_var}'$
{4}	$\text{some}[\text{Cpp_Type_}](\lambda (z: \text{Cpp_Type_}): t'_1 = z \vee t'_1 \ll z)(\text{function_type2_var}')$

Rewriting using `some_iff_exists`, matching in `*`,

Rewriting using `some_iff_exists`, matching in `*`,

Rewriting using `every_iff_forall`, matching in `*`,

Repeatedly Skolemizing and flattening,

Instantiating the top quantifier in 4 with the terms: (t!1),

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Instantiating the top quantifier in -2 with the terms: (t!1),

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Instantiating quantified variables,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of **subterm_less.19.2**.

subterm_less.20:

{1}	$\forall (\text{pointer1_var}: \text{Cpp_Type_}):$
	$(\forall (t_1: \text{Cpp_Type_}): \text{subterm}(t_1, \text{pointer1_var}) \wedge t_1 \neq \text{pointer1_var} \supset t_1 \ll \text{pointer1_var})$
	\supset
	$(\forall (t_1: \text{Cpp_Type_}):$
	$\text{subterm}(t_1, \text{pointer}(\text{pointer1_var})) \wedge t_1 \neq \text{pointer}(\text{pointer1_var}) \supset$
	$t_1 \ll \text{pointer}(\text{pointer1_var}))$

Repeatedly Skolemizing and flattening,

Expanding the definition of `■`,

Expanding the definition of `subterm`,

Instantiating quantified variables,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of **subterm_less.20**.

subterm_less.21:

$\{1\} \quad \forall (\text{reference1_var}: \text{Cpp_Type_}):$ $\quad (\forall (t_1: \text{Cpp_Type_}):$ $\quad \quad \text{subterm}(t_1, \text{reference1_var}) \wedge t_1 \neq \text{reference1_var} \supset t_1 \ll \text{reference1_var})$ $\quad \supset$ $\quad (\forall (t_1: \text{Cpp_Type_}):$ $\quad \quad \text{subterm}(t_1, \text{reference}(\text{reference1_var})) \wedge t_1 \neq \text{reference}(\text{reference1_var}) \supset$ $\quad \quad \quad t_1 \ll \text{reference}(\text{reference1_var}))$

Repeatedly Skolemizing and flattening,

Expanding the definition of \blacksquare ,

Expanding the definition of subterm,

Instantiating quantified variables,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of **subterm_less.21**.**subterm_less.22:**

$\{1\} \quad \forall (\text{class1_var}: \text{string}, \text{class2_var}: \text{bool}):$ $\quad \forall (t_1: \text{Cpp_Type_}):$ $\quad \quad \text{subterm}(t_1, \text{class}(\text{class1_var}, \text{class2_var})) \wedge t_1 \neq \text{class}(\text{class1_var}, \text{class2_var}) \supset$ $\quad \quad \quad t_1 \ll \text{class}(\text{class1_var}, \text{class2_var})$
--

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of **subterm_less.22**.**subterm_less.23:**

$\{1\} \quad \forall (\text{union1_var}: \text{string}):$ $\quad \forall (t_1: \text{Cpp_Type_}):$ $\quad \quad \text{subterm}(t_1, \text{union}(\text{union1_var})) \wedge t_1 \neq \text{union}(\text{union1_var}) \supset t_1 \ll \text{union}(\text{union1_var})$

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of **subterm_less.23**.**subterm_less.24:**

$\{1\} \quad \forall (\text{bitfield1_var}: \text{Cpp_Type_}, \text{bitfield2_var}: \text{nat}, \text{bitfield3_var}: \text{posnat}):$ $\quad (\forall (t_1: \text{Cpp_Type_}):$ $\quad \quad \text{subterm}(t_1, \text{bitfield1_var}) \wedge t_1 \neq \text{bitfield1_var} \supset t_1 \ll \text{bitfield1_var})$ $\quad \supset$ $\quad (\forall (t_1: \text{Cpp_Type_}):$ $\quad \quad \text{subterm}(t_1, \text{bitfield}(\text{bitfield1_var}, \text{bitfield2_var}, \text{bitfield3_var})) \wedge$ $\quad \quad \quad t_1 \neq \text{bitfield}(\text{bitfield1_var}, \text{bitfield2_var}, \text{bitfield3_var})$ $\quad \quad \quad \supset t_1 \ll \text{bitfield}(\text{bitfield1_var}, \text{bitfield2_var}, \text{bitfield3_var}))$
--

Repeatedly Skolemizing and flattening,

Expanding the definition of subterm,

Expanding the definition of \blacksquare ,

Instantiating quantified variables,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of **subterm_less.24**.

subterm_less.25:

$\{1\} \quad \forall (\text{enum1_var: string, enum2_var: Cpp_Type_}, \text{enum3_var: posnat, enum4_var: [below[enum3_var] \rightarrow \text{int}]}):$ $(\forall (t_1: \text{Cpp_Type_}): \text{subterm}(t_1, \text{enum2_var}) \wedge t_1 \neq \text{enum2_var} \supset t_1 \ll \text{enum2_var}) \supset$ $(\forall (t_1: \text{Cpp_Type_}):$ $\quad \text{subterm}(t_1, \text{enum}(\text{enum1_var}, \text{enum2_var}, \text{enum3_var}, \text{enum4_var})) \wedge$ $\quad t_1 \neq \text{enum}(\text{enum1_var}, \text{enum2_var}, \text{enum3_var}, \text{enum4_var})$ $\quad \supset t_1 \ll \text{enum}(\text{enum1_var}, \text{enum2_var}, \text{enum3_var}, \text{enum4_var}))$
--

Repeatedly Skolemizing and flattening,
 Expanding the definition of subterm,
 Expanding the definition of ■,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 Instantiating quantified variables,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 This completes the proof of **subterm_less.25**.

subterm_less.26:

$\{1\} \quad \forall (\text{pointer_to_member1_var: (class?)}, \text{pointer_to_member2_var: Cpp_Type_}):$ $((\forall (t_1: \text{Cpp_Type_}):$ $\quad \text{subterm}(t_1, \text{pointer_to_member1_var}) \wedge t_1 \neq \text{pointer_to_member1_var} \supset$ $\quad t_1 \ll \text{pointer_to_member1_var})$ \wedge $(\forall (t_1: \text{Cpp_Type_}):$ $\quad \text{subterm}(t_1, \text{pointer_to_member2_var}) \wedge t_1 \neq \text{pointer_to_member2_var} \supset$ $\quad t_1 \ll \text{pointer_to_member2_var}))$ \supset $(\forall (t_1: \text{Cpp_Type_}):$ $\quad \text{subterm}(t_1, \text{pointer_to_member}(\text{pointer_to_member1_var}, \text{pointer_to_member2_var})) \wedge$ $\quad t_1 \neq \text{pointer_to_member}(\text{pointer_to_member1_var}, \text{pointer_to_member2_var})$ $\quad \supset t_1 \ll \text{pointer_to_member}(\text{pointer_to_member1_var}, \text{pointer_to_member2_var}))$
--

Repeatedly Skolemizing and flattening,
 Expanding the definition of ■,
 Expanding the definition of subterm,
 Instantiating quantified variables,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 Instantiating the top quantifier in -2 with the terms: (t1!1),
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 This completes the proof of **subterm_less.26**.

subterm_less.27:

$\{1\} \quad \forall (\text{const1_var: Cpp_Type_}):$ $(\forall (t_1: \text{Cpp_Type_}): \text{subterm}(t_1, \text{const1_var}) \wedge t_1 \neq \text{const1_var} \supset t_1 \ll \text{const1_var}) \supset$ $(\forall (t_1: \text{Cpp_Type_}):$ $\quad \text{subterm}(t_1, \text{const}(\text{const1_var})) \wedge t_1 \neq \text{const}(\text{const1_var}) \supset$ $\quad t_1 \ll \text{const}(\text{const1_var}))$
--

Repeatedly Skolemizing and flattening,
 Expanding the definition of ■,
 Expanding the definition of subterm,
 Instantiating quantified variables,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `subterm_less.27`.

`subterm_less.28`:

$\{1\} \quad \forall (\text{volatile1_var}: \text{Cpp_Type_}):$ $\quad (\forall (t_1: \text{Cpp_Type_}):$ $\quad \quad \text{subterm}(t_1, \text{volatile1_var}) \wedge t_1 \neq \text{volatile1_var} \supset t_1 \ll \text{volatile1_var})$ $\quad \supset$ $\quad (\forall (t_1: \text{Cpp_Type_}):$ $\quad \quad \text{subterm}(t_1, \text{volatile}(\text{volatile1_var})) \wedge t_1 \neq \text{volatile}(\text{volatile1_var}) \supset$ $\quad \quad \quad t_1 \ll \text{volatile}(\text{volatile1_var}))$

Repeatedly Skolemizing and flattening,
Expanding the definition of \blacksquare ,
Expanding the definition of `subterm`,
Instantiating quantified variables,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `subterm_less.28`.
Q.E.D.

C.41.5 Cpp_Deep_Types.subterm_transitive

Terse proof for `subterm_transitive`.

`subterm_transitive`:

$\{1\} \quad \forall (t_1, t_2, t_3: \text{Cpp_Type_}): \text{subterm}(t_1, t_2) \wedge \text{subterm}(t_2, t_3) \supset \text{subterm}(t_1, t_3)$
--

Using lemma `wf_induction[Cpp_Type_, <<]`,
For the top quantifier in 1, we introduce Skolem constants: $(t1!1 \ t2!1 \ _)$,
Instantiating the top quantifier in -1 with the terms: $(\text{LAMBDA } (t3: \text{Cpp_Type_}): \text{subterm}(t1!1, t2!1) \text{ AND } \text{subterm}(t2!1, t3) \text{ IMPLIES } \text{subterm}(t1!1, t3))$,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Hiding formulas: 2,
Repeatedly Skolemizing and flattening,
Expanding the definition of `subterm`,
Splitting conjunctions,
we get 3 subgoals:

`subterm_transitive.1`:

$\{-1\} \quad t'_2 = x'$ $\{-2\} \quad \forall (y: \text{Cpp_Type_}):$ $\quad y \ll x' \supset \text{subterm}(t'_1, t'_2) \wedge \text{subterm}(t'_2, y) \supset \text{subterm}(t'_1, y)$ $\{-3\} \quad \text{subterm}(t'_1, t'_2)$
$\{1\} \quad \text{subterm}(t'_1, x')$

Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `subterm_transitive.1`.

`subterm_transitive.2`:

$\{-1\} \quad \forall (y: \text{Cpp_Type_}):$ $\quad y \ll x' \supset \text{subterm}(t'_1, t'_2) \wedge \text{subterm}(t'_2, y) \supset \text{subterm}(t'_1, y)$ $\{-2\} \quad \text{subterm}(t'_1, t'_2)$
$\{1\} \quad \text{TRUE}$ $\{2\} \quad \text{subterm}(t'_1, x')$

which is trivially true.

This completes the proof of `subterm_transitive.2`.

`subterm_transitive.3`:

<pre> {-1} ¬ uchar?(x') ∧ IF schar?(x') THEN FALSE ELSIF char?(x') THEN FALSE ELSIF short?(x') THEN FALSE ELSIF int?(x') THEN FALSE ELSIF long?(x') THEN FALSE ELSIF longlong?(x') THEN FALSE ELSIF ushort?(x') THEN FALSE ELSIF uint?(x') THEN FALSE ELSIF ulong?(x') THEN FALSE ELSIF ulonglong?(x') THEN FALSE ELSIF wchar_t?(x') THEN FALSE ELSIF bool?(x') THEN FALSE ELSIF float?(x') THEN FALSE ELSIF double?(x') THEN FALSE ELSIF longdouble?(x') THEN FALSE ELSIF void?(x') THEN FALSE ELSIF array?(x') THEN subterm(t'_2, typ(x')) ELSIF function?(x') THEN subterm(t'_2, ret_typ(x')) ∨ some[Cpp_Type_](λ (z: Cpp_Type_): subterm(t'_2, z))(args_typ(x')) ELSIF pointer?(x') THEN subterm(t'_2, typ(x')) ELSIF reference?(x') THEN subterm(t'_2, typ(x')) ELSIF class?(x') THEN FALSE ELSIF union?(x') THEN FALSE ELSIF bitfield?(x') THEN subterm(t'_2, typ(x')) ELSIF enum?(x') THEN subterm(t'_2, typ(x')) ELSIF pointer_to_member?(x') THEN subterm(t'_2, typ(x')) ∨ subterm(t'_2, member_typ(x')) ELSIF const?(x') THEN subterm(t'_2, typ(x')) ELSE subterm(t'_2, typ(x')) {-2} ∀ (y: Cpp_Type_): y ≪ x' ⊃ subterm(t'_1, t'_2) ∧ subterm(t'_2, y) ⊃ subterm(t'_1, y) {-3} subterm(t'_1, t'_2) </pre>	<pre> {1} subterm(t'_1, x') </pre>
---	-------------------------------------

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting, we get 7 subgoals:

`subterm_transitive.3.1`:

<pre> {-1} array?(x') {-2} subterm(t'_2, typ(x')) {-3} ∀ (y: Cpp_Type_): y ≪ x' ⊃ subterm(t'_2, y) ⊃ subterm(t'_1, y) {-4} subterm(t'_1, t'_2) </pre>	<pre> {1} subterm(t'_1, x') </pre>
---	-------------------------------------

Instantiating the top quantifier in -3 with the terms: `(typ(x!1))`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting, we get 2 subgoals:

subterm_transitive.3.1.1:

{-1}	array?(x')
{-2}	subterm(t'_2 , typ(x'))
{-3}	subterm(t'_1 , typ(x'))
{-4}	subterm(t'_1 , t'_2)
{1}	subterm(t'_1 , x')

Expanding the definition of subterm,
which is trivially true.

This completes the proof of **subterm_transitive.3.1.1**.

subterm_transitive.3.1.2:

{-1}	array?(x')
{-2}	subterm(t'_2 , typ(x'))
{-3}	subterm(t'_1 , t'_2)
{1}	typ(x') \ll x'
{2}	subterm(t'_1 , x')

Expanding the definition of \blacksquare ,
which is trivially true.

This completes the proof of **subterm_transitive.3.1.2**.

subterm_transitive.3.2:

{-1}	function?(x')
{-2}	subterm(t'_2 , ret_typ(x'))
{-3}	$\forall (y: \text{Cpp_Type_}): y \ll x' \supset \text{subterm}(t'_2, y) \supset \text{subterm}(t'_1, y)$
{-4}	subterm(t'_1 , t'_2)
{1}	subterm(t'_1 , x')

Instantiating the top quantifier in -3 with the terms: (ret_typ(x!1)),

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of **subterm_transitive.3.2**.

subterm_transitive.3.3:

{-1}	function?(x')
{-2}	some[Cpp_Type_]($\lambda (z: \text{Cpp_Type_}): \text{subterm}(t'_2, z)(\text{args_typ}(x'))$)
{-3}	$\forall (y: \text{Cpp_Type_}): y \ll x' \supset \text{subterm}(t'_2, y) \supset \text{subterm}(t'_1, y)$
{-4}	subterm(t'_1 , t'_2)
{1}	subterm(t'_1 , x')

Expanding the definition of subterm,

Applying disjunctive simplification to flatten sequent,

Rewriting using some_iff_exists, matching in *,

Rewriting using some_iff_exists, matching in *,

Repeatedly Skolemizing and flattening,

Instantiating quantified variables,

Instantiating quantified variables,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Expanding the definition of \blacksquare ,

Applying disjunctive simplification to flatten sequent,

Rewriting using some_iff_exists, matching in *,

Using lemma forall_not[Cpp_Type_],

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Instantiating the top quantifier in -1 with the terms: (x!1),

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

C Proof scripts

we get 2 subgoals:

subterm_transitive.3.3.1:

{-1}	function?(x')
{-2}	member(t' , args_typ(x'))
{-3}	subterm(t'_2 , t')
{-4}	subterm(t'_1 , t'_2)
{1}	$t' = x'$
{2}	$t' \ll x'$
{3}	$\exists (t: \text{Cpp_Type_}): \text{member}(t, \text{args_typ}(x')) \wedge (t' = t \vee t' \ll t)$
{4}	$t' = \text{ret_typ}(x')$
{5}	$t' \ll \text{ret_typ}(x')$
{6}	subterm(t'_1 , t')
{7}	$t'_1 = x'$
{8}	subterm(t'_1 , ret_typ(x'))

Using lemma subterm_less,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Instantiating quantified variables,

This completes the proof of **subterm_transitive.3.3.1**.

subterm_transitive.3.3.2:

{-1}	function?(x')
{-2}	member(t' , args_typ(x'))
{-3}	subterm(t'_2 , t')
{-4}	subterm(t'_1 , t'_2)
{1}	member(x' , args_typ(x'))
{2}	$\exists (t: \text{Cpp_Type_}): \text{member}(t, \text{args_typ}(x')) \wedge (t' = t \vee t' \ll t)$
{3}	$t' = \text{ret_typ}(x')$
{4}	$t' \ll \text{ret_typ}(x')$
{5}	subterm(t'_1 , t')
{6}	$t'_1 = x'$
{7}	subterm(t'_1 , ret_typ(x'))

Instantiating the top quantifier in 2 with the terms: (t!1),

which is trivially true.

This completes the proof of **subterm_transitive.3.3.2**.

subterm_transitive.3.4:

{-1}	subterm(t'_2 , typ(x'))
{-2}	pointer?(x')
{-3}	$\forall (y: \text{Cpp_Type_}): y \ll x' \supset \text{subterm}(t'_2, y) \supset \text{subterm}(t'_1, y)$
{-4}	subterm(t'_1 , t'_2)
{1}	subterm(t'_1 , x')

Instantiating the top quantifier in -3 with the terms: (typ(x!1)),

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of **subterm_transitive.3.4**.

subterm_transitive.3.5:

{-1}	subterm(t'_2 , typ(x'))
{-2}	reference?(x')
{-3}	$\forall (y: \text{Cpp_Type_}): y \ll x' \supset \text{subterm}(t'_2, y) \supset \text{subterm}(t'_1, y)$
{-4}	subterm(t'_1 , t'_2)
{1}	subterm(t'_1 , x')

Instantiating the top quantifier in -3 with the terms: (typ(x!1)),

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `subterm_transitive.3.5`.

`subterm_transitive.3.6`:

{-1}	subterm(t'_2 , typ(x'))
{-2}	$\forall (y: \text{Cpp_Type_}): y \ll x' \supset \text{subterm}(t'_2, y) \supset \text{subterm}(t'_1, y)$
{-3}	subterm(t'_1 , t'_2)
{1}	uchar?(x')
{2}	schar?(x')
{3}	char?(x')
{4}	short?(x')
{5}	int?(x')
{6}	long?(x')
{7}	longlong?(x')
{8}	ushort?(x')
{9}	uint?(x')
{10}	ulong?(x')
{11}	ulonglong?(x')
{12}	wchar_t?(x')
{13}	bool?(x')
{14}	float?(x')
{15}	double?(x')
{16}	longdouble?(x')
{17}	void?(x')
{18}	function?(x')
{19}	class?(x')
{20}	union?(x')
{21}	subterm(t'_1 , x')

Instantiating the top quantifier in -2 with the terms: (typ(x!1)),

Expanding the definition of \blacksquare ,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Expanding the definition of subterm,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `subterm_transitive.3.6`.

`subterm_transitive.3.7`:

{-1}	pointer_to_member?(x')
{-2}	subterm(t'_2 , member_typ(x'))
{-3}	$\forall (y: \text{Cpp_Type_}): y \ll x' \supset \text{subterm}(t'_2, y) \supset \text{subterm}(t'_1, y)$
{-4}	subterm(t'_1 , t'_2)
{1}	subterm(t'_1 , x')

Instantiating the top quantifier in -3 with the terms: (member_typ(x!1)),

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `subterm_transitive.3.7`.

Q.E.D.

C.41.6 Cpp_Deep_Types.c_TCC1

Terse proof for `c_TCC1`.

c_TCC1:

```
{1}  ∃ (p: PRED[Cpp_Type_], typ: Cpp_Type_):
      const?(typ) ∧ ¬ p(typ) ⊃
      array?(typ) ∨
      pointer?(typ) ∨
      reference?(typ) ∨
      bitfield?(typ) ∨
      enum?(typ) ∨ pointer_to_member?(typ) ∨ const?(typ) ∨ volatile?(typ)
```

Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of c_TCC1.
 Q.E.D.

C.41.7 Cpp_Deep_Types.v_TCC1

Terse proof for v_TCC1.

v_TCC1:

```
{1}  ∃ (p: PRED[Cpp_Type_], typ: Cpp_Type_):
      volatile?(typ) ∧ ¬ p(typ) ⊃
      array?(typ) ∨
      pointer?(typ) ∨
      reference?(typ) ∨
      bitfield?(typ) ∨
      enum?(typ) ∨ pointer_to_member?(typ) ∨ const?(typ) ∨ volatile?(typ)
```

Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of v_TCC1.
 Q.E.D.

C.41.8 Cpp_Deep_Types.cv_TCC1

Terse proof for cv_TCC1.

cv_TCC1:

```
{1}  ∃ (p: PRED[Cpp_Type_], typ: Cpp_Type_):
      const?(typ) ∧ ¬ v(p)(typ) ∧ ¬ c(p)(typ) ⊃
      array?(typ) ∨
      pointer?(typ) ∨
      reference?(typ) ∨
      bitfield?(typ) ∨
      enum?(typ) ∨ pointer_to_member?(typ) ∨ const?(typ) ∨ volatile?(typ)
```

Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of cv_TCC1.
 Q.E.D.

C.41.9 Cpp_Deep_Types.cv_TCC2

Terse proof for cv_TCC2.

cv_TCC2:

```
{1}  ∀ (p: PRED[Cpp_Type_], typ: Cpp_Type_):
      volatile?(typ(typ)) ∧ const?(typ) ∧ ¬ v(p)(typ) ∧ ¬ c(p)(typ) ⊃
      array?(typ(typ)) ∨
      pointer?(typ(typ)) ∨
      reference?(typ(typ)) ∨
      bitfield?(typ(typ)) ∨
      enum?(typ(typ)) ∨
      pointer_to_member?(typ(typ)) ∨ const?(typ(typ)) ∨ volatile?(typ(typ))
```

Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of cv_TCC2.
 Q.E.D.

C.41.10 Cpp_Deep_Types.cv_implied

Terse proof for cv_implied.

cv_implied:

```
{1}  ∀ (p1, p2: PRED[Cpp_Type_], typ: Cpp_Type_):
      (p1 ⊆ p2) ∧ cv(p1)(typ) ⊃ cv(p2)(typ)
```

Repeatedly Skolemizing and flattening,
 Expanding the definition of subset?,
 Expanding the definition of member,
 Expanding the definition of cv,
 Expanding the definition of c,
 Expanding the definition of v,
 Applying disjunctive simplification to flatten sequent,
 Splitting conjunctions,
 we get 5 subgoals:

cv_implied.1:

```
{-1}  p'1(typ')
{-2}  ∀ (x: Cpp_Type_): p'1(x) ⇒ p'2(x)
{1}   p'2(typ')
{2}   (const?(typ') ∧ p'2(typ(typ')))
{3}   p'2(typ')
{4}   (volatile?(typ') ∧ p'2(typ(typ')))
{5}   (const?(typ') ∧ volatile?(typ(typ'))) ∧ p'2(typ(typ(typ'))))
```

Instantiating quantified variables,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 This completes the proof of cv_implied.1.

cv_implied.2:

```
{-1}  (const?(typ') ∧ p'1(typ(typ')))
{-2}  ∀ (x: Cpp_Type_): p'1(x) ⇒ p'2(x)
{1}   p'2(typ')
{2}   (const?(typ') ∧ p'2(typ(typ')))
{3}   p'2(typ')
{4}   (volatile?(typ') ∧ p'2(typ(typ')))
{5}   (const?(typ') ∧ volatile?(typ(typ'))) ∧ p'2(typ(typ(typ'))))
```


C Proof scripts

Instantiating quantified variables,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `cv_implied.2`.

`cv_implied.3`:

{-1}	$p'_1(\text{typ}')$
{-2}	$\forall (x: \text{Cpp_Type_}): p'_1(x) \Rightarrow p'_2(x)$
<hr/>	
{1}	$p'_2(\text{typ}')$
{2}	$(\text{const}?(\text{typ}') \wedge p'_2(\text{typ}(\text{typ}')))$
{3}	$p'_2(\text{typ}')$
{4}	$(\text{volatile}?(\text{typ}') \wedge p'_2(\text{typ}(\text{typ}')))$
{5}	$(\text{const}?(\text{typ}') \wedge \text{volatile}?(\text{typ}(\text{typ}')) \wedge p'_2(\text{typ}(\text{typ}(\text{typ}'))))$

Instantiating quantified variables,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `cv_implied.3`.

`cv_implied.4`:

{-1}	$(\text{volatile}?(\text{typ}') \wedge p'_1(\text{typ}(\text{typ}')))$
{-2}	$\forall (x: \text{Cpp_Type_}): p'_1(x) \Rightarrow p'_2(x)$
<hr/>	
{1}	$p'_2(\text{typ}')$
{2}	$(\text{const}?(\text{typ}') \wedge p'_2(\text{typ}(\text{typ}')))$
{3}	$p'_2(\text{typ}')$
{4}	$(\text{volatile}?(\text{typ}') \wedge p'_2(\text{typ}(\text{typ}')))$
{5}	$(\text{const}?(\text{typ}') \wedge \text{volatile}?(\text{typ}(\text{typ}')) \wedge p'_2(\text{typ}(\text{typ}(\text{typ}'))))$

Instantiating quantified variables,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `cv_implied.4`.

`cv_implied.5`:

{-1}	$(\text{const}?(\text{typ}') \wedge \text{volatile}?(\text{typ}(\text{typ}')) \wedge p'_1(\text{typ}(\text{typ}(\text{typ}'))))$
{-2}	$\forall (x: \text{Cpp_Type_}): p'_1(x) \Rightarrow p'_2(x)$
<hr/>	
{1}	$p'_2(\text{typ}')$
{2}	$(\text{const}?(\text{typ}') \wedge p'_2(\text{typ}(\text{typ}')))$
{3}	$p'_2(\text{typ}')$
{4}	$(\text{volatile}?(\text{typ}') \wedge p'_2(\text{typ}(\text{typ}')))$
{5}	$(\text{const}?(\text{typ}') \wedge \text{volatile}?(\text{typ}(\text{typ}')) \wedge p'_2(\text{typ}(\text{typ}(\text{typ}'))))$

Instantiating quantified variables,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `cv_implied.5`.

Q.E.D.

C.41.11 Cpp_Deep_Types.pod?_TCC1

Terse proof for `pod?_TCC1`.

pod?_TCC1:

```
{1}  ∀ (t: Cpp_Type_):
      array?(t) ∧
      ¬ (class?(t) ∧ (pod_struct?(t) ∨ pod_union?(t))) ∧ ¬ scalar?(t)
      ⊃
      array?(t) ∨
      pointer?(t) ∨
      reference?(t) ∨
      bitfield?(t) ∨ enum?(t) ∨ pointer_to_member?(t) ∨ const?(t) ∨ volatile?(t)
```

Repeatedly Skolemizing and flattening,
This completes the proof of pod?_TCC1.
Q.E.D.

C.41.12 Cpp_Deep_Types.uidt_void_TCC1

Terse proof for uidt_void_TCC1.

uidt_void_TCC1:

```
{1}  uninterpreted_data_type?((#size := 0, valid? := λ (l: list[Byte], a: Ad-
      dress): FALSE#))
```

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of uidt_void_TCC1.
Q.E.D.

C.42 Proofs for Cpp_Examples (cpp-examples.pvs)

C.42.1 Cpp_Examples.spec01

Terse proof for spec01.

spec01:

```
{1}  Valid(STATES, skip[State], STATES)
```

Installing automatic rewrites from: valid Valid PRECONDITION POSTCONDITION
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Repeatedly Skolemizing and flattening,
Installing automatic rewrites from: literal id list_props.member postinc postdec not_exp
preint predec IntExpressions.* IntExpressions.div IntExpressions.rem IntExpressions.- IntEx-
pressions.■ IntExpressions.■ IntExpressions.< IntExpressions.> IntExpressions.<= IntExpres-
sions.>= EqualityExpressions.== not_equal and_exp or_exp assign assign_times IntExpressions.+
assign_plus assign_minus assign_div assign_rem comma assign_bsr assign_bsl deref IntExpres-
sions2.+ IntExpressions2.- Cpp_pointer.+ Cpp_pointer.- ptr_addr_ptr_of pointer_add_n_pointer_minus
pointer_add_difference Aligned_Ptr plain_memory_write_ok_single plain_memory_write_ok_q_expr
plain_memory_write_ok_q_stmt in_blessed_memory_rw_ro pm_q_prop_ok pm_q_prop_single_read
pm_q_prop_single_write pm_q_prop_read_write_single pm_q_prop_read_write_other_single pm_q_prop_read_read_single
plain_memory_read_write_other_single_data plain_memory_read_read_single_data plain_memory_read_write_single_data
pm_q_prop_read_ok_expr pm_q_prop_read_ok_stmt pm_q_prop_read_pm_q_prop_expr pm_q_prop_read_pm_q_prop_stmt
pm_q_prop_read_write_q_expr pm_q_prop_read_write_q_stmt plain_memory_read_write_q_data_expr

plain_memory_read_write_q_data_stmt pm_q_prop_read_write_other_q_expr pm_q_prop_read_write_other_q_stmt
 pm_q_prop_read_read_q_expr pm_q_prop_read_read_q_stmt plain_memory_read_write_other_q_data_expr
 plain_memory_read_write_other_q_data_stmt plain_memory_read_read_q_data_expr plain_memory_read_read_q_data_stmt
 pm_q_prop_ok_result_single pm_q_prop_ok_result_q_expr pm_q_prop_ok_result_q_stmt pm_q_prop_read_ok_result_single
 pm_q_prop_read_ok_result_q_expr pm_q_prop_read_ok_result_q_stmt pm_q_prop_e2s pm_q_prop_stmt_e2s
 pm_q_prop_read_e2s pm_q_prop_read_stmt_e2s if_else switch member break_break_stmt break_break_catch_continue
 break_break_catch_default break_break_lift_stmt break_break_case break_break_default break_break
 stmt_break_break break_stmt_break break_stmt_catch_continue break_stmt_catch_default break_stmt_lift
 break_stmt_case break_stmt_default continue_break_stmt continue_break_catch_continue con-
 tinue_break_catch_default continue_break_lift_stmt continue_break_case continue_break_default con-
 tinue_continue stmt_continue_continue continue_stmt_continue continue_stmt_catch_continue con-
 tinue_stmt_catch_default continue_stmt_lift continue_stmt_case continue_stmt_default switch_stmt
 stmt_switch_stmt switch_case_taken stmt_switch_case_taken switch_case_not_taken stmt_switch_case_not_taken
 switch_default stmt_switch_default switch_catch_break stmt_switch_catch_break switch_catch_continue
 stmt_switch_catch_continue switch_catch_default stmt_switch_catch_default default_stmt stmt_default_stmt
 default_case stmt_default_case default_default stmt_default_default default_catch_break stmt_default_catch_break
 default_catch_continue stmt_default_catch_continue default_catch_default stmt_default_catch_default
 return_void_result return_ex_result stmt_remove_catch_default stmt_remove_case stmt_remove_default
 stmt_remove_catch_continue stmt_remove_catch_break break_break_stmt_expr break_break_catch_continue_expr
 break_break_catch_default_expr break_break_lift_stmt_expr break_break_case_expr break_break_default_expr
 continue_break_stmt_expr continue_break_catch_continue_expr continue_break_catch_default_expr
 continue_break_lift_stmt_expr continue_break_case_expr continue_break_default_expr skip_ok
 skip_state skip_elimination stmt_skip_elimination stmt_ok_lift stmt_ok_lift_expr e2s_ok stmt_e2s_ok
 e2s_state stmt_e2s_state ok_result_fexpr ok_result_fstmt e2s_merge stmt_e2s_merge e2s_expr
 stmt_e2s_expr while_unroll stmt_while_unroll iterate_while_ok while_hang while_unroll_expr
 while_unroll_lexpr stmt_while_unroll_expr stmt_while_unroll_lexpr while_unroll_0 do_as_while
 do_while_unroll for_as_while for_unroll composition_assoc_rewrite_stmt_ok forward_s_c_s_of_s_expr
 comp_eval_if_ok_fstmt stmt_eval_if_ok_fstmt comp_eval_if_ok_fstmt_cstmt stmt_eval_if_ok_fstmt_cstmt
 composition_assoc_rewrite_1 composition_assoc_rewrite_2 composition_assoc_rewrite_3 com-
 position_assoc_rewrite_4 composition_assoc_rewrite_5 composition_assoc_rewrite_6 composi-
 tion_assoc_rewrite_7 composition_assoc_rewrite_8 composition_assoc_rewrite_expr_1 composi-
 tion_assoc_rewrite_expr_2 composition_assoc_rewrite_expr_3 composition_assoc_rewrite_expr_4 composi-
 tion_assoc_rewrite_expr_5 composition_assoc_rewrite_expr_6 composition_assoc_rewrite_stmt_ok compo-
 sition_assoc_rewrite_stmt_ok_expr composition_assoc_rewrite_stmt_ok_expr_lexpr comp_eval_if_ok_fstmt
 comp_eval_if_ok_fstmt_expr comp_eval_if_ok_fstmt_cstmt_expr stmt_eval_if_ok_fstmt_expr stmt_eval_if_ok_fstmt_cst-
 comp_eval_if_ok_fexpr_expr stmt_eval_if_ok_fexpr expr_eval_if_ok_fexpr composition_assoc_expression_rewrite
 composition_assoc_expression_rewrite_stmt stmt_eval_if_ok_fexpr_expr expr_eval_if_ok_fexpr_expr
 ok_result_fexpr ok_result_fstmt allocate_stack deallocate_stack with_new_stackvar ok_result_ok
 ok_result_data ok_result_q_ok ok_result_q_state ok_result_q_data ok_result_elimination_1 ok_result_elimination_2
 ok_result_state l2r n2b a2b comp_simple_expr_simple_expr_ok comp_simple_expr_simple_expr_data
 comp_simple_expr_simple_expr_state comp_simple_stmt_simple_expr_ok comp_simple_stmt_simple_expr_data
 comp_simple_stmt_simple_expr_state

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `spec01`.

Q.E.D.

C.42.2 Cpp_Examples.PRECONDITION_TCC1

Terse proof for `PRECONDITION_TCC1`.

`PRECONDITION_TCC1` :

$\{1\} \quad \forall (s: \text{State}): \text{pm}'\text{states}(s) \supset \text{Cpp_Type?}(\text{int}) \wedge \text{cv}(\text{non_bool_Integral_Enum?})(\text{int})$

Repeatedly Skolemizing and flattening,
Applying propositional simplification,
we get 2 subgoals:

PRECONDITION_TCC1.1:

{-1}	pm' states(<i>s'</i>)
{1}	Cpp_Type?(int)

Installing automatic rewrites from: no_pointers_to_bitfield? no_pointers_to_references?
no_cv_references? no_reference_to_reference? no_reference_to_bitfields? no_pointer_to_member_to_reference?
no_pointer_to_member_to_cv_void? no_cv_void_parameter? no_array_of_references? no_array_of_bitfields?
no_array_of_cv_void? no_array_of_function? no_array_of_abstract_class? no_pointer_or_ref_to_incomplete_array_parameter?
no_array_return_type? no_function_return_type? bitfield_underlying_integral_or_enum_type? cv_array?
enum_underlying_integral? enum_constants? const_volatile? const_stutter? volatile_stutter?
no_cv_class? no_cv_union? no_cv_function? no_reference_to_void? no_cv_void?

Expanding the definition of Cpp_Type?,
Expanding the definition of subterm,
Repeatedly Skolemizing and flattening,
Replacing using formula -1,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of PRECONDITION_TCC1.1.

PRECONDITION_TCC1.2:

{-1}	pm' states(<i>s'</i>)
{1}	cv(non_boolIntegralEnum?)(int)

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of PRECONDITION_TCC1.2.
Q.E.D.

C.42.3 Cpp_Examples.PRECONDITION_TCC2

Terse proof for PRECONDITION_TCC2.

PRECONDITION_TCC2:

{1}	$\forall (s: \text{State}):$ $\text{in_blessed_memory?}(\text{dt_bool}, c, \text{pm}'\text{rw_addr}) \wedge$ $\text{in_blessed_memory?}(\text{dt}(\text{int}), m, \text{pm}'\text{rw_addr}) \wedge$ $\text{in_blessed_memory?}(\text{dt}(\text{int}), n, \text{pm}'\text{rw_addr}) \wedge \text{pm}'\text{states}(s)$ $\supset \text{Cpp_Type?}(\text{char}) \wedge \text{cv}(\text{non_bool_integral_enum?})(\text{char})$
-----	---

Repeatedly Skolemizing and flattening,
Applying propositional simplification,
we get 2 subgoals:

PRECONDITION_TCC2.1:

{-1}	in_blessed_memory?(dt_bool, <i>c</i> , pm'rw_addr)
{-2}	in_blessed_memory?(dt(int), <i>m</i> , pm'rw_addr)
{-3}	in_blessed_memory?(dt(int), <i>n</i> , pm'rw_addr)
{-4}	pm' states(<i>s'</i>)
{1}	Cpp_Type?(char)

Installing automatic rewrites from: no_pointers_to_bitfield? no_pointers_to_references?
no_cv_references? no_reference_to_reference? no_reference_to_bitfields? no_pointer_to_member_to_reference?
no_pointer_to_member_to_cv_void? no_cv_void_parameter? no_array_of_references? no_array_of_bitfields?

no_array_of_cv_void? no_array_of_function? no_array_of_abstract_class? no_pointer_or_ref_to_incomplete_array_par
no_array_return_type? no_function_return_type? bitfield_underlying_integral_or_enum_type? cv_array?
enum_underlying_integral? enum_constants? const_volatile? const_stutter? volatile_stutter?
no_cv_class? no_cv_union? no_cv_function? no_reference_to_void? no_cv_void?

Expanding the definition of Cpp_Type?,

Expanding the definition of subterm,

Repeatedly Skolemizing and flattening,

Replacing using formula -1,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of PRECONDITION_TCC2.1.

PRECONDITION_TCC2.2:

{-1}	in_blessed_memory?(dt_bool, c, pm'rw_addr)
{-2}	in_blessed_memory?(dt(int), m, pm'rw_addr)
{-3}	in_blessed_memory?(dt(int), n, pm'rw_addr)
{-4}	pm'states(s')
{1}	cv(non_bool_integral_enum?)(char)

Keeping (1) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of PRECONDITION_TCC2.2.

Q.E.D.

C.42.4 Cpp_Examples.PRECONDITION_TCC3

Terse proof for PRECONDITION_TCC3.

PRECONDITION_TCC3:

{1}	$\forall (s: \text{State}):$ in_blessed_memory?(dt(char), b, pm'rw_addr) \wedge in_blessed_memory?(dt(char), a, pm'rw_addr) \wedge in_blessed_memory?(dt_bool, c, pm'rw_addr) \wedge in_blessed_memory?(dt(int), m, pm'rw_addr) \wedge in_blessed_memory?(dt(int), n, pm'rw_addr) \wedge pm'states(s) \supset Cpp_Type?(char)
-----	---

Repeatedly Skolemizing and flattening,

Using lemma PRECONDITION_TCC2,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of PRECONDITION_TCC3.

Q.E.D.

C.42.5 Cpp_Examples.PRECONDITION_TCC4

Terse proof for PRECONDITION_TCC4.

PRECONDITION_TCC4:

```
{1}  ∀ (s: State):
      blocks_disjoint?(b, size_of(char), a, size_of(char)) ∧
      blocks_disjoint?(a, size_of(char), b, size_of(char)) ∧
      in_blessed_memory?(dt(char), b, pm'rw_addr) ∧
      in_blessed_memory?(dt(char), a, pm'rw_addr) ∧
      in_blessed_memory?(dt_bool, c, pm'rw_addr) ∧
      in_blessed_memory?(dt(int), m, pm'rw_addr) ∧
      in_blessed_memory?(dt(int), n, pm'rw_addr) ∧ pm'states(s)
      ⊃ Cpp_Type?(int)
```

Repeatedly Skolemizing and flattening,
 Using lemma PRECONDITION_TCC1,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 This completes the proof of PRECONDITION_TCC4.
 Q.E.D.

C.42.6 Cpp_Examples.POSTCONDITION_TCC1

Terse proof for POSTCONDITION_TCC1.

POSTCONDITION_TCC1:

```
{1}  ∀ (s: State): Cpp_Type?(int) ∧ cv(non_bool_integral_enum?)(int)
```

Repeatedly Skolemizing and flattening,
 Applying propositional simplification,
 we get 2 subgoals:

POSTCONDITION_TCC1.1:

```
{1}  Cpp_Type?(int)
```

Installing automatic rewrites from: no_pointers_to_bitfield? no_pointers_to_references?
 no_cv_references? no_reference_to_reference? no_reference_to_bitfields? no_pointer_to_member_to_reference?
 no_pointer_to_member_to_cv_void? no_cv_void_parameter? no_array_of_references? no_array_of_bitfields?
 no_array_of_cv_void? no_array_of_function? no_array_of_abstract_class? no_pointer_or_ref_to_incomplete_array_parameter?
 no_array_return_type? no_function_return_type? bitfield_underlying_integral_or_enum_type? cv_array?
 enum_underlying_integral? enum_constants? const_volatile? const_stutter? volatile_stutter?
 no_cv_class? no_cv_union? no_cv_function? no_reference_to_void? no_cv_void?

Expanding the definition of Cpp_Type?,
 Expanding the definition of subterm,
 Repeatedly Skolemizing and flattening,
 Replacing using formula -1,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 This completes the proof of POSTCONDITION_TCC1.1.

POSTCONDITION_TCC1.2:

```
{1}  cv(non_bool_integral_enum?)(int)
```

Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of POSTCONDITION_TCC1.2.
 Q.E.D.

C.42.7 Cpp_Examples.spec02_TCC1

Terse proof for spec02_TCC1.

spec02_TCC1:

{1} plain_memory?(pm) \supset range(int)(42)

Applying disjunctive simplification to flatten sequent,
 Expanding the definition of range,
 Expanding the definition of range_integral,
 Rewriting using int_binary, matching in *,
 Using lemma min_int,
 Using lemma max_int,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 This completes the proof of spec02_TCC1.
 Q.E.D.

C.42.8 Cpp_Examples.spec02

Terse proof for spec02.

spec02:

{1} plain_memory?(pm) \supset
 valid(PRECONDITION, e2s(assign(pm, dt(int))(id(n), literal(42))), POSTCONDI-
 TION)

Installing automatic rewrites from: valid Valid PRECONDITION POSTCONDITION
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 Repeatedly Skolemizing and flattening,
 Installing automatic rewrites from: literal id list_props.member postinc postdec not_exp preint predec
 IntExpressions.* IntExpressions.div IntExpressions.rem IntExpressions.- IntExpressions.■ IntExpressions-
 IntExpressions.■ IntExpressions.< IntExpressions.> IntExpressions.<= IntExpressions.>= EqualityExpressions.
 == not_equal and_exp or_exp assign assign_times IntExpressions.+ assign_plus assign_minus
 assign_div assign_rem comma assign_bsr assign_bsl deref IntExpressions2.+ IntExpressions2.-
 Cpp_pointer.+ Cpp_pointer.- ptr_addr_ptr_of pointer_add_n_pointer_minus pointer_add_difference
 Aligned_Ptr
 Installing automatic rewrites from: plain_memory_write_ok_single plain_memory_write_ok_q_expr
 plain_memory_write_ok_q_stmt in_blessed_memory_rw_ro pm_q_prop_ok pm_q_prop_single_read
 pm_q_prop_single_write pm_q_prop_read_write_single pm_q_prop_read_write_other_single pm_q_prop_read_read_sing
 plain_memory_read_write_other_single_data plain_memory_read_read plain_memory_read_write_res
 pm_q_prop_read_ok_expr pm_q_prop_read_ok_stmt pm_q_prop_read_pm_q_prop_expr pm_q_prop_read_pm_q_prop_s
 pm_q_prop_read_write_q_expr pm_q_prop_read_write_q_stmt plain_memory_read_write_q_data_expr
 plain_memory_read_write_q_data_stmt pm_q_prop_read_write_other_q_expr pm_q_prop_read_write_other_q_stmt
 pm_q_prop_read_read_q_expr pm_q_prop_read_read_q_stmt plain_memory_read_write_other_q_data_expr
 plain_memory_read_write_other_q_data_stmt plain_memory_read_read_q_data_expr plain_memory_read_read_q_data
 pm_q_prop_ok_result_single pm_q_prop_ok_result_q_expr pm_q_prop_ok_result_q_stmt pm_q_prop_read_ok_result_s
 pm_q_prop_read_ok_result_q_expr pm_q_prop_read_ok_result_q_stmt pm_q_prop_e2s pm_q_prop_stmt_e2s
 pm_q_prop_read_e2s pm_q_prop_read_stmt_e2s
 Installing automatic rewrites from: if_else switch member break_catch_break break_break_stmt
 break_break_catch_continue break_break_catch_default break_break_lift_stmt break_break_case
 break_break_default break_break stmt_break_break break_stmt_break break_stmt_catch_continue
 break_stmt_catch_default break_stmt_lift break_stmt_case break_stmt_default continue_catch_continue
 continue_break_stmt continue_break_catch_continue continue_break_catch_default continue_break_lift_stmt

continue_break_case continue_break_default continue_continue stmt_continue_continue continue_stmt_continue continue_stmt_continue continue_stmt_catch_continue continue_stmt_catch_default continue_stmt_lift continue_stmt_case continue_stmt_default switch_stmt stmt_switch_stmt switch_case_taken stmt_switch_case_taken switch_case_not_taken stmt_switch_case_not_taken switch_default stmt_switch_default switch_catch_break stmt_switch_catch_break switch_catch_continue stmt_switch_catch_continue switch_catch_default stmt_switch_catch_default default_stmt stmt_default_stmt default_case stmt_default_case default_default stmt_default_default default_catch_break stmt_default_catch_break default_catch_continue stmt_default_catch_continue default_catch_default stmt_default_catch_default return_void_result return_ex_result stmt_remove_catch_default stmt_remove_case stmt_remove_default stmt_remove_catch_continue stmt_remove_catch_break break_break_stmt_expr break_break_catch_continue_expr break_break_catch_default_expr break_break_lift_stmt_expr break_break_case_expr break_break_default_expr continue_break_stmt_expr continue_break_catch_continue_expr continue_break_catch_default_expr continue_break_lift_stmt_expr continue_break_case_expr continue_break_default_expr

Installing automatic rewrites from: skip_ok skip_state skip_elimination stmt_skip_elimination stmt_ok_lift stmt_ok_lift_expr e2s_ok stmt_e2s_ok e2s_state stmt_e2s_state ok_result_fexpr ok_result_fstmt e2s_merge stmt_e2s_merge e2s_expr stmt_e2s_expr

Installing automatic rewrites from: while_unroll stmt_while_unroll iterate_while_ok while_hang while_unroll_expr while_unroll_lexpr stmt_while_unroll_expr stmt_while_unroll_lexpr while_unroll_0 do_as_while do_while_unroll for_as_while for_unroll

Installing automatic rewrites from: composition_assoc_rewrite_stmt_ok forward_s_c_s_of_s_expr comp_eval_if_ok_fstmt stmt_eval_if_ok_fstmt comp_eval_if_ok_fstmt_cstmt stmt_eval_if_ok_fstmt_cstmt composition_assoc_rewrite_1 composition_assoc_rewrite_2 composition_assoc_rewrite_3 composition_assoc_rewrite_4 composition_assoc_rewrite_5 composition_assoc_rewrite_6 composition_assoc_rewrite_7 composition_assoc_rewrite_8 composition_assoc_rewrite_expr_1 composition_assoc_rewrite_expr_2 composition_assoc_rewrite_expr_3 composition_assoc_rewrite_expr_4 composition_assoc_rewrite_expr_5 composition_assoc_rewrite_expr_6 composition_assoc_rewrite_stmt_ok composition_assoc_rewrite_stmt_ok_expr composition_assoc_rewrite_stmt_ok_expr_lexpr comp_eval_if_ok_fstmt comp_eval_if_ok_fstmt_expr comp_eval_if_ok_fstmt_cstmt_expr stmt_eval_if_ok_fstmt_expr stmt_eval_if_ok_fstmt_cstmt_expr comp_eval_if_ok_fexpr_expr stmt_eval_if_ok_fexpr expr_eval_if_ok_fexpr composition_assoc_expression_rewrite composition_assoc_expression_rewrite_stmt stmt_eval_if_ok_fexpr_expr expr_eval_if_ok_fexpr_expr ok_result_fexpr ok_result_fstmt composition_assoc_rewrite_stmt_ok_cstmt

Installing automatic rewrites from: allocate_stack deallocate_stack with_new_stackvar

Installing automatic rewrites from: ok_result_ok ok_result_data ok_result_q_ok ok_result_q_state ok_result_q_data ok_result_elimination_1 ok_result_elimination_2 ok_result_state

Installing automatic rewrites from: l2r n2b a2b

Installing automatic rewrites from: comp_simple_expr_simple_expr_ok comp_simple_expr_simple_expr_data comp_simple_expr_simple_expr_state comp_simple_stmt_simple_expr_ok comp_simple_stmt_simple_expr_data comp_simple_stmt_simple_expr_state

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of spec02.

Q.E.D.

C.42.9 Cpp_Examples.spec03

Terse proof for spec03.

spec03:

{1}	plain_memory?(pm) \supset Valid(PRECONDITION, e2s(assign(pm, dt(int))(id(n), literal(42))), POSTCONDITION)
-----	---

Installing automatic rewrites from: valid Valid PRECONDITION POSTCONDITION

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 Repeatedly Skolemizing and flattening,

Installing automatic rewrites from: literal id list_props.member postinc postdec not_exp preint predec
 IntExpressions.* IntExpressions.div IntExpressions.rem IntExpressions.- IntExpressions.■ IntExpres-
 sions.■ IntExpressions.< IntExpressions.> IntExpressions.<= IntExpressions.>= EqualityExpres-
 sions.== not_equal and_exp or_exp assign assign_times IntExpressions.+ assign_plus assign_minus
 assign_div assign_rem comma assign_bsr assign_bsl deref IntExpressions2.+ IntExpressions2.-
 Cpp_pointer.+ Cpp_pointer.- ptr_addr_ptr_of pointer_add_n_pointer_minus pointer_add_difference
 Aligned_Ptr

Installing automatic rewrites from: plain_memory_write_ok_single plain_memory_write_ok_q_expr
 plain_memory_write_ok_q_stmt in_blessed_memory_rw_ro pm_q_prop_ok pm_q_prop_single_read
 pm_q_prop_single_write pm_q_prop_read_write_single pm_q_prop_read_write_other_single pm_q_prop_read_read_sing
 plain_memory_read_write_other_single_data plain_memory_read_read plain_memory_read_write_res
 pm_q_prop_read_ok_expr pm_q_prop_read_ok_stmt pm_q_prop_read_pm_q_prop_expr pm_q_prop_read_pm_q_prop_st
 pm_q_prop_read_write_q_expr pm_q_prop_read_write_q_stmt plain_memory_read_write_q_data_expr
 plain_memory_read_write_q_data_stmt pm_q_prop_read_write_other_q_expr pm_q_prop_read_write_other_q_stmt
 pm_q_prop_read_read_q_expr pm_q_prop_read_read_q_stmt plain_memory_read_write_other_q_data_expr
 plain_memory_read_write_other_q_data_stmt plain_memory_read_read_q_data_expr plain_memory_read_read_q_data
 pm_q_prop_ok_result_single pm_q_prop_ok_result_q_expr pm_q_prop_ok_result_q_stmt pm_q_prop_read_ok_result_sir
 pm_q_prop_read_ok_result_q_expr pm_q_prop_read_ok_result_q_stmt pm_q_prop_e2s pm_q_prop_stmt_e2s
 pm_q_prop_read_e2s pm_q_prop_read_stmt_e2s

Installing automatic rewrites from: if_else switch member break_catch_break break_break_stmt
 break_break_catch_continue break_break_catch_default break_break_lift_stmt break_break_case
 break_break_default break_break stmt_break_break break_stmt_break break_stmt_catch_continue
 break_stmt_catch_default break_stmt_lift break_stmt_case break_stmt_default continue_catch_continue
 continue_break_stmt continue_break_catch_continue continue_break_catch_default continue_break_lift_stmt
 continue_break_case continue_break_default continue_continue stmt_continue_continue con-
 tinue_stmt_continue continue_stmt_catch_continue continue_stmt_catch_default continue_stmt_lift
 continue_stmt_case continue_stmt_default switch_stmt stmt_switch_stmt switch_case_taken
 stmt_switch_case_taken switch_case_not_taken stmt_switch_case_not_taken switch_default stmt_switch_default
 switch_catch_break stmt_switch_catch_break switch_catch_continue stmt_switch_catch_continue
 switch_catch_default stmt_switch_catch_default default_stmt stmt_default_stmt default_case
 stmt_default_case default_default stmt_default_default default_catch_break stmt_default_catch_break
 default_catch_continue stmt_default_catch_continue default_catch_default stmt_default_catch_default
 return_void_result return_ex_result stmt_remove_catch_default stmt_remove_case stmt_remove_default
 stmt_remove_catch_continue stmt_remove_catch_break break_break_stmt_expr break_break_catch_continue_expr
 break_break_catch_default_expr break_break_lift_stmt_expr break_break_case_expr break_break_default_expr
 continue_break_stmt_expr continue_break_catch_continue_expr continue_break_catch_default_expr con-
 tinue_break_lift_stmt_expr continue_break_case_expr continue_break_default_expr

Installing automatic rewrites from: skip_ok skip_state skip_elimination stmt_skip_elimination
 stmt_ok_lift stmt_ok_lift_expr e2s_ok stmt_e2s_ok e2s_state stmt_e2s_state ok_result_fexpr
 ok_result_fstmt e2s_merge stmt_e2s_merge e2s_expr stmt_e2s_expr

Installing automatic rewrites from: while_unroll stmt_while_unroll iterate_while_ok while_hang
 while_unroll_expr while_unroll_lexpr stmt_while_unroll_expr stmt_while_unroll_lexpr while_unroll_0
 do_as_while do_while_unroll for_as_while for_unroll

Installing automatic rewrites from: composition_assoc_rewrite_stmt_ok forward_s_c_s_of_s_expr
 comp_eval_if_ok_fstmt stmt_eval_if_ok_fstmt comp_eval_if_ok_fstmt_cstmt stmt_eval_if_ok_fstmt_cstmt
 composition_assoc_rewrite_1 composition_assoc_rewrite_2 composition_assoc_rewrite_3 compo-
 sition_assoc_rewrite_4 composition_assoc_rewrite_5 composition_assoc_rewrite_6 compo-
 sition_assoc_rewrite_7 composition_assoc_rewrite_8 composition_assoc_rewrite_expr_1 compo-
 sition_assoc_rewrite_expr_2 composition_assoc_rewrite_expr_3 composition_assoc_rewrite_expr_4 compo-
 sition_assoc_rewrite_expr_5 composition_assoc_rewrite_expr_6 composition_assoc_rewrite_stmt_ok compo-

sition_assoc_rewrite_stmt_ok_expr composition_assoc_rewrite_stmt_ok_expr_lexpr comp_eval_if_ok_fstmt
 comp_eval_if_ok_fstmt_expr comp_eval_if_ok_fstmt_cstmt_expr stmt_eval_if_ok_fstmt_expr stmt_eval_if_ok_fstmt_cstmt_expr
 comp_eval_if_ok_fexpr_expr stmt_eval_if_ok_fexpr_expr_eval_if_ok_fexpr composition_assoc_expression_rewrite
 composition_assoc_expression_rewrite_stmt stmt_eval_if_ok_fexpr_expr expr_eval_if_ok_fexpr_expr
 ok_result_fexpr ok_result_fstmt composition_assoc_rewrite_stmt_ok_cstmt

Installing automatic rewrites from: allocate_stack deallocate_stack with_new_stackvar

Installing automatic rewrites from: ok_result_ok ok_result_data ok_result_q_ok ok_result_q_state
 ok_result_q_data ok_result_elimination_1 ok_result_elimination_2 ok_result_state

Installing automatic rewrites from: l2r n2b a2b

Installing automatic rewrites from: comp_simple_expr_simple_expr_ok comp_simple_expr_simple_expr_data
 comp_simple_expr_simple_expr_state comp_simple_stmt_simple_expr_ok comp_simple_stmt_simple_expr_data
 comp_simple_stmt_simple_expr_state

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of spec03.

Q.E.D.

C.42.10 Cpp_Examples.spec04_TCC1

Terse proof for spec04_TCC1.

spec04_TCC1:

{1} plain_memory?(pm) \supset range(char)(42)

Applying disjunctive simplification to flatten sequent,

Expanding the definition of range,

Expanding the definition of range_integral,

Rewriting using char_binary, matching in *,

Using lemma min_char,

Using lemma max_char,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of spec04_TCC1.

Q.E.D.

C.42.11 Cpp_Examples.spec04_TCC2

Terse proof for spec04_TCC2.

spec04_TCC2:

{1} plain_memory?(pm) \supset Cpp_Type?(char) \wedge cv(non_bool_integral_enum?)(char)

Repeatedly Skolemizing and flattening,

Hiding formulas: -1,

Applying propositional simplification,

we get 2 subgoals:

spec04_TCC2.1:

{1} Cpp_Type?(char)

Expanding the definition of Cpp_Type?,

Expanding the definition of subterm,

Repeatedly Skolemizing and flattening,

Replacing using formula -1,

Installing automatic rewrites from: no_pointers_to_bitfield? no_pointers_to_references? no_cv_references? no_reference_to_reference? no_reference_to_bitfields? no_pointer_to_member_to_reference? no_pointer_to_member_to_cv_void? no_cv_void_parameter? no_array_of_references? no_array_of_bitfields? no_array_of_cv_void? no_array_of_function? no_array_of_abstract_class? no_pointer_or_ref_to_incomplete_array_parameter? no_array_return_type? no_function_return_type? bitfield_underlying_integral_or_enum_type? cv_array? enum_underlying_integral? enum_constants? const_volatile? const_stutter? volatile_stutter? no_cv_class? no_cv_union? no_cv_function? no_reference_to_void? no_cv_void?

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of spec04_TCC2.1.

spec04_TCC2.2:

```
{1} cv(non_bool_integral_enum?)(char)
```

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of spec04_TCC2.2.

Q.E.D.

C.42.12 Cpp_Examples.spec04

Terse proof for spec04.

spec04:

```
{1} plain_memory?(pm)  $\supset$ 
    valid(PRECONDITION,
          e2s(assign(pm, dt(char))(id(a), literal(42))) ##
            e2s(assign(pm, dt(char))(id(b), literal(42))),
           $\lambda$  (s: State):
            OK?(read_data(pm, dt(char))(a)(s))  $\wedge$ 
            data(read_data(pm, dt(char))(a)(s)) = 42)
```

Installing automatic rewrites from: valid Valid PRECONDITION POSTCONDITION

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Repeatedly Skolemizing and flattening,

Installing automatic rewrites from: address_of_spec address_of_spec2 add_simplification add_spec add_spec2 same_array_spec same_array_spec2 mod_range size_of uidt check_bounds_spec er_plus_ax er_minus_ax er_div_ax er_times_ax er_neg_ax extended_real_superset_of_number_fields

Installing automatic rewrites from: literal id member postinc postdec deref arrow unary_minus unary_minus_unsigned unary_not unary_bitnot preinc predec UnaryExpressions.sizeof ptm_member ptm_arrow arrow times div div_float mod plus plus_ptr postinc_ptr preinc_ptr subscript minus minus_ptr postdec_ptr predec_ptr minus_ptr_ptr cmp lt gt le ge lt_ptr gt_ptr le_ptr ge_ptr cmp_pointer eq_bool eq not_equal not_equal_bool eq_ptr not_equal_ptr bitand bitxor bitor and_exp or_exp conditional_expr assign assign_times assign_div assign_mod assign_div_float assign_plus assign_plus_ptr assign_minus assign_minus_ptr assign_bitand assign_bitxor assign_bitor comma

Installing automatic rewrites from: plain_memory_write_ok_single plain_memory_read_data_ok plain_memory_write_ok_q_expr plain_memory_write_ok_q_stmt in_blessed_memory_rw_ro pm_q_prop_ok pm_q_prop_single_read pm_q_prop_single_write pm_q_prop_read_write_single pm_q_prop_read_write_other_single pm_q_prop_read_read_single plain_memory_read_write_other_single_data plain_memory_read_read plain_memory_read_write_res pm_q_prop_read_ok_expr pm_q_prop_read_ok_stmt pm_q_prop_read_pm_q_prop_expr pm_q_prop_read_pm_q_prop_stmt pm_q_prop_read_write_q_expr pm_q_prop_read_write_q_stmt plain_memory_read_write_q_data_expr plain_memory_read_write_q_data_stmt pm_q_prop_read_write_other_q_expr pm_q_prop_read_write_other_q_stmt pm_q_prop_read_read_q_expr pm_q_prop_read_read_q_stmt plain_memory_read_write_other_q_data_expr plain_memory_read_write_other_q_data_stmt plain_memory_read_read

plain_memory_read_read_q_data_stmt pm_q_prop_ok_result_single pm_q_prop_ok_result_q_expr
 pm_q_prop_ok_result_q_stmt pm_q_prop_read_ok_result_single pm_q_prop_read_ok_result_q_expr
 pm_q_prop_read_ok_result_q_stmt pm_q_prop_e2s pm_q_prop_stmt_e2s pm_q_prop_read_e2s
 pm_q_prop_read_stmt_e2s pm_q_prop_read_ok_pm_q_prop_read_expr pm_q_prop_read_ok_pm_q_prop_read_stmt

Installing automatic rewrites from: if_else switch member break_catch_break break_break_stmt
 break_break_catch_continue break_break_catch_default break_break_lift_stmt break_break_case
 break_break_default break_break stmt_break_break break_stmt_break break_stmt_catch_continue
 break_stmt_catch_default break_stmt_lift break_stmt_case break_stmt_default continue_catch_continue
 continue_break_stmt continue_break_catch_continue continue_break_catch_default continue_break_lift_stmt
 continue_break_case continue_break_default continue_continue stmt_continue_continue con-
 tinue_stmt_continue continue_stmt_catch_continue continue_stmt_catch_default continue_stmt_lift
 continue_stmt_case continue_stmt_default switch_stmt stmt_switch_stmt switch_case_taken
 stmt_switch_case_taken switch_case_not_taken stmt_switch_case_not_taken switch_default stmt_switch_default
 switch_catch_break stmt_switch_catch_break switch_catch_continue stmt_switch_catch_continue
 switch_catch_default stmt_switch_catch_default default_stmt stmt_default_stmt default_case
 stmt_default_case default_default stmt_default_default default_catch_break stmt_default_catch_break
 default_catch_continue stmt_default_catch_continue default_catch_default stmt_default_catch_default
 return_void_result return_ex_result stmt_remove_catch_default stmt_remove_case stmt_remove_default
 stmt_remove_catch_continue stmt_remove_catch_break break_break_stmt_expr break_break_catch_continue_expr
 break_break_catch_default_expr break_break_lift_stmt_expr break_break_case_expr break_break_default_expr
 continue_break_stmt_expr continue_break_catch_continue_expr continue_break_catch_default_expr con-
 tinue_break_lift_stmt_expr continue_break_case_expr continue_break_default_expr

Installing automatic rewrites from: skip_ok skip_state skip_elimination stmt_skip_elimination
 stmt_ok_lift stmt_ok_lift_expr e2s_ok stmt_e2s_ok e2s_state stmt_e2s_state ok_result_fexpr
 ok_result_fstmt e2s_merge stmt_e2s_merge e2s_expr stmt_e2s_expr

Installing automatic rewrites from: while_unroll stmt_while_unroll iterate_while_ok while_hang
 while_unroll_expr while_unroll_lexpr stmt_while_unroll_expr stmt_while_unroll_lexpr while_unroll_0
 do_while_unroll do_while_inv_unroll for_unroll for_inv_unroll while_to_while_no_cb while_invariant?!
 while_variant?! while_inv_rewrite_termination_result! while_inv_rewrite_data_ok! while_inv_rewrite_data_break!
 while_inv_rewrite_data_return! stmt_while_inv_rewrite_termination_result! stmt_while_inv_rewrite_data_ok!
 stmt_while_inv_rewrite_data_break! stmt_while_inv_rewrite_data_return! pm_q_prop_while!
 pm_q_prop_stmt_while!

Installing automatic rewrites from: composition_assoc_rewrite_stmt_ok forward_s_c_s_of_s_expr
 comp_eval_if_ok_fstmt stmt_eval_if_ok_fstmt comp_eval_if_ok_fstmt_cstmt stmt_eval_if_ok_fstmt_cstmt
 composition_assoc_rewrite_1 composition_assoc_rewrite_2 composition_assoc_rewrite_3 com-
 position_assoc_rewrite_4 composition_assoc_rewrite_5 composition_assoc_rewrite_6 composi-
 tion_assoc_rewrite_7 composition_assoc_rewrite_8 composition_assoc_rewrite_expr_1 composi-
 tion_assoc_rewrite_expr_2 composition_assoc_rewrite_expr_3 composition_assoc_rewrite_expr_4 composi-
 tion_assoc_rewrite_expr_5 composition_assoc_rewrite_expr_6 composition_assoc_rewrite_stmt_ok composi-
 tion_assoc_rewrite_stmt_ok_expr composition_assoc_rewrite_stmt_ok_expr_lexpr comp_eval_if_ok_fstmt
 comp_eval_if_ok_fstmt_expr comp_eval_if_ok_fstmt_cstmt_expr stmt_eval_if_ok_fstmt_expr stmt_eval_if_ok_fstmt_cstmt_expr
 comp_eval_if_ok_fexpr_expr comp_eval_if_ok_fexpr stmt_eval_if_ok_fexpr expr_eval_if_ok_fexpr composi-
 tion_assoc_expression_rewrite composition_assoc_expression_rewrite_stmt stmt_eval_if_ok_fexpr_expr
 expr_eval_if_ok_fexpr_expr ok_result_fexpr ok_result_fstmt composition_assoc_rewrite_stmt_ok_cstmt

Installing automatic rewrites from: allocate_stack deallocate_stack with_new_stackvar

Installing automatic rewrites from: ok_result_ok ok_result_data ok_result_ok ok_result_q_state
 ok_result_q_data ok_result_elimination_1 ok_result_elimination_2 ok_result_state

Installing automatic rewrites from: l2r integral_to_bool pointer_to_bool bool2int

Installing automatic rewrites from: comp_simple_expr_simple_expr_ok comp_simple_expr_simple_expr_data
 comp_simple_expr_simple_expr_state comp_simple_stmt_simple_expr_ok comp_simple_stmt_simple_expr_data
 comp_simple_stmt_simple_expr_state

Installing automatic rewrites from: floating_point?

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 This completes the proof of `spec04`.
 Q.E.D.

C.42.13 Cpp_Examples.spec05

Terse proof for `spec05`.

`spec05`:

```

{1} plain_memory?(pm) ⊃
    valid(PRECONDITION,
        e2s(assign(pm, dt_bool)(id(c), literal(TRUE))) ##
        if_else(l2r(pm, dt_bool)(id(c)),
            e2s(assign(pm, dt(int))(id(n), literal(42))), skip),
        POSTCONDITION)
    
```

Installing automatic rewrites from: `valid Valid PRECONDITION POSTCONDITION`

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Repeatedly Skolemizing and flattening,

Installing automatic rewrites from: `address_of_spec address_of_spec2 add_simplification add_spec add_spec2 same_array_spec same_array_spec2 mod_range size_of uidt check_bounds_spec er_plus_ax er_minus_ax er_div_ax er_times_ax er_neg_ax extended_real_superset_of_number_fields`

Installing automatic rewrites from: `literal id member postinc postdec deref arrow unary_minus unary_minus_unsigned unary_not unary_bitnot preinc predec UnaryExpressions.sizeof ptm_member ptm_arrow arrow times div div_float mod plus plus_ptr postinc_ptr preinc_ptr subscript minus minus_ptr postdec_ptr predec_ptr minus_ptr_ptr cmp lt gt le ge lt_ptr gt_ptr le_ptr ge_ptr cmp_pointer eq_bool eq not_equal not_equal_bool eq_ptr not_equal_ptr bitand bitxor bitor and_exp or_exp conditional_exp assign assign_times assign_div assign_mod assign_div_float assign_plus assign_plus_ptr assign_minus assign_minus_ptr assign_bitand assign_bitxor assign_bitor comma`

Installing automatic rewrites from: `plain_memory_write_ok_single plain_memory_read_data_ok plain_memory_write_ok_q_expr plain_memory_write_ok_q_stmt in_blessed_memory_rw_ro pm_q_prop_ok pm_q_prop_single_read pm_q_prop_single_write pm_q_prop_read_write_single pm_q_prop_read_write_other_single pm_q_prop_read_read_single plain_memory_read_write_other_single_data plain_memory_read_read plain_memory_read_write_res pm_q_prop_read_ok_expr pm_q_prop_read_ok_stmt pm_q_prop_read_pm_q_prop_expr pm_q_prop_read_pm_q_prop_stmt pm_q_prop_read_write_q_expr pm_q_prop_read_write_q_stmt plain_memory_read_write_q_data_expr plain_memory_read_write_q_data_stmt pm_q_prop_read_write_other_q_expr pm_q_prop_read_write_other_q_stmt pm_q_prop_read_read_q_expr pm_q_prop_read_read_q_stmt plain_memory_read_write_other_q_data_expr plain_memory_read_write_other_q_data_stmt plain_memory_read_read plain_memory_read_read_q_data_stmt pm_q_prop_ok_result_single pm_q_prop_ok_result_q_expr pm_q_prop_ok_result_q_stmt pm_q_prop_read_ok_result_single pm_q_prop_read_ok_result_q_expr pm_q_prop_read_ok_result_q_stmt pm_q_prop_e2s pm_q_prop_stmt_e2s pm_q_prop_read_e2s pm_q_prop_read_stmt_e2s pm_q_prop_read_ok_pm_q_prop_read_expr pm_q_prop_read_ok_pm_q_prop_read_stmt`

Installing automatic rewrites from: `if_else switch member break_catch_break break_break_stmt break_break_catch_continue break_break_catch_default break_break_lift_stmt break_break_case break_break_default break_break_stmt_break_break break_stmt_break break_stmt_catch_continue break_stmt_catch_default break_stmt_lift break_stmt_case break_stmt_default continue_catch_continue continue_break_stmt continue_break_catch_continue continue_break_catch_default continue_break_lift_stmt continue_break_case continue_break_default continue_continue stmt_continue_continue continue_stmt_continue continue_stmt_catch_continue continue_stmt_catch_default continue_stmt_lift continue_stmt_case continue_stmt_default switch_stmt stmt_switch_stmt switch_case_taken stmt_switch_case_taken switch_case_not_taken stmt_switch_case_not_taken switch_default stmt_switch_default switch_catch_break stmt_switch_catch_break switch_catch_continue stmt_switch_catch_continue`

switch_catch_default stmt_switch_catch_default default_stmt stmt_default_stmt default_case
 stmt_default_case default_default stmt_default_default default_catch_break stmt_default_catch_break
 default_catch_continue stmt_default_catch_continue default_catch_default stmt_default_catch_default
 return_void_result return_ex_result stmt_remove_catch_default stmt_remove_case stmt_remove_default
 stmt_remove_catch_continue stmt_remove_catch_break break_break_stmt_expr break_break_catch_continue_expr
 break_break_catch_default_expr break_break_lift_stmt_expr break_break_case_expr break_break_default_expr
 continue_break_stmt_expr continue_break_catch_continue_expr continue_break_catch_default_expr con-
 tinue_break_lift_stmt_expr continue_break_case_expr continue_break_default_expr

Installing automatic rewrites from: skip_ok skip_state skip_elimination stmt_skip_elimination
 stmt_ok_lift stmt_ok_lift_expr e2s_ok stmt_e2s_ok e2s_state stmt_e2s_state ok_result_fexpr
 ok_result_fstmt e2s_merge stmt_e2s_merge e2s_expr stmt_e2s_expr

Installing automatic rewrites from: while_unroll stmt_while_unroll iterate_while_ok while_hang
 while_unroll_expr while_unroll_lexpr stmt_while_unroll_expr stmt_while_unroll_lexpr while_unroll_0
 do_while_unroll do_while_inv_unroll for_unroll for_inv_unroll while_to_while_no_cb while_invariant?!
 while_variant?! while_inv_rewrite_termination_result! while_inv_rewrite_data_ok! while_inv_rewrite_data_break!
 while_inv_rewrite_data_return! stmt_while_inv_rewrite_termination_result! stmt_while_inv_rewrite_data_ok!
 stmt_while_inv_rewrite_data_break! stmt_while_inv_rewrite_data_return! pm_q_prop_while!
 pm_q_prop_stmt_while!

Installing automatic rewrites from: composition_assoc_rewrite_stmt_ok forward_s_c_s_of_s_expr
 comp_eval_if_ok_fstmt stmt_eval_if_ok_fstmt comp_eval_if_ok_fstmt_cstmt stmt_eval_if_ok_fstmt_cstmt
 composition_assoc_rewrite_1 composition_assoc_rewrite_2 composition_assoc_rewrite_3 com-
 position_assoc_rewrite_4 composition_assoc_rewrite_5 composition_assoc_rewrite_6 composi-
 tion_assoc_rewrite_7 composition_assoc_rewrite_8 composition_assoc_rewrite_expr_1 composi-
 tion_assoc_rewrite_expr_2 composition_assoc_rewrite_expr_3 composition_assoc_rewrite_expr_4 composi-
 tion_assoc_rewrite_expr_5 composition_assoc_rewrite_expr_6 composition_assoc_rewrite_stmt_ok compo-
 sition_assoc_rewrite_stmt_ok_expr composition_assoc_rewrite_stmt_ok_expr_lexpr comp_eval_if_ok_fstmt
 comp_eval_if_ok_fstmt_expr comp_eval_if_ok_fstmt_cstmt_expr stmt_eval_if_ok_fstmt_expr stmt_eval_if_ok_fstmt_cstmt_expr
 comp_eval_if_ok_fexpr_expr comp_eval_if_ok_fexpr stmt_eval_if_ok_fexpr expr_eval_if_ok_fexpr compo-
 sition_assoc_expression_rewrite composition_assoc_expression_rewrite_stmt stmt_eval_if_ok_fexpr_expr
 expr_eval_if_ok_fexpr_expr ok_result_fexpr ok_result_fstmt composition_assoc_rewrite_stmt_ok_cstmt

Installing automatic rewrites from: allocate_stack deallocate_stack with_new_stackvar

Installing automatic rewrites from: ok_result_ok ok_result_data ok_result_q_ok ok_result_q_state
 ok_result_q_data ok_result_elimination_1 ok_result_elimination_2 ok_result_state

Installing automatic rewrites from: l2r integral_to_bool pointer_to_bool bool2int

Installing automatic rewrites from: comp_simple_expr_simple_expr_ok comp_simple_expr_simple_expr_data
 comp_simple_expr_simple_expr_state comp_simple_stmt_simple_expr_ok comp_simple_stmt_simple_expr_data
 comp_simple_stmt_simple_expr_state

Installing automatic rewrites from: floating_point?

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of spec05.

Q.E.D.

C.42.14 Cpp_Examples.spec06_TCC1

Terse proof for spec06_TCC1.

spec06_TCC1:

{1} plain_memory?(pm) \supset range(char)(0)

Applying disjunctive simplification to flatten sequent,

Expanding the definition of range,

Expanding the definition of range_integral,

Rewriting using `char_binary`, matching in `*`,
Using lemma `min_char`,
Using lemma `max_char`,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `spec06_TCC1`.
Q.E.D.

C.42.15 Cpp_Examples.spec06_TCC2

Terse proof for `spec06_TCC2`.

`spec06_TCC2`:

$$\{1\} \text{ plain_memory?}(pm) \supset \text{range}(\text{char})(1)$$

Applying disjunctive simplification to flatten sequent,
Expanding the definition of `range`,
Expanding the definition of `range_integral`,
Rewriting using `char_binary`, matching in `*`,
Using lemma `min_char`,
Using lemma `max_char`,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `spec06_TCC2`.
Q.E.D.

C.42.16 Cpp_Examples.spec06_TCC3

Terse proof for `spec06_TCC3`.

`spec06_TCC3`:

$$\{1\} \text{ plain_memory?}(pm) \supset \text{every}[\text{real}](\lambda (x: \text{real}): \text{rational_pred}(x) \wedge \text{integer_pred}(x))((:1:))$$

Repeatedly Skolemizing and flattening,
Using lemma `spec04_TCC2`,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `spec06_TCC3`.
Q.E.D.

C.42.17 Cpp_Examples.spec06_TCC4

Terse proof for `spec06_TCC4`.

`spec06_TCC4`:

$$\{1\} \text{ plain_memory?}(pm) \supset \text{range}(\text{int})(0)$$

Applying disjunctive simplification to flatten sequent,
Expanding the definition of `range`,
Expanding the definition of `range_integral`,
Rewriting using `int_binary`, matching in `*`,
Using lemma `min_int`,

Using lemma `max_int`,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 This completes the proof of `spec06_TCC4`.
 Q.E.D.

C.42.18 Cpp_Examples.spec06

Terse proof for `spec06`.

`spec06`:

```
{1} plain_memory?(pm)  $\supset$ 
  valid(PRECONDITION,
    e2s(assign(pm, dt(char))(id(a), literal(0))) ##
    e2s(assign(pm, dt(char))(id(b), literal(1)))
    ##
    switch(l2r(pm, dt(char))(id(a)), (:0, 1:),
      case(0) ## skip ## case(1) ##
      e2s(assign(pm, dt(int))(id(n), literal(42)))
      ## break
      ## default
      ## e2s(assign(pm, dt(int))(id(n), literal(0))),
    POSTCONDITION)
```

Installing automatic rewrites from: `valid Valid PRECONDITION POSTCONDITION`

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Repeatedly Skolemizing and flattening,

Installing automatic rewrites from: `address_of_spec address_of_spec2 add_simplification add_spec add_spec2 same_array_spec same_array_spec2 mod_range size_of uidt check_bounds_spec er_plus_ax er_minus_ax er_div_ax er_times_ax er_neg_ax extended_real_superset_of_number_fields`

Installing automatic rewrites from: `literal id member postinc postdec deref arrow unary_minus unary_minus_unsigned unary_not unary_bitnot preinc predec UnaryExpressions.sizeof ptm_member ptm_arrow arrow times div div_float mod plus plus_ptr postinc_ptr preinc_ptr subscript minus minus_ptr postdec_ptr predec_ptr minus_ptr_ptr cmp lt gt le ge lt_ptr gt_ptr le_ptr ge_ptr cmp_pointer eq_bool eq not_equal not_equal_bool eq_ptr not_equal_ptr bitand bitxor bitor and_exp or_exp conditional_exp assign assign_times assign_div assign_mod assign_div_float assign_plus assign_plus_ptr assign_minus assign_minus_ptr assign_bitand assign_bitxor assign_bitor comma`

Installing automatic rewrites from: `plain_memory_write_ok_single plain_memory_read_data_ok plain_memory_write_ok_q_expr plain_memory_write_ok_q_stmt in_blessed_memory_rw_ro pm_q_prop_ok pm_q_prop_single_read pm_q_prop_single_write pm_q_prop_read_write_single pm_q_prop_read_write_other_single pm_q_prop_read_read_single plain_memory_read_write_other_single_data plain_memory_read_read plain_memory_read_write_res pm_q_prop_read_ok_expr pm_q_prop_read_ok_stmt pm_q_prop_read_pm_q_prop_expr pm_q_prop_read_pm_q_prop_stmt pm_q_prop_read_write_q_expr pm_q_prop_read_write_q_stmt plain_memory_read_write_q_data_expr plain_memory_read_write_q_data_stmt pm_q_prop_read_write_other_q_expr pm_q_prop_read_write_other_q_stmt pm_q_prop_read_read_q_expr pm_q_prop_read_read_q_stmt plain_memory_read_write_other_q_data_expr plain_memory_read_write_other_q_data_stmt plain_memory_read_read_q_data_expr plain_memory_read_read_q_data_stmt pm_q_prop_ok_result_single pm_q_prop_ok_result_q_expr pm_q_prop_ok_result_q_stmt pm_q_prop_read_ok_result_single pm_q_prop_read_ok_result_q_expr pm_q_prop_read_ok_result_q_stmt pm_q_prop_e2s pm_q_prop_stmt_e2s pm_q_prop_read_e2s pm_q_prop_read_stmt_e2s pm_q_prop_read_ok_pm_q_prop_read_expr pm_q_prop_read_ok_pm_q_prop_read_stmt`

Installing automatic rewrites from: `if_else switch member break_catch_break break_break_stmt break_break_catch_continue break_break_catch_default break_break_lift_stmt break_break_case break_break_default break_break stmt_break_break break_stmt_break break_stmt_catch_continue`

break_stmt_catch_default break_stmt_lift break_stmt_case break_stmt_default continue_catch_continue
 continue_break_stmt continue_break_catch_continue continue_break_catch_default continue_break_lift_stmt
 continue_break_case continue_break_default continue_continue stmt_continue_continue con-
 tinue_stmt_continue continue_stmt_catch_continue continue_stmt_catch_default continue_stmt_lift
 continue_stmt_case continue_stmt_default switch_stmt stmt_switch_stmt switch_case_taken
 stmt_switch_case_taken switch_case_not_taken stmt_switch_case_not_taken switch_default stmt_switch_default
 switch_catch_break stmt_switch_catch_break switch_catch_continue stmt_switch_catch_continue
 switch_catch_default stmt_switch_catch_default default_stmt stmt_default_stmt default_case
 stmt_default_case default_default stmt_default_default default_catch_break stmt_default_catch_break
 default_catch_continue stmt_default_catch_continue default_catch_default stmt_default_catch_default
 return_void_result return_ex_result stmt_remove_catch_default stmt_remove_case stmt_remove_default
 stmt_remove_catch_continue stmt_remove_catch_break break_break_stmt_expr break_break_catch_continue_expr
 break_break_catch_default_expr break_break_lift_stmt_expr break_break_case_expr break_break_default_expr
 continue_break_stmt_expr continue_break_catch_continue_expr continue_break_catch_default_expr con-
 tinue_break_lift_stmt_expr continue_break_case_expr continue_break_default_expr

Installing automatic rewrites from: skip_ok skip_state skip_elimination stmt_skip_elimination
 stmt_ok_lift stmt_ok_lift_expr e2s_ok stmt_e2s_ok e2s_state stmt_e2s_state ok_result_fexpr
 ok_result_fstmt e2s_merge stmt_e2s_merge e2s_expr stmt_e2s_expr

Installing automatic rewrites from: while_unroll stmt_while_unroll iterate_while_ok while_hang
 while_unroll_expr while_unroll_lexpr stmt_while_unroll_expr stmt_while_unroll_lexpr while_unroll_0
 do_while_unroll do_while_inv_unroll for_unroll for_inv_unroll while_to_while_no_cb while_invariant?!
 while_variant?! while_inv_rewrite_termination_result! while_inv_rewrite_data_ok! while_inv_rewrite_data_break!
 while_inv_rewrite_data_return! stmt_while_inv_rewrite_termination_result! stmt_while_inv_rewrite_data_ok!
 stmt_while_inv_rewrite_data_break! stmt_while_inv_rewrite_data_return! pm_q_prop_while!
 pm_q_prop_stmt_while!

Installing automatic rewrites from: composition_assoc_rewrite_stmt_ok forward_s_c_s_of_s_expr
 comp_eval_if_ok_fstmt stmt_eval_if_ok_fstmt comp_eval_if_ok_fstmt_cstmt stmt_eval_if_ok_fstmt_cstmt
 composition_assoc_rewrite_1 composition_assoc_rewrite_2 composition_assoc_rewrite_3 com-
 position_assoc_rewrite_4 composition_assoc_rewrite_5 composition_assoc_rewrite_6 composi-
 tion_assoc_rewrite_7 composition_assoc_rewrite_8 composition_assoc_rewrite_expr_1 composi-
 tion_assoc_rewrite_expr_2 composition_assoc_rewrite_expr_3 composition_assoc_rewrite_expr_4 composi-
 tion_assoc_rewrite_expr_5 composition_assoc_rewrite_expr_6 composition_assoc_rewrite_stmt_ok compo-
 sition_assoc_rewrite_stmt_ok_expr composition_assoc_rewrite_stmt_ok_expr_lexpr comp_eval_if_ok_fstmt
 comp_eval_if_ok_fstmt_expr comp_eval_if_ok_fstmt_cstmt_expr stmt_eval_if_ok_fstmt_expr stmt_eval_if_ok_fstmt_cst-
 comp_eval_if_ok_fexpr_expr comp_eval_if_ok_fexpr stmt_eval_if_ok_fexpr expr_eval_if_ok_fexpr compo-
 sition_assoc_expression_rewrite composition_assoc_expression_rewrite_stmt stmt_eval_if_ok_fexpr_expr
 expr_eval_if_ok_fexpr_expr ok_result_fexpr ok_result_fstmt composition_assoc_rewrite_stmt_ok_cstmt

Installing automatic rewrites from: allocate_stack deallocate_stack with_new_stackvar

Installing automatic rewrites from: ok_result_ok ok_result_data ok_result_q_ok ok_result_q_state
 ok_result_q_data ok_result_elimination_1 ok_result_elimination_2 ok_result_state

Installing automatic rewrites from: l2r integral_to_bool pointer_to_bool bool2int

Installing automatic rewrites from: comp_simple_expr_simple_expr_ok comp_simple_expr_simple_expr_data
 comp_simple_expr_simple_expr_state comp_simple_stmt_simple_expr_ok comp_simple_stmt_simple_expr_data
 comp_simple_stmt_simple_expr_state

Installing automatic rewrites from: floating_point?

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `spec06`.

Q.E.D.

C.42.19 Cpp_Examples.spec07

Terse proof for `spec07`.

spec07:

```
{1} valid(PRECONDITION, while(literal(TRUE), skip), λ (s: State): FALSE)
```

Installing automatic rewrites from: valid Valid PRECONDITION POSTCONDITION

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Repeatedly Skolemizing and flattening,

Installing automatic rewrites from: literal id list_props.member postinc postdec not_exp preint predec IntExpressions.* IntExpressions.div IntExpressions.rem IntExpressions.- IntExpressions.■ IntExpressions.■ IntExpressions.< IntExpressions.> IntExpressions.<= IntExpressions.>= EqualityExpressions.== not_equal and_exp or_exp assign assign_times IntExpressions.+ assign_plus assign_minus assign_div assign_rem comma assign_bsr assign_bsl deref IntExpressions2.+ IntExpressions2.- Cpp_pointer.+ Cpp_pointer.- ptr_addr_ptr_of pointer_add_n_pointer_minus pointer_add_difference Aligned_Ptr

Installing automatic rewrites from: plain_memory_write_ok_single plain_memory_write_ok_q_expr plain_memory_write_ok_q_stmt in_blessed_memory_rw_ro pm_q_prop_ok pm_q_prop_single_read pm_q_prop_single_write pm_q_prop_read_write_single pm_q_prop_read_write_other_single pm_q_prop_read_read_single plain_memory_read_write_other_single_data plain_memory_read_read plain_memory_read_write_res pm_q_prop_read_ok_expr pm_q_prop_read_ok_stmt pm_q_prop_read_pm_q_prop_expr pm_q_prop_read_pm_q_prop_stmt pm_q_prop_read_write_q_expr pm_q_prop_read_write_q_stmt plain_memory_read_write_q_data_expr plain_memory_read_write_q_data_stmt pm_q_prop_read_write_other_q_expr pm_q_prop_read_write_other_q_stmt pm_q_prop_read_read_q_expr pm_q_prop_read_read_q_stmt plain_memory_read_write_other_q_data_expr plain_memory_read_write_other_q_data_stmt plain_memory_read_read_q_data_expr plain_memory_read_read_q_data_stmt pm_q_prop_ok_result_single pm_q_prop_ok_result_q_expr pm_q_prop_ok_result_q_stmt pm_q_prop_read_ok_result_single pm_q_prop_read_ok_result_q_expr pm_q_prop_read_ok_result_q_stmt pm_q_prop_e2s pm_q_prop_stmt_e2s pm_q_prop_read_e2s pm_q_prop_read_stmt_e2s

Installing automatic rewrites from: if_else switch member break_catch_break break_break_stmt break_break_catch_continue break_break_catch_default break_break_lift_stmt break_break_case break_break_default break_break_stmt_break_break break_stmt_break break_stmt_catch_continue break_stmt_catch_default break_stmt_lift break_stmt_case break_stmt_default continue_catch_continue continue_break_stmt continue_break_catch_continue continue_break_catch_default continue_break_lift_stmt continue_break_case continue_break_default continue_continue stmt_continue_continue continue_stmt_continue continue_stmt_catch_continue continue_stmt_catch_default continue_stmt_lift continue_stmt_case continue_stmt_default switch_stmt stmt_switch_stmt switch_case_taken stmt_switch_case_taken switch_case_not_taken stmt_switch_case_not_taken switch_default stmt_switch_default switch_catch_break stmt_switch_catch_break switch_catch_continue stmt_switch_catch_continue switch_catch_default stmt_switch_catch_default default_stmt stmt_default_stmt default_case stmt_default_case default_default stmt_default_default default_catch_break stmt_default_catch_break default_catch_continue stmt_default_catch_continue default_catch_default stmt_default_catch_default return_void_result return_ex_result stmt_remove_catch_default stmt_remove_case stmt_remove_default stmt_remove_catch_continue stmt_remove_catch_break break_break_stmt_expr break_break_catch_continue_expr break_break_catch_default_expr break_break_lift_stmt_expr break_break_case_expr break_break_default_expr continue_break_stmt_expr continue_break_catch_continue_expr continue_break_catch_default_expr continue_break_lift_stmt_expr continue_break_case_expr continue_break_default_expr

Installing automatic rewrites from: skip_ok skip_state skip_elimination stmt_skip_elimination stmt_ok_lift stmt_ok_lift_expr e2s_ok stmt_e2s_ok e2s_state stmt_e2s_state ok_result_fexpr ok_result_fstmt e2s_merge stmt_e2s_merge e2s_expr stmt_e2s_expr

Installing automatic rewrites from: while_unroll stmt_while_unroll iterate_while_ok while_hang while_unroll_expr while_unroll_lexpr stmt_while_unroll_expr stmt_while_unroll_lexpr while_unroll_0 do_as_while do_while_unroll for_as_while for_unroll

Installing automatic rewrites from: composition_assoc_rewrite_stmt_ok forward_s_c_s_of_s_expr comp_eval_if_ok_fstmt stmt_eval_if_ok_fstmt comp_eval_if_ok_fstmt_cstmt stmt_eval_if_ok_fstmt_cstmt

composition_assoc_rewrite_1 composition_assoc_rewrite_2 composition_assoc_rewrite_3 composition_assoc_rewrite_4 composition_assoc_rewrite_5 composition_assoc_rewrite_6 composition_assoc_rewrite_7 composition_assoc_rewrite_8 composition_assoc_rewrite_expr_1 composition_assoc_rewrite_expr_2 composition_assoc_rewrite_expr_3 composition_assoc_rewrite_expr_4 composition_assoc_rewrite_expr_5 composition_assoc_rewrite_expr_6 composition_assoc_rewrite_stmt_ok composition_assoc_rewrite_stmt_ok_expr composition_assoc_rewrite_stmt_ok_expr_lexpr comp_eval_if_ok_fstmt comp_eval_if_ok_fstmt_expr comp_eval_if_ok_fstmt_cstmt_expr stmt_eval_if_ok_fstmt_expr stmt_eval_if_ok_fstmt_cstmt_expr comp_eval_if_ok_fexpr_expr stmt_eval_if_ok_fexpr expr_eval_if_ok_fexpr composition_assoc_expression_rewrite composition_assoc_expression_rewrite_stmt stmt_eval_if_ok_fexpr_expr expr_eval_if_ok_fexpr_expr ok_result_fexpr ok_result_fstmt composition_assoc_rewrite_stmt_ok_cstmt

Installing automatic rewrites from: allocate_stack deallocate_stack with_new_stackvar

Installing automatic rewrites from: ok_result_ok ok_result_data ok_result_q_ok ok_result_q_state ok_result_q_data ok_result_elimination_1 ok_result_elimination_2 ok_result_state

Installing automatic rewrites from: l2r n2b a2b

Installing automatic rewrites from: comp_simple_expr_simple_expr_ok comp_simple_expr_simple_expr_data comp_simple_expr_simple_expr_state comp_simple_stmt_simple_expr_ok comp_simple_stmt_simple_expr_data comp_simple_stmt_simple_expr_state

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Expanding the definition of hang_result,

which is trivially true.

This completes the proof of spec07.

Q.E.D.

C.42.20 Cpp_Examples.N_TCC1

Terse proof for N_TCC1.

N_TCC1:

{1} $2 \geq 0 \wedge \text{range}(\text{int})(2)$

Installing automatic rewrites from: int_binary

Trying repeated skolemization, instantiation, and if-lifting,

we get 2 subgoals:

N_TCC1.1:

{1} $2 \leq \text{max}(\text{range_int})$

Using lemma max_int,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of N_TCC1.1.

N_TCC1.2:

{1} $\text{min}(\text{range_int}) \leq 2$

Using lemma min_int,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of N_TCC1.2.

Q.E.D.

C.42.21 Cpp_Examples.range_int_values_TCC1

Terse proof for range_int_values_TCC1.

range_int_values_TCC1:

{1} $\forall (i: \text{int}):$
 $-32767 \leq i \wedge i \leq 32767 \supset \text{Cpp_Type?}(\text{int}) \wedge \text{cv}(\text{non_bool_integral_enum?})(\text{int})$

Repeatedly Skolemizing and flattening,
 Keeping (1) and hiding *,
 Applying propositional simplification,
 we get 2 subgoals:

range_int_values_TCC1.1:

{1} Cpp_Type?(int)

Expanding the definition of Cpp_Type?,
 Repeatedly Skolemizing and flattening,
 Expanding the definition of subterm,
 Replacing using formula -1,

Installing automatic rewrites from: no_pointers_to_bitfield? no_pointers_to_references?
 no_cv_references? no_reference_to_reference? no_reference_to_bitfields? no_pointer_to_member_to_reference?
 no_pointer_to_member_to_cv_void? no_cv_void_parameter? no_array_of_references? no_array_of_cv_void?
 no_array_of_function? no_array_of_abstract_class? no_pointer_or_ref_to_incomplete_array_parameter?
 no_array_return_type? no_function_return_type? bitfield_underlying_integral_or_enum_type? cv_array?
 enum_underlying_integral? enum_constants? const_volatile? const_stutter? volatile_stutter?
 no_cv_class? no_cv_union? no_cv_function? no_reference_to_void? no_cv_void? no_array_of_bitfields?

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of range_int_values_TCC1.1.

range_int_values_TCC1.2:

{1} $\text{cv}(\text{non_bool_integral_enum?})(\text{int})$

Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of range_int_values_TCC1.2.
 Q.E.D.

C.42.22 Cpp_Examples.range_int_values

Terse proof for range_int_values.

range_int_values:

{1} $\forall (i: \text{int}): -32767 \leq i \wedge i \leq 32767 \supset \text{range}(\text{int})(i)$

Repeatedly Skolemizing and flattening,
 Expanding the definition of range,
 Expanding the definition of range_integral,
 Rewriting using int_binary, matching in *,
 Using lemma max_int,
 Using lemma min_int,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 This completes the proof of range_int_values.
 Q.E.D.

C.42.23 Cpp_Examples.spec08_TCC1

Terse proof for spec08_TCC1.

spec08_TCC1:

{1}	plain_memory?(pm) \supset Cpp_Type?(int) \wedge v(non_bool_integral?)(int)
-----	--

Repeatedly Skolemizing and flattening,
Applying propositional simplification,
we get 2 subgoals:

spec08_TCC1.1:

{-1}	plain_memory?(pm)
{1}	Cpp_Type?(int)

Expanding the definition of Cpp_Type?,
Repeatedly Skolemizing and flattening,
Expanding the definition of subterm,
Replacing using formula -1,

Installing automatic rewrites from: no_pointers_to_bitfield? no_pointers_to_references?
no_cv_references? no_reference_to_reference? no_reference_to_bitfields? no_pointer_to_member_to_reference?
no_pointer_to_member_to_cv_void? no_cv_void_parameter? no_array_of_references? no_array_of_cv_void?
no_array_of_function? no_array_of_abstract_class? no_pointer_or_ref_to_incomplete_array_parameter?
no_array_return_type? no_function_return_type? bitfield_underlying_integral_or_enum_type? cv_array?
enum_underlying_integral? enum_constants? const_volatile? const_stutter? volatile_stutter?
no_cv_class? no_cv_union? no_cv_function? no_reference_to_void? no_cv_void? no_array_of_bitfields?

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of spec08_TCC1.1.

spec08_TCC1.2:

{-1}	plain_memory?(pm)
{1}	v(non_bool_integral?)(int)

Hiding formulas: -1,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of spec08_TCC1.2.
Q.E.D.

C.42.24 Cpp_Examples.spec08

Terse proof for spec08.

spec08:

{1}	plain_memory?(pm) \supset valid(PRECONDITION, e2s(assign(pm, dt(int))(id(n), literal(0))) ## e2s(assign(pm, dt(int))(id(m), literal(0))) ## while(N + 1, lt(int)(l2r(pm, dt(int))(id(n)), literal(N)), e2s(postinc(pm, int)(id(n))) ## e2s(postinc(pm, int)(id(m))),), λ (s: State): OK?(read_data(pm, dt(int))(m)(s)) \supset data(read_data(pm, dt(int))(m)(s)) = N)
-----	---

Installing automatic rewrites from: valid Valid PRECONDITION POSTCONDITION

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Repeatedly Skolemizing and flattening,

Installing automatic rewrites from: address_of_spec address_of_spec2 add_simplification add_spec add_spec2 same_array_spec same_array_spec2 mod_range size_of uidt check_bounds_spec er_plus_ax er_minus_ax er_div_ax er_times_ax er_neg_ax extended_real_superset_of_number_fields

Installing automatic rewrites from: literal id member postinc postdec deref arrow unary_minus unary_minus_unsigned unary_not unary_bitnot preinc predec UnaryExpressions.sizeof ptm_member ptm_arrow arrow times div div_float mod plus plus_ptr postinc_ptr preinc_ptr subscript minus minus_ptr postdec_ptr predec_ptr minus_ptr_ptr cmp lt gt le ge lt_ptr gt_ptr le_ptr ge_ptr cmp_pointer eq_bool eq not_equal not_equal_bool eq_ptr not_equal_ptr bitand bitxor bitor and_exp or_exp conditional_exp assign assign_times assign_div assign_mod assign_div_float assign_plus assign_plus_ptr assign_minus assign_minus_ptr assign_bitand assign_bitxor assign_bitor comma

Installing automatic rewrites from: plain_memory_write_ok_single plain_memory_read_data_ok plain_memory_write_ok_q_expr plain_memory_write_ok_q_stmt in_blessed_memory_rw_ro pm_q_prop_ok pm_q_prop_single_read pm_q_prop_single_write pm_q_prop_read_write_single pm_q_prop_read_write_other_single pm_q_prop_read_read_single plain_memory_read_write_other_single_data plain_memory_read_read plain_memory_read_write_res pm_q_prop_read_ok_expr pm_q_prop_read_ok_stmt pm_q_prop_read_pm_q_prop_expr pm_q_prop_read_pm_q_prop_stmt pm_q_prop_read_write_q_expr pm_q_prop_read_write_q_stmt plain_memory_read_write_q_data_expr plain_memory_read_write_q_data_stmt pm_q_prop_read_write_other_q_expr pm_q_prop_read_write_other_q_stmt pm_q_prop_read_read_q_expr pm_q_prop_read_read_q_stmt plain_memory_read_write_other_q_data_expr plain_memory_read_write_other_q_data_stmt plain_memory_read_read_q_data_expr plain_memory_read_read_q_data_stmt pm_q_prop_ok_result_single pm_q_prop_ok_result_q_expr pm_q_prop_ok_result_q_stmt pm_q_prop_read_ok_result_single pm_q_prop_read_ok_result_q_expr pm_q_prop_read_ok_result_q_stmt pm_q_prop_e2s pm_q_prop_stmt_e2s pm_q_prop_read_e2s pm_q_prop_read_stmt_e2s pm_q_prop_read_ok_pm_q_prop_read_expr pm_q_prop_read_ok_pm_q_prop_read_stmt

Installing automatic rewrites from: if_else switch member break_catch_break break_break_stmt break_break_catch_continue break_break_catch_default break_break_lift_stmt break_break_case break_break_default break_break_stmt_break_break break_stmt_break break_stmt_catch_continue break_stmt_catch_default break_stmt_lift break_stmt_case break_stmt_default continue_catch_continue continue_break_stmt continue_break_catch_continue continue_break_catch_default continue_break_lift_stmt continue_break_case continue_break_default continue_continue stmt_continue_continue continue_stmt_continue continue_stmt_catch_continue continue_stmt_catch_default continue_stmt_lift continue_stmt_case continue_stmt_default switch_stmt stmt_switch_stmt switch_case_taken stmt_switch_case_taken switch_case_not_taken stmt_switch_case_not_taken switch_default stmt_switch_default switch_catch_break stmt_switch_catch_break switch_catch_continue stmt_switch_catch_continue switch_catch_default stmt_switch_catch_default default_stmt stmt_default_stmt default_case stmt_default_case default_default stmt_default_default default_catch_break stmt_default_catch_break default_catch_continue stmt_default_catch_continue default_catch_default stmt_default_catch_default return_void_result return_ex_result stmt_remove_catch_default stmt_remove_case stmt_remove_default stmt_remove_catch_continue stmt_remove_catch_break break_break_stmt_expr break_break_catch_continue_expr break_break_catch_default_expr break_break_lift_stmt_expr break_break_case_expr break_break_default_expr continue_break_stmt_expr continue_break_catch_continue_expr continue_break_catch_default_expr continue_break_lift_stmt_expr continue_break_case_expr continue_break_default_expr

Installing automatic rewrites from: skip_ok skip_state skip_elimination stmt_skip_elimination stmt_ok_lift stmt_ok_lift_expr e2s_ok stmt_e2s_ok e2s_state stmt_e2s_state ok_result_fexpr ok_result_fstmt e2s_merge stmt_e2s_merge e2s_expr stmt_e2s_expr

Installing automatic rewrites from: while_unroll stmt_while_unroll iterate_while_ok while_hang while_unroll_expr while_unroll_lexpr stmt_while_unroll_expr stmt_while_unroll_lexpr while_unroll_0 do_while_unroll do_while_inv_unroll for_unroll for_inv_unroll while_to_while_no_cb while_invariant?! while_variant?! while_inv_rewrite_termination_result! while_inv_rewrite_data_ok! while_inv_rewrite_data_break! while_inv_rewrite_data_return! stmt_while_inv_rewrite_termination_result! stmt_while_inv_rewrite_data_ok!

stmt_while_inv_rewrite_data_break! stmt_while_inv_rewrite_data_return! pm_q_prop_while!
 pm_q_prop_stmt_while!

Installing automatic rewrites from: composition_assoc_rewrite_stmt_ok forward_s_c_s_of_s_expr
 comp_eval_if_ok_fstmt stmt_eval_if_ok_fstmt comp_eval_if_ok_fstmt_cstmt stmt_eval_if_ok_fstmt_cstmt
 composition_assoc_rewrite_1 composition_assoc_rewrite_2 composition_assoc_rewrite_3 com-
 position_assoc_rewrite_4 composition_assoc_rewrite_5 composition_assoc_rewrite_6 composi-
 tion_assoc_rewrite_7 composition_assoc_rewrite_8 composition_assoc_rewrite_expr_1 composi-
 tion_assoc_rewrite_expr_2 composition_assoc_rewrite_expr_3 composition_assoc_rewrite_expr_4 composi-
 tion_assoc_rewrite_expr_5 composition_assoc_rewrite_expr_6 composition_assoc_rewrite_stmt_ok compo-
 sition_assoc_rewrite_stmt_ok_expr composition_assoc_rewrite_stmt_ok_expr_lexpr comp_eval_if_ok_fstmt
 comp_eval_if_ok_fstmt_expr comp_eval_if_ok_fstmt_cstmt_expr stmt_eval_if_ok_fstmt_expr stmt_eval_if_ok_fstmt_cst-
 comp_eval_if_ok_fexpr_expr comp_eval_if_ok_fexpr stmt_eval_if_ok_fexpr expr_eval_if_ok_fexpr compo-
 sition_assoc_expression_rewrite composition_assoc_expression_rewrite_stmt stmt_eval_if_ok_fexpr_expr
 expr_eval_if_ok_fexpr_expr ok_result_fexpr ok_result_fstmt composition_assoc_rewrite_stmt_ok_cstmt

Installing automatic rewrites from: allocate_stack deallocate_stack with_new_stackvar

Installing automatic rewrites from: ok_result_ok ok_result_data ok_result_q_ok ok_result_q_state
 ok_result_q_data ok_result_elimination_1 ok_result_elimination_2 ok_result_state

Installing automatic rewrites from: l2r integral_to_bool pointer_to_bool bool2int

Installing automatic rewrites from: comp_simple_expr_simple_expr_ok comp_simple_expr_simple_expr_data
 comp_simple_expr_simple_expr_state comp_simple_stmt_simple_expr_ok comp_simple_stmt_simple_expr_data
 comp_simple_stmt_simple_expr_state

Installing automatic rewrites from: floating_point?

Installing automatic rewrites from: N

Installing automatic rewrites from: range_int_values

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of spec08.

Q.E.D.

C.42.25 Cpp_Examples.spec09

Terse proof for spec09.

spec09:

```
{1} plain_memory?(pm)  $\supset$ 
    valid(PRECONDITION,
        e2s(assign(pm, dt(int))(id(m), literal(0))) ##
        for(N + 1, e2s(assign(pm, dt(int))(id(n), literal(0))),
            lt(int)(l2r(pm, dt(int))(id(n)), literal(N)),
            postinc(pm, int)(id(n)), e2s(postinc(pm, int)(id(m)))),
         $\lambda$  (s: State):
        OK?(read_data(pm, dt(int))(m)(s))  $\supset$ 
        data(read_data(pm, dt(int))(m)(s)) = N
```

Installing automatic rewrites from: valid Valid PRECONDITION POSTCONDITION

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Repeatedly Skolemizing and flattening,

Installing automatic rewrites from: address_of_spec address_of_spec2 add_simplification add_spec
 add_spec2 same_array_spec same_array_spec2 mod_range size_of uidt check_bounds_spec er_plus_ax
 er_minus_ax er_div_ax er_times_ax er_neg_ax extended_reals_superset_of_number_fields

Installing automatic rewrites from: literal id member postinc postdec deref arrow unary_minus
 unary_minus_unsigned unary_not unary_bitnot preinc predec UnaryExpressions.sizeof ptm_member
 ptm_arrow arrow times div div_float mod plus plus_ptr postinc_ptr preinc_ptr subscript minus minus_ptr

postdec_ptr predec_ptr minus_ptr_ptr cmp lt gt le ge lt_ptr gt_ptr le_ptr ge_ptr cmp_pointer eq_bool
 eq not_equal not_equal_bool eq_ptr not_equal_ptr bitand bitxor bitor and_exp or_exp conditional_expr
 assign assign_times assign_div assign_mod assign_div_float assign_plus assign_plus_ptr assign_minus as-
 sign_minus_ptr assign_bitand assign_bitxor assign_bitor comma

Installing automatic rewrites from: plain_memory_write_ok_single plain_memory_read_data_ok
 plain_memory_write_ok_q_expr plain_memory_write_ok_q_stmt in_blessed_memory_rw_ro pm_q_prop_ok
 pm_q_prop_single_read pm_q_prop_single_write pm_q_prop_read_write_single pm_q_prop_read_write_other_single
 pm_q_prop_read_read_single plain_memory_read_write_other_single_data plain_memory_read_read
 plain_memory_read_write_res pm_q_prop_read_ok_expr pm_q_prop_read_ok_stmt pm_q_prop_read_pm_q_prop_expr
 pm_q_prop_read_pm_q_prop_stmt pm_q_prop_read_write_q_expr pm_q_prop_read_write_q_stmt
 plain_memory_read_write_q_data_expr plain_memory_read_write_q_data_stmt pm_q_prop_read_write_other_q_expr
 pm_q_prop_read_write_other_q_stmt pm_q_prop_read_read_q_expr pm_q_prop_read_read_q_stmt
 plain_memory_read_write_other_q_data_expr plain_memory_read_write_other_q_data_stmt plain_memory_read_read_q_data_expr
 plain_memory_read_read_q_data_stmt pm_q_prop_ok_result_single pm_q_prop_ok_result_q_expr
 pm_q_prop_ok_result_q_stmt pm_q_prop_read_ok_result_single pm_q_prop_read_ok_result_q_expr
 pm_q_prop_read_ok_result_q_stmt pm_q_prop_e2s pm_q_prop_stmt_e2s pm_q_prop_read_e2s
 pm_q_prop_read_stmt_e2s pm_q_prop_read_ok_pm_q_prop_read_expr pm_q_prop_read_ok_pm_q_prop_read_stmt

Installing automatic rewrites from: if_else switch member break_catch_break break_break_stmt
 break_break_catch_continue break_break_catch_default break_break_lift_stmt break_break_case
 break_break_default break_break_stmt_break_break break_stmt_break break_stmt_catch_continue
 break_stmt_catch_default break_stmt_lift break_stmt_case break_stmt_default continue_catch_continue
 continue_break_stmt continue_break_catch_continue continue_break_catch_default continue_break_lift_stmt
 continue_break_case continue_break_default continue_continue stmt_continue_continue con-
 tinue_stmt_continue continue_stmt_catch_continue continue_stmt_catch_default continue_stmt_lift
 continue_stmt_case continue_stmt_default switch_stmt stmt_switch_stmt switch_case_taken
 stmt_switch_case_taken switch_case_not_taken stmt_switch_case_not_taken switch_default stmt_switch_default
 switch_catch_break stmt_switch_catch_break switch_catch_continue stmt_switch_catch_continue
 switch_catch_default stmt_switch_catch_default default_stmt stmt_default_stmt default_case
 stmt_default_case default_default stmt_default_default default_catch_break stmt_default_catch_break
 default_catch_continue stmt_default_catch_continue default_catch_default stmt_default_catch_default
 return_void_result return_ex_result stmt_remove_catch_default stmt_remove_case stmt_remove_default
 stmt_remove_catch_continue stmt_remove_catch_break break_break_stmt_expr break_break_catch_continue_expr
 break_break_catch_default_expr break_break_lift_stmt_expr break_break_case_expr break_break_default_expr
 continue_break_stmt_expr continue_break_catch_continue_expr continue_break_catch_default_expr con-
 tinue_break_lift_stmt_expr continue_break_case_expr continue_break_default_expr

Installing automatic rewrites from: skip_ok skip_state skip_elimination stmt_skip_elimination
 stmt_ok_lift stmt_ok_lift_expr e2s_ok stmt_e2s_ok e2s_state stmt_e2s_state ok_result_fexpr
 ok_result_fstmt e2s_merge stmt_e2s_merge e2s_expr stmt_e2s_expr

Installing automatic rewrites from: while_unroll stmt_while_unroll iterate_while_ok while_hang
 while_unroll_expr while_unroll_lexpr stmt_while_unroll_expr stmt_while_unroll_lexpr while_unroll_0
 do_while_unroll do_while_inv_unroll for_unroll for_inv_unroll while_to_while_no_cb while_invariant?!
 while_variant?! while_inv_rewrite_termination_result! while_inv_rewrite_data_ok! while_inv_rewrite_data_break!
 while_inv_rewrite_data_return! stmt_while_inv_rewrite_termination_result! stmt_while_inv_rewrite_data_ok!
 stmt_while_inv_rewrite_data_break! stmt_while_inv_rewrite_data_return! pm_q_prop_while!
 pm_q_prop_stmt_while!

Installing automatic rewrites from: composition_assoc_rewrite_stmt_ok forward_s_c_s_of_s_expr
 comp_eval_if_ok_fstmt stmt_eval_if_ok_fstmt comp_eval_if_ok_fstmt_cstmt stmt_eval_if_ok_fstmt_cstmt
 composition_assoc_rewrite_1 composition_assoc_rewrite_2 composition_assoc_rewrite_3 com-
 position_assoc_rewrite_4 composition_assoc_rewrite_5 composition_assoc_rewrite_6 composi-
 tion_assoc_rewrite_7 composition_assoc_rewrite_8 composition_assoc_rewrite_expr_1 composi-
 tion_assoc_rewrite_expr_2 composition_assoc_rewrite_expr_3 composition_assoc_rewrite_expr_4 composi-
 tion_assoc_rewrite_expr_5 composition_assoc_rewrite_expr_6 composition_assoc_rewrite_stmt_ok compo-

C Proof scripts

sition_assoc_rewrite_stmt_ok_expr composition_assoc_rewrite_stmt_ok_expr_lexpr comp_eval_if_ok_fstmt
comp_eval_if_ok_fstmt_expr comp_eval_if_ok_fstmt_cstmt_expr stmt_eval_if_ok_fstmt_expr stmt_eval_if_ok_fstmt_cstr
comp_eval_if_ok_fexpr_expr comp_eval_if_ok_fexpr stmt_eval_if_ok_fexpr expr_eval_if_ok_fexpr compo-
sition_assoc_expression_rewrite composition_assoc_expression_rewrite_stmt stmt_eval_if_ok_fexpr_expr
expr_eval_if_ok_fexpr_expr ok_result_fexpr ok_result_fstmt composition_assoc_rewrite_stmt_ok_cstmt

Installing automatic rewrites from: allocate_stack deallocate_stack with_new_stackvar

Installing automatic rewrites from: ok_result_ok ok_result_data ok_result_q_ok ok_result_q_state
ok_result_q_data ok_result_elimination_1 ok_result_elimination_2 ok_result_state

Installing automatic rewrites from: l2r integral_to_bool pointer_to_bool bool2int

Installing automatic rewrites from: comp_simple_expr_simple_expr_ok comp_simple_expr_simple_expr_data
comp_simple_expr_simple_expr_state comp_simple_stmt_simple_expr_ok comp_simple_stmt_simple_expr_data
comp_simple_stmt_simple_expr_state

Installing automatic rewrites from: floating_point?

Installing automatic rewrites from: N

Installing automatic rewrites from: range_int_values

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `spec09`.

Q.E.D.

C.42.26 Cpp_Examples.spec10_TCC1

Terse proof for `spec10_TCC1`.

`spec10_TCC1`:

```
{1} plain_memory?(pm)  $\supset$ 
  ( $\forall$  (s0: State, res: StmtResult[State]):
    OK?(res)  $\supset$ 
      OK?[State](res)  $\vee$ 
      Break?[State](res)  $\vee$ 
      Continue?[State](res)  $\vee$ 
      Return?[State](res)  $\vee$ 
      Switch?[State](res)  $\vee$  Default?[State](res)  $\vee$  Exception?[State](res))
```

Repeatedly Skolemizing and flattening,

This completes the proof of `spec10_TCC1`.

Q.E.D.

C.42.27 Cpp_Examples.spec10

Terse proof for `spec10`.

spec10:

```

{1} plain_memory?(pm)  $\supset$ 
  valid(PRECONDITION,
    for(e2s(assign(pm, dt(int))(id(n), literal(0))),
      lt(int)(l2r(pm, dt(int))(id(n)), literal(N)),
      postinc(pm, int)(id(n)), skip)
    (( $\lambda$  (s0: State):
       $\lambda$  (res: StmtResult[State]):
        OK?(res)  $\wedge$ 
        OK?(read_data(pm, dt(int))(n)(state(res)))  $\wedge$ 
        0  $\leq$  data(read_data(pm, dt(int))(n)(state(res)))  $\wedge$ 
        data(read_data(pm, dt(int))(n)(state(res)))  $<$  N),
      ( $\lambda$  (s0: State):
         $\lambda$  (res: StmtResult[State]):
          OK?(res)  $\wedge$ 
          OK?(read_data(pm, dt(int))(n)(state(res)))  $\wedge$ 
          data(read_data(pm, dt(int))(n)(state(res))) = N),
      ( $\lambda$  (s0: State):
         $\lambda$  (res: StmtResult[State]):
          OK?(res)  $\wedge$ 
          OK?(read_data(pm, dt(int))(n)(state(res)))  $\wedge$ 
          data(read_data(pm, dt(int))(n)(state(res)))  $\leq$  N),
      ( $\lambda$  (s0: State):
         $\lambda$  (s: State):
          IF OK?(read_data(pm, dt(int))(n)(s))  $\wedge$ 
            data(read_data(pm, dt(int))(n)(s))  $\leq$  N
            THEN N - data(read_data(pm, dt(int))(n)(s))
            ELSE N
          ENDIF),
      extend([[numfield, numfield], [int, int], bool, FALSE]
        ( $\lambda$  (i, j: int): i < j)),
       $\lambda$  (s: State):
        OK?(read_data(pm, dt(int))(n)(s))  $\supset$ 
        data(read_data(pm, dt(int))(n)(s)) = N)
  )

```

This completes the proof of spec10.

C.43 Proofs for Cpp_Integral_Types (types.pvs)

C.43.1 Cpp_Integral_Types.range_integral_TCC1

Terse proof for range_integral_TCC1.

range_integral_TCC1:

```

{1}  $\forall$  (typ: (non_bool_integral?):
  typ = uchar  $\supset$   $\neg$  empty?[int](extend[int, nat, bool, FALSE](range_uchar))

```

Repeatedly Skolemizing and flattening,
 Adding type constraints for range_uchar,
 Keeping (-4 1) and hiding *,

Expanding the definition of `empty?`,
Repeatedly Skolemizing and flattening,
Instantiating quantified variables,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `range_integral_TCC1`.
Q.E.D.

C.43.2 Cpp_Integral_Types.range_integral_TCC2

Terse proof for `range_integral_TCC2`.

`range_integral_TCC2`:

$$\{1\} \quad \forall (\text{typ}: (\text{non_bool_integral?})): \\ \text{typ} = \text{ushort} \supset \neg \text{empty?}[\text{int}](\text{extend}[\text{int}, \text{nat}, \text{bool}, \text{FALSE}](\text{range_ushort}))$$

Repeatedly Skolemizing and flattening,
Adding type constraints for `range_ushort`,
Keeping (-4 1) and hiding *,
Expanding the definition of `empty?`,
Repeatedly Skolemizing and flattening,
Instantiating quantified variables,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `range_integral_TCC2`.
Q.E.D.

C.43.3 Cpp_Integral_Types.range_integral_TCC3

Terse proof for `range_integral_TCC3`.

`range_integral_TCC3`:

$$\{1\} \quad \forall (\text{typ}: (\text{non_bool_integral?})): \\ \text{typ} = \text{uint} \supset \neg \text{empty?}[\text{int}](\text{extend}[\text{int}, \text{nat}, \text{bool}, \text{FALSE}](\text{range_uint}))$$

Repeatedly Skolemizing and flattening,
Adding type constraints for `range_uint`,
Keeping (-4 1) and hiding *,
Expanding the definition of `empty?`,
Repeatedly Skolemizing and flattening,
Instantiating quantified variables,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `range_integral_TCC3`.
Q.E.D.

C.43.4 Cpp_Integral_Types.range_integral_TCC4

Terse proof for `range_integral_TCC4`.

`range_integral_TCC4`:

$$\{1\} \quad \forall (\text{typ}: (\text{non_bool_integral?})): \\ \text{typ} = \text{ulong} \supset \neg \text{empty?}[\text{int}](\text{extend}[\text{int}, \text{nat}, \text{bool}, \text{FALSE}](\text{range_ulong}))$$

Repeatedly Skolemizing and flattening,
 Adding type constraints for range_ulong,
 Keeping (-4 1) and hiding *,
 Expanding the definition of empty?,
 Repeatedly Skolemizing and flattening,
 Instantiating quantified variables,
 Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of range_integral_TCC4.
 Q.E.D.

C.43.5 Cpp_Integral_Types.range_integral_TCC5

Terse proof for range_integral_TCC5.

range_integral_TCC5:

```
{1}  ∀ (typ: (non_bool_integral?):
      typ = ulonglong ⊃ ¬ empty?[int](extend[int, nat, bool, FALSE](range_ulonglong))
```

Repeatedly Skolemizing and flattening,
 Adding type constraints for range_ulonglong,
 Keeping (-4 1) and hiding *,
 Expanding the definition of empty?,
 Repeatedly Skolemizing and flattening,
 Instantiating quantified variables,
 Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of range_integral_TCC5.
 Q.E.D.

C.43.6 Cpp_Integral_Types.range_integral_TCC6

Terse proof for range_integral_TCC6.

range_integral_TCC6:

```
{1}  ∀ (typ: (non_bool_integral?):
      ¬ (bool?(typ) ∨
         float?(typ) ∨
         double?(typ) ∨
         longdouble?(typ) ∨
         void?(typ) ∨
         array?(typ) ∨
         function?(typ) ∨
         pointer?(typ) ∨
         reference?(typ) ∨
         class?(typ) ∨
         union?(typ) ∨
         bitfield?(typ) ∨
         enum?(typ) ∨ pointer_to_member?(typ) ∨ const?(typ) ∨ volatile?(typ))
```

Repeatedly Skolemizing and flattening,
 Expanding the definition of non_bool_integral?,
 Expanding the definition of signed_integer?,
 Expanding the definition of unsigned_integer?,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `range_integral_TCC6`.
Q.E.D.

C.43.7 Cpp_Integral_Types.range_floating_point_TCC1

Terse proof for `range_floating_point_TCC1`.

`range_floating_point_TCC1`:

```
{1}  ∀ (typ: (floating_point?):
      ¬ (uchar?(typ) ∨
         schar?(typ) ∨
         char?(typ) ∨
         short?(typ) ∨
         int?(typ) ∨
         long?(typ) ∨
         longlong?(typ) ∨
         ushort?(typ) ∨
         uint?(typ) ∨
         ulong?(typ) ∨
         ulonglong?(typ) ∨
         wchar_t?(typ) ∨
         bool?(typ) ∨
         void?(typ) ∨
         array?(typ) ∨
         function?(typ) ∨
         pointer?(typ) ∨
         reference?(typ) ∨
         class?(typ) ∨
         union?(typ) ∨
         bitfield?(typ) ∨
         enum?(typ) ∨
         pointer_to_member?(typ) ∨ const?(typ) ∨ volatile?(typ))
```

Repeatedly Skolemizing and flattening,
Expanding the definition of `floating_point?`,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `range_floating_point_TCC1`.
Q.E.D.

C.43.8 Cpp_Integral_Types.dt_integral_TCC1

Terse proof for `dt_integral_TCC1`.

`dt_integral_TCC1`:

```
{1}  ∀ (typ: (non_bool_integral?):
      typ = uchar ⊃
      (∀ (x: int): x ≥ 0 ∧ (range_uchar)(x) ≡ range_integral(typ)(x)) ∧
      (∀ (x: int): x ≥ 0 ∧ (range_uchar)(x) ≡ range_integral(typ)(x)) ∧
      (∀ (x1: [list[Byte], Address]):
         every[(range_uchar)]((range_integral(typ))(dt_uchar 'from_byte(x1)))
         ∧ pod_data_type?[(range_integral(typ))](dt_uchar)
```

Repeatedly Skolemizing and flattening,
 Replacing using formula -2,
 Expanding the definition of range_integral,
 Expanding the definition of extend,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 we get 3 subgoals:

dt_integral_TCC1.1.1:

{-1}	non_bool_integral?(uchar)
{-2}	uchar?(typ')
{1}	pod_data_type?[(range_integral(typ'))](dt_uchar)

Adding type constraints for dt_uchar,
 Expanding the definition of pod_data_type?,
 Applying propositional simplification,
 Expanding the definition of interpreted_data_type?,
 Applying propositional simplification,
 we get 2 subgoals:

dt_integral_TCC1.1.1.1:

{-1}	uninterpreted_data_type?(dt_uchar 'uidt)
{-2}	$\forall (d: ((range_uchar)), a: Address):$ valid?(uidt(dt_uchar))(to_byte(dt_uchar)(d, a), a)
{-3}	$\forall (l: list[Byte], a: Address):$ valid?(uidt(dt_uchar))(l, a) \equiv up?(from_byte(dt_uchar)(l, a))
{-4}	$\forall (d: ((range_uchar)), a: Address):$ down(from_byte(dt_uchar)(to_byte(dt_uchar)(d, a), a)) = d
{-5}	$\forall (l: list[Byte], a_1, a_2: Address):$ valid?(uidt(dt_uchar))(l, a_1) \equiv valid?(uidt(dt_uchar))(l, a_2)
{-6}	size(uidt(dt_uchar)) > 0
{-7}	non_bool_integral?(uchar)
{-8}	uchar?(typ')
{1}	$\forall (d: ((range_integral(typ'))), a: Address):$ valid?(uidt(dt_uchar))(to_byte(dt_uchar)(d, a), a)

Repeatedly Skolemizing and flattening,
 Instantiating quantified variables,
 Expanding the definition of range_integral,
 Expanding the definition of extend,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 This completes the proof of dt_integral_TCC1.1.1.

dt_integral_TCC1.1.2:

{-1}	uninterpreted_data_type?(dt_uchar'uidt)
{-2}	$\forall (d: ((\text{range_uchar})), a: \text{Address}):$ valid?(uidt(dt_uchar))(to_byte(dt_uchar)(d, a), a)
{-3}	$\forall (l: \text{list}[\text{Byte}], a: \text{Address}):$ valid?(uidt(dt_uchar))(l, a) \equiv up?(from_byte(dt_uchar)(l, a))
{-4}	$\forall (d: ((\text{range_uchar})), a: \text{Address}):$ down(from_byte(dt_uchar)(to_byte(dt_uchar)(d, a), a)) = d
{-5}	$\forall (l: \text{list}[\text{Byte}], a_1, a_2: \text{Address}):$ valid?(uidt(dt_uchar))(l, a_1) \equiv valid?(uidt(dt_uchar))(l, a_2)
{-6}	size(uidt(dt_uchar)) > 0
{-7}	non_bool_integral?(uchar)
{-8}	uchar?(typ')
{1}	
	$\forall (d: ((\text{range_integral}(\text{typ}')), a: \text{Address}):$ down(from_byte(dt_uchar)(to_byte(dt_uchar)(d, a), a)) = d

Repeatedly Skolemizing and flattening,
Instantiating quantified variables,
Expanding the definition of range_integral,
Expanding the definition of extend,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of dt_integral_TCC1.1.2.

dt_integral_TCC1.2:

{-1}	non_bool_integral?(uchar)
{-2}	uchar?(typ')
{1}	
	$\forall (x_1: [\text{list}[\text{Byte}], \text{Address}]):$ every[(range_uchar)]((range_integral(uchar)))(dt_uchar'from_byte(x_1))

Repeatedly Skolemizing and flattening,
Expanding the definition of every,
Expanding the definition of range_integral,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Expanding the definition of extend,
which is trivially true.
This completes the proof of dt_integral_TCC1.2.

dt_integral_TCC1.3:

{-1}	non_bool_integral?(uchar)
{-2}	uchar?(typ')
{1}	
	$\forall (x: \text{int}):$ $x \geq 0 \wedge (\text{range_uchar})(x) \equiv \text{IF } x \geq 0 \text{ THEN range_uchar}(x) \text{ ELSE FALSE ENDIF}$

Repeatedly Skolemizing and flattening,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of dt_integral_TCC1.3.
Q.E.D.

C.43.9 Cpp_Integral_Types.dt_integral_TCC2

Terse proof for dt_integral_TCC2.

dt_integral_TCC2:

{1}	$\forall (\text{typ}: (\text{non_bool_integral?})): \\ \text{typ} = \text{schar} \supset \\ (\forall (x: \text{int}): (\text{range_schar})(x) \equiv \text{range_integral}(\text{typ})(x)) \wedge \\ (\forall (x: \text{int}): (\text{range_schar})(x) \equiv \text{range_integral}(\text{typ})(x)) \wedge \\ (\forall (x_1: [\text{list}[\text{Byte}], \text{Address}]): \\ \text{every}[(\text{range_schar})](\text{range_integral}(\text{typ}))(\text{dt_schar}'\text{from_byte}(x_1))) \\ \wedge \text{pod_data_type?}[(\text{range_integral}(\text{typ}))](\text{dt_schar})$
-----	---

Repeatedly Skolemizing and flattening,

Replacing using formula -2,

Expanding the definition of range_integral,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 2 subgoals:

dt_integral_TCC2.1:

{-1}	non_bool_integral?(schar)
{-2}	schar?(typ')
{1}	pod_data_type?[(range_integral(typ'))](dt_schar)

Adding type constraints for dt_schar,

Expanding the definition of pod_data_type?,

Applying propositional simplification,

Expanding the definition of interpreted_data_type?,

Applying propositional simplification,

we get 2 subgoals:

dt_integral_TCC2.1.1:

{-1}	uninterpreted_data_type?(dt_schar'uidt)
{-2}	$\forall (d: ((\text{range_schar})), a: \text{Address}): \\ \text{valid?}(\text{uidt}(\text{dt_schar}))(\text{to_byte}(\text{dt_schar})(d, a), a)$
{-3}	$\forall (l: \text{list}[\text{Byte}], a: \text{Address}): \\ \text{valid?}(\text{uidt}(\text{dt_schar}))(l, a) \equiv \text{up?}(\text{from_byte}(\text{dt_schar}))(l, a)$
{-4}	$\forall (d: ((\text{range_schar})), a: \text{Address}): \\ \text{down}(\text{from_byte}(\text{dt_schar}))(\text{to_byte}(\text{dt_schar})(d, a), a) = d$
{-5}	$\forall (l: \text{list}[\text{Byte}], a_1, a_2: \text{Address}): \\ \text{valid?}(\text{uidt}(\text{dt_schar}))(l, a_1) \equiv \text{valid?}(\text{uidt}(\text{dt_schar}))(l, a_2)$
{-6}	size(uidt(dt_schar)) > 0
{-7}	non_bool_integral?(schar)
{-8}	schar?(typ')
{1}	$\forall (d: ((\text{range_integral}(\text{typ}'))), a: \text{Address}): \\ \text{valid?}(\text{uidt}(\text{dt_schar}))(\text{to_byte}(\text{dt_schar})(d, a), a)$

Repeatedly Skolemizing and flattening,

Instantiating quantified variables,

Expanding the definition of range_integral,

which is trivially true.

This completes the proof of dt_integral_TCC2.1.1.

dt_integral_TCC2.1.2:

{-1}	uninterpreted_data_type?(dt_schar 'uidt)
{-2}	$\forall (d: ((\text{range_schar})), a: \text{Address}):$ valid?(uidt(dt_schar))(to_byte(dt_schar)(d, a), a)
{-3}	$\forall (l: \text{list}[\text{Byte}], a: \text{Address}):$ valid?(uidt(dt_schar))(l, a) \equiv up?(from_byte(dt_schar)(l, a))
{-4}	$\forall (d: ((\text{range_schar})), a: \text{Address}):$ down(from_byte(dt_schar)(to_byte(dt_schar)(d, a), a)) = d
{-5}	$\forall (l: \text{list}[\text{Byte}], a_1, a_2: \text{Address}):$ valid?(uidt(dt_schar))(l, a_1) \equiv valid?(uidt(dt_schar))(l, a_2)
{-6}	size(uidt(dt_schar)) > 0
{-7}	non_bool_integral?(schar)
{-8}	schar?(typ')
{1}	
	$\forall (d: ((\text{range_integral}(\text{typ}')), a: \text{Address}):$ down(from_byte(dt_schar)(to_byte(dt_schar)(d, a), a)) = d

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 Repeatedly Skolemizing and flattening,
 Instantiating quantified variables,
 Expanding the definition of range_integral,
 which is trivially true.

This completes the proof of dt_integral_TCC2.1.2.

dt_integral_TCC2.2:

{-1}	non_bool_integral?(schar)
{-2}	schar?(typ')
{1}	
	$\forall (x_1: [\text{list}[\text{Byte}], \text{Address}]):$ every[(range_schar)]((range_integral(schar)))(dt_schar 'from_byte(x_1))

Repeatedly Skolemizing and flattening,
 Expanding the definition of every,
 Expanding the definition of range_integral,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 This completes the proof of dt_integral_TCC2.2.
 Q.E.D.

C.43.10 Cpp_Integral_Types.dt_integral_TCC3

Terse proof for dt_integral_TCC3.

dt_integral_TCC3:

{1}	$\forall (\text{typ}: (\text{non_bool_integral?})):$ typ = char \supset ($\forall (x: \text{int}): (\text{range_char})(x) \equiv \text{range_integral}(\text{typ})(x)$) \wedge ($\forall (x: \text{int}): (\text{range_char})(x) \equiv \text{range_integral}(\text{typ})(x)$) \wedge ($\forall (x_1: [\text{list}[\text{Byte}], \text{Address}]):$ every[(range_char)]((range_integral(typ)))(dt_char 'from_byte(x_1))) \wedge pod_data_type?[(range_integral(typ))](dt_char)
-----	---

Repeatedly Skolemizing and flattening,
 Replacing using formula -2,
 Expanding the definition of range_integral,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 2 subgoals:

`dt_integral_TCC3.1:`

{-1}	<code>non_bool_integral?(char)</code>
{-2}	<code>char?(typ')</code>
{1}	<code>pod_data_type?[((range_integral(typ')))](dt_char)</code>

Adding type constraints for `dt_char`,

Expanding the definition of `pod_data_type?`,

Applying propositional simplification,

Expanding the definition of `interpreted_data_type?`,

Applying propositional simplification,

we get 2 subgoals:

`dt_integral_TCC3.1.1:`

{-1}	<code>uninterpreted_data_type?(dt_char 'uidt)</code>
{-2}	$\forall (d: ((range_char)), a: Address):$ <code>valid?(uidt(dt_char))(to_byte(dt_char)(d, a), a)</code>
{-3}	$\forall (l: list[Byte], a: Address):$ <code>valid?(uidt(dt_char))(l, a) \equiv up?(from_byte(dt_char)(l, a))</code>
{-4}	$\forall (d: ((range_char)), a: Address):$ <code>down(from_byte(dt_char)(to_byte(dt_char)(d, a), a)) = d</code>
{-5}	$\forall (l: list[Byte], a_1, a_2: Address):$ <code>valid?(uidt(dt_char))(l, a_1) \equiv valid?(uidt(dt_char))(l, a_2)</code>
{-6}	<code>size(uidt(dt_char)) > 0</code>
{-7}	<code>non_bool_integral?(char)</code>
{-8}	<code>char?(typ')</code>
{1}	$\forall (d: ((range_integral(typ'))), a: Address):$ <code>valid?(uidt(dt_char))(to_byte(dt_char)(d, a), a)</code>

Repeatedly Skolemizing and flattening,

Instantiating quantified variables,

Expanding the definition of `range_integral`,

which is trivially true.

This completes the proof of `dt_integral_TCC3.1.1`.

`dt_integral_TCC3.1.2:`

{-1}	<code>uninterpreted_data_type?(dt_char 'uidt)</code>
{-2}	$\forall (d: ((range_char)), a: Address):$ <code>valid?(uidt(dt_char))(to_byte(dt_char)(d, a), a)</code>
{-3}	$\forall (l: list[Byte], a: Address):$ <code>valid?(uidt(dt_char))(l, a) \equiv up?(from_byte(dt_char)(l, a))</code>
{-4}	$\forall (d: ((range_char)), a: Address):$ <code>down(from_byte(dt_char)(to_byte(dt_char)(d, a), a)) = d</code>
{-5}	$\forall (l: list[Byte], a_1, a_2: Address):$ <code>valid?(uidt(dt_char))(l, a_1) \equiv valid?(uidt(dt_char))(l, a_2)</code>
{-6}	<code>size(uidt(dt_char)) > 0</code>
{-7}	<code>non_bool_integral?(char)</code>
{-8}	<code>char?(typ')</code>
{1}	$\forall (d: ((range_integral(typ'))), a: Address):$ <code>down(from_byte(dt_char)(to_byte(dt_char)(d, a), a)) = d</code>

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Repeatedly Skolemizing and flattening,

Instantiating quantified variables,

Expanding the definition of `range_integral`,

which is trivially true.

This completes the proof of `dt_integral_TCC3.1.2`.

`dt_integral_TCC3.2`:

$$\frac{\begin{array}{l} \{-1\} \text{ non_bool_integral?}(\text{char}) \\ \{-2\} \text{ char?}(\text{typ}') \end{array}}{\begin{array}{l} \{1\} \forall (x_1: [\text{list}[\text{Byte}], \text{Address}]): \\ \text{every}[(\text{range_char})](\text{range_integral}(\text{char}))(\text{dt_char}'\text{from_byte}(x_1)) \end{array}}$$

Repeatedly Skolemizing and flattening,

Expanding the definition of every,

Expanding the definition of range_integral,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `dt_integral_TCC3.2`.

Q.E.D.

C.43.11 Cpp_Integral_Types.dt_integral_TCC4

Terse proof for `dt_integral_TCC4`.

`dt_integral_TCC4`:

$$\frac{\begin{array}{l} \{1\} \forall (\text{typ}: (\text{non_bool_integral?})): \\ \text{typ} = \text{short} \supset \\ (\forall (x: \text{int}): (\text{range_short})(x) \equiv \text{range_integral}(\text{typ})(x)) \wedge \\ (\forall (x: \text{int}): (\text{range_short})(x) \equiv \text{range_integral}(\text{typ})(x)) \wedge \\ (\forall (x_1: [\text{list}[\text{Byte}], \text{Address}]): \\ \text{every}[(\text{range_short})](\text{range_integral}(\text{typ}))(\text{dt_short}'\text{from_byte}(x_1))) \\ \wedge \text{pod_data_type?}[(\text{range_integral}(\text{typ}))](\text{dt_short}) \end{array}}$$

Repeatedly Skolemizing and flattening,

Replacing using formula -2,

Expanding the definition of range_integral,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 2 subgoals:

`dt_integral_TCC4.1`:

$$\frac{\begin{array}{l} \{-1\} \text{ non_bool_integral?}(\text{short}) \\ \{-2\} \text{ short?}(\text{typ}') \end{array}}{\begin{array}{l} \{1\} \text{ pod_data_type?}[(\text{range_integral}(\text{typ}'))](\text{dt_short}) \end{array}}$$

Adding type constraints for `dt_short`,

Expanding the definition of `pod_data_type?`,

Applying propositional simplification,

Expanding the definition of `interpreted_data_type?`,

Applying propositional simplification,

we get 2 subgoals:

dt_integral_TCC4.1.1:

{-1}	uninterpreted_data_type?(dt_short 'uidt)
{-2}	$\forall (d: ((\text{range_short})), a: \text{Address}):$ valid?(uidt(dt_short))(to_byte(dt_short)(d, a), a)
{-3}	$\forall (l: \text{list}[\text{Byte}], a: \text{Address}):$ valid?(uidt(dt_short))(l, a) \equiv up?(from_byte(dt_short)(l, a))
{-4}	$\forall (d: ((\text{range_short})), a: \text{Address}):$ down(from_byte(dt_short)(to_byte(dt_short)(d, a), a)) = d
{-5}	$\forall (l: \text{list}[\text{Byte}], a_1, a_2: \text{Address}):$ valid?(uidt(dt_short))(l, a_1) \equiv valid?(uidt(dt_short))(l, a_2)
{-6}	size(uidt(dt_short)) > 0
{-7}	non_bool_integral?(short)
{-8}	short?(typ')
{1}	$\forall (d: ((\text{range_integral}(\text{typ}')), a: \text{Address}):$ valid?(uidt(dt_short))(to_byte(dt_short)(d, a), a)

Repeatedly Skolemizing and flattening,

Instantiating quantified variables,

Expanding the definition of range_integral,

which is trivially true.

This completes the proof of dt_integral_TCC4.1.1.

dt_integral_TCC4.1.2:

{-1}	uninterpreted_data_type?(dt_short 'uidt)
{-2}	$\forall (d: ((\text{range_short})), a: \text{Address}):$ valid?(uidt(dt_short))(to_byte(dt_short)(d, a), a)
{-3}	$\forall (l: \text{list}[\text{Byte}], a: \text{Address}):$ valid?(uidt(dt_short))(l, a) \equiv up?(from_byte(dt_short)(l, a))
{-4}	$\forall (d: ((\text{range_short})), a: \text{Address}):$ down(from_byte(dt_short)(to_byte(dt_short)(d, a), a)) = d
{-5}	$\forall (l: \text{list}[\text{Byte}], a_1, a_2: \text{Address}):$ valid?(uidt(dt_short))(l, a_1) \equiv valid?(uidt(dt_short))(l, a_2)
{-6}	size(uidt(dt_short)) > 0
{-7}	non_bool_integral?(short)
{-8}	short?(typ')
{1}	$\forall (d: ((\text{range_integral}(\text{typ}')), a: \text{Address}):$ down(from_byte(dt_short)(to_byte(dt_short)(d, a), a)) = d

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Repeatedly Skolemizing and flattening,

Instantiating quantified variables,

Expanding the definition of range_integral,

which is trivially true.

This completes the proof of dt_integral_TCC4.1.2.

dt_integral_TCC4.2:

{-1}	non_bool_integral?(short)
{-2}	short?(typ')
{1}	$\forall (x_1: [\text{list}[\text{Byte}], \text{Address}]):$ every[(range_short)]((range_integral(short))(dt_short 'from_byte(x_1)))

Repeatedly Skolemizing and flattening,

Expanding the definition of every,

Expanding the definition of range_integral,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `dt_integral_TCC4.2`.
 Q.E.D.

C.43.12 Cpp_Integral_Types.dt_integral_TCC5

Terse proof for `dt_integral_TCC5`.

`dt_integral_TCC5`:

<pre>{1} ∀ (typ: (non_bool_integral?): typ = int ⊃ (∀ (x: int): (range_int)(x) ≡ range_integral(typ)(x)) ∧ (∀ (x: int): (range_int)(x) ≡ range_integral(typ)(x)) ∧ (∀ (x₁: [list[Byte], Address]): every[(range_int)]((range_integral(typ))(dt_int 'from_byte(x₁))) ∧ pod_data_type?[(range_integral(typ))](dt_int)</pre>
--

Repeatedly Skolemizing and flattening,
 Replacing using formula -2,
 Expanding the definition of `range_integral`,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 we get 2 subgoals:

`dt_integral_TCC5.1`:

<pre>{-1} non_bool_integral?(int) {-2} int?(typ')</pre>
<pre>{1} pod_data_type?[(range_integral(typ'))](dt_int)</pre>

Adding type constraints for `dt_int`,
 Expanding the definition of `pod_data_type?`,
 Applying propositional simplification,
 Expanding the definition of `interpreted_data_type?`,
 Applying propositional simplification,
 we get 2 subgoals:

`dt_integral_TCC5.1.1`:

<pre>{-1} uninterpreted_data_type?(dt_int 'uidt) {-2} ∀ (d: ((range_int)), a: Address): valid?(uidt(dt_int))(to_byte(dt_int)(d, a), a) {-3} ∀ (l: list[Byte], a: Address): valid?(uidt(dt_int))(l, a) ≡ up?(from_byte(dt_int)(l, a)) {-4} ∀ (d: ((range_int)), a: Address): down(from_byte(dt_int)(to_byte(dt_int)(d, a), a)) = d {-5} ∀ (l: list[Byte], a₁, a₂: Address): valid?(uidt(dt_int))(l, a₁) ≡ valid?(uidt(dt_int))(l, a₂) {-6} size(uidt(dt_int)) > 0 {-7} non_bool_integral?(int) {-8} int?(typ')</pre>
<pre>{1} ∀ (d: ((range_integral(typ'))), a: Address): valid?(uidt(dt_int))(to_byte(dt_int)(d, a), a)</pre>

Repeatedly Skolemizing and flattening,
 Instantiating quantified variables,
 Expanding the definition of `range_integral`,
 which is trivially true.

This completes the proof of `dt_integral_TCC5.1.1`.

`dt_integral_TCC5.1.2`:

{-1}	<code>uninterpreted_data_type?(dt_int ' uidt)</code>
{-2}	$\forall (d: ((\text{range_int})), a: \text{Address}):$ <code>valid?(uidt(dt_int))(to_byte(dt_int)(d, a), a)</code>
{-3}	$\forall (l: \text{list}[\text{Byte}], a: \text{Address}):$ <code>valid?(uidt(dt_int))(l, a) \equiv up?(from_byte(dt_int)(l, a))</code>
{-4}	$\forall (d: ((\text{range_int})), a: \text{Address}):$ <code>down(from_byte(dt_int)(to_byte(dt_int)(d, a), a)) = d</code>
{-5}	$\forall (l: \text{list}[\text{Byte}], a_1, a_2: \text{Address}):$ <code>valid?(uidt(dt_int))(l, a_1) \equiv valid?(uidt(dt_int))(l, a_2)</code>
{-6}	<code>size(uidt(dt_int)) > 0</code>
{-7}	<code>non_bool_integral?(int)</code>
{-8}	<code>int?(typ')</code>
{1}	$\forall (d: ((\text{range_integral}(\text{typ}'))), a: \text{Address}):$ <code>down(from_byte(dt_int)(to_byte(dt_int)(d, a), a)) = d</code>

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Repeatedly Skolemizing and flattening,

Instantiating quantified variables,

Expanding the definition of `range_integral`,

which is trivially true.

This completes the proof of `dt_integral_TCC5.1.2`.

`dt_integral_TCC5.2`:

{-1}	<code>non_bool_integral?(int)</code>
{-2}	<code>int?(typ')</code>
{1}	$\forall (x_1: [\text{list}[\text{Byte}], \text{Address}]):$ <code>every[[range_int]]((range_integral(int)))(dt_int ' from_byte(x_1))</code>

Repeatedly Skolemizing and flattening,

Expanding the definition of `every`,

Expanding the definition of `range_integral`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `dt_integral_TCC5.2`.

Q.E.D.

C.43.13 Cpp_Integral_Types.dt_integral_TCC6

Terse proof for `dt_integral_TCC6`.

`dt_integral_TCC6`:

{1}	$\forall (\text{typ}: (\text{non_bool_integral?})):$ <code>typ = long \supset</code> <code>($\forall (x: \text{int}): (\text{range_long})(x) \equiv \text{range_integral}(\text{typ})(x)) \wedge$</code> <code>($\forall (x: \text{int}): (\text{range_long})(x) \equiv \text{range_integral}(\text{typ})(x)) \wedge$</code> <code>($\forall (x_1: [\text{list}[\text{Byte}], \text{Address}]):$</code> <code> <code>every[[range_long]]((range_integral(typ)))(dt_long ' from_byte(x_1))</code></code> <code> \wedge pod_data_type?[[range_integral(typ)]](dt_long)</code>
-----	--

Repeatedly Skolemizing and flattening,

Replacing using formula -2,

Expanding the definition of `range_integral`,

C Proof scripts

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
we get 2 subgoals:

dt_integral_TCC6.1:

{-1}	non_bool_integral?(long)
{-2}	long?(typ')
{1}	pod_data_type?[((range_integral(typ')))](dt_long)

Adding type constraints for dt_long,
Expanding the definition of pod_data_type?,
Applying propositional simplification,
Expanding the definition of interpreted_data_type?,
Applying propositional simplification,
we get 2 subgoals:

dt_integral_TCC6.1.1:

{-1}	uninterpreted_data_type?(dt_long 'uidt)
{-2}	$\forall (d: ((range_long)), a: Address):$ valid?(uidt(dt_long))(to_byte(dt_long)(d, a), a)
{-3}	$\forall (l: list[Byte], a: Address):$ valid?(uidt(dt_long))(l, a) \equiv up?(from_byte(dt_long)(l, a))
{-4}	$\forall (d: ((range_long)), a: Address):$ down(from_byte(dt_long)(to_byte(dt_long)(d, a), a)) = d
{-5}	$\forall (l: list[Byte], a_1, a_2: Address):$ valid?(uidt(dt_long))(l, a_1) \equiv valid?(uidt(dt_long))(l, a_2)
{-6}	size(uidt(dt_long)) > 0
{-7}	non_bool_integral?(long)
{-8}	long?(typ')
{1}	$\forall (d: ((range_integral(typ'))), a: Address):$ valid?(uidt(dt_long))(to_byte(dt_long)(d, a), a)

Repeatedly Skolemizing and flattening,
Instantiating quantified variables,
Expanding the definition of range_integral,
which is trivially true.
This completes the proof of dt_integral_TCC6.1.1.

dt_integral_TCC6.1.2:

{-1}	uninterpreted_data_type?(dt_long 'uidt)
{-2}	$\forall (d: ((range_long)), a: Address):$ valid?(uidt(dt_long))(to_byte(dt_long)(d, a), a)
{-3}	$\forall (l: list[Byte], a: Address):$ valid?(uidt(dt_long))(l, a) \equiv up?(from_byte(dt_long)(l, a))
{-4}	$\forall (d: ((range_long)), a: Address):$ down(from_byte(dt_long)(to_byte(dt_long)(d, a), a)) = d
{-5}	$\forall (l: list[Byte], a_1, a_2: Address):$ valid?(uidt(dt_long))(l, a_1) \equiv valid?(uidt(dt_long))(l, a_2)
{-6}	size(uidt(dt_long)) > 0
{-7}	non_bool_integral?(long)
{-8}	long?(typ')
{1}	$\forall (d: ((range_integral(typ'))), a: Address):$ down(from_byte(dt_long)(to_byte(dt_long)(d, a), a)) = d

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Repeatedly Skolemizing and flattening,
Instantiating quantified variables,

Expanding the definition of `range_integral`,
which is trivially true.

This completes the proof of `dt_integral_TCC6.1.2`.

`dt_integral_TCC6.2`:

$\{-1\}$	<code>non_bool_integral?(long)</code>
$\{-2\}$	<code>long?(typ')</code>
$\{1\}$	$\forall (x_1: [\text{list}[\text{Byte}], \text{Address}]):$ $\text{every}[(\text{range_long})((\text{range_integral}(\text{long})))(\text{dt_long}' \text{ from_byte}(x_1))]$

Repeatedly Skolemizing and flattening,

Expanding the definition of `every`,

Expanding the definition of `range_integral`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `dt_integral_TCC6.2`.

Q.E.D.

C.43.14 Cpp_Integral_Types.dt_integral_TCC7

Terse proof for `dt_integral_TCC7`.

`dt_integral_TCC7`:

$\{1\}$	$\forall (\text{typ}: (\text{non_bool_integral?})):$ $\text{typ} = \text{longlong} \supset$ $(\forall (x: \text{int}): (\text{range_longlong})(x) \equiv \text{range_integral}(\text{typ})(x)) \wedge$ $(\forall (x: \text{int}): (\text{range_longlong})(x) \equiv \text{range_integral}(\text{typ})(x)) \wedge$ $(\forall (x_1: [\text{list}[\text{Byte}], \text{Address}]):$ $\text{every}[(\text{range_longlong})((\text{range_integral}(\text{typ})))(\text{dt_longlong}' \text{ from_byte}(x_1)))]$ $\wedge \text{pod_data_type?}[(\text{range_integral}(\text{typ}))](\text{dt_longlong})$
---------	---

Repeatedly Skolemizing and flattening,

Replacing using formula -2,

Expanding the definition of `range_integral`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 2 subgoals:

`dt_integral_TCC7.1`:

$\{-1\}$	<code>non_bool_integral?(longlong)</code>
$\{-2\}$	<code>longlong?(typ')</code>
$\{1\}$	$\text{pod_data_type?}[(\text{range_integral}(\text{typ}'))](\text{dt_longlong})$

Adding type constraints for `dt_longlong`,

Expanding the definition of `pod_data_type?`,

Applying propositional simplification,

Expanding the definition of `interpreted_data_type?`,

Applying propositional simplification,

we get 2 subgoals:

dt_integral_TCC7.1.1.1:

{-1}	uninterpreted_data_type?(dt_longlong 'uidt)
{-2}	$\forall (d: ((\text{range_longlong})), a: \text{Address}):$ valid?(uidt(dt_longlong))(to_byte(dt_longlong)(d, a), a)
{-3}	$\forall (l: \text{list}[\text{Byte}], a: \text{Address}):$ valid?(uidt(dt_longlong))(l, a) \equiv up?(from_byte(dt_longlong)(l, a))
{-4}	$\forall (d: ((\text{range_longlong})), a: \text{Address}):$ down(from_byte(dt_longlong)(to_byte(dt_longlong)(d, a), a)) = d
{-5}	$\forall (l: \text{list}[\text{Byte}], a_1, a_2: \text{Address}):$ valid?(uidt(dt_longlong))(l, a_1) \equiv valid?(uidt(dt_longlong))(l, a_2)
{-6}	size(uidt(dt_longlong)) > 0
{-7}	non_bool_integral?(longlong)
{-8}	longlong?(typ')
{1}	$\forall (d: ((\text{range_integral}(\text{typ}')), a: \text{Address}):$ valid?(uidt(dt_longlong))(to_byte(dt_longlong)(d, a), a)

Repeatedly Skolemizing and flattening,
Instantiating quantified variables,
Expanding the definition of range_integral,
which is trivially true.

This completes the proof of dt_integral_TCC7.1.1.

dt_integral_TCC7.1.2:

{-1}	uninterpreted_data_type?(dt_longlong 'uidt)
{-2}	$\forall (d: ((\text{range_longlong})), a: \text{Address}):$ valid?(uidt(dt_longlong))(to_byte(dt_longlong)(d, a), a)
{-3}	$\forall (l: \text{list}[\text{Byte}], a: \text{Address}):$ valid?(uidt(dt_longlong))(l, a) \equiv up?(from_byte(dt_longlong)(l, a))
{-4}	$\forall (d: ((\text{range_longlong})), a: \text{Address}):$ down(from_byte(dt_longlong)(to_byte(dt_longlong)(d, a), a)) = d
{-5}	$\forall (l: \text{list}[\text{Byte}], a_1, a_2: \text{Address}):$ valid?(uidt(dt_longlong))(l, a_1) \equiv valid?(uidt(dt_longlong))(l, a_2)
{-6}	size(uidt(dt_longlong)) > 0
{-7}	non_bool_integral?(longlong)
{-8}	longlong?(typ')
{1}	$\forall (d: ((\text{range_integral}(\text{typ}')), a: \text{Address}):$ down(from_byte(dt_longlong)(to_byte(dt_longlong)(d, a), a)) = d

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Repeatedly Skolemizing and flattening,
Instantiating quantified variables,
Expanding the definition of range_integral,
which is trivially true.

This completes the proof of dt_integral_TCC7.1.2.

dt_integral_TCC7.2:

{-1}	non_bool_integral?(longlong)
{-2}	longlong?(typ')
{1}	$\forall (x_1: [\text{list}[\text{Byte}], \text{Address}]):$ every[(range_longlong)]((range_integral(longlong))(dt_longlong 'from_byte(x_1))

Repeatedly Skolemizing and flattening,
Expanding the definition of every,
Expanding the definition of range_integral,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of dt_integral_TCC7.2.
Q.E.D.

C.43.15 Cpp_Integral_Types.dt_integral_TCC8

Terse proof for dt_integral_TCC8.

dt_integral_TCC8:

<pre>{1} ∀ (typ: (non_bool_integral?): typ = ushort ⊃ (∀ (x: int): x ≥ 0 ∧ (range_ushort)(x) ≡ range_integral(typ)(x)) ∧ (∀ (x: int): x ≥ 0 ∧ (range_ushort)(x) ≡ range_integral(typ)(x)) ∧ (∀ (x₁: [list[Byte], Address]): every[(range_ushort)]((range_integral(typ))(dt_ushort 'from_byte(x₁))) ∧ pod_data_type?[(range_integral(typ))(dt_ushort)</pre>

Repeatedly Skolemizing and flattening,
Replacing using formula -2,
Expanding the definition of range_integral,
Expanding the definition of extend,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
we get 3 subgoals:

dt_integral_TCC8.1:

<pre>{-1} non_bool_integral?(ushort) {-2} ushort?(typ')</pre>
<pre>{1} pod_data_type?[(range_integral(typ'))](dt_ushort)</pre>

Adding type constraints for dt_ushort,
Expanding the definition of pod_data_type?,
Applying propositional simplification,
Expanding the definition of interpreted_data_type?,
Applying propositional simplification,
we get 2 subgoals:

dt_integral_TCC8.1.1:

<pre>{-1} uninterpreted_data_type?(dt_ushort 'uidt) {-2} ∀ (d: ((range_ushort)), a: Address): valid?(uidt(dt_ushort))(to_byte(dt_ushort)(d, a), a) {-3} ∀ (l: list[Byte], a: Address): valid?(uidt(dt_ushort))(l, a) ≡ up?(from_byte(dt_ushort)(l, a)) {-4} ∀ (d: ((range_ushort)), a: Address): down(from_byte(dt_ushort)(to_byte(dt_ushort)(d, a), a)) = d {-5} ∀ (l: list[Byte], a₁, a₂: Address): valid?(uidt(dt_ushort))(l, a₁) ≡ valid?(uidt(dt_ushort))(l, a₂) {-6} size(uidt(dt_ushort)) > 0 {-7} non_bool_integral?(ushort) {-8} ushort?(typ')</pre>
<pre>{1} ∀ (d: ((range_integral(typ'))), a: Address): valid?(uidt(dt_ushort))(to_byte(dt_ushort)(d, a), a)</pre>

Repeatedly Skolemizing and flattening,
Instantiating quantified variables,
Expanding the definition of range_integral,

C Proof scripts

Expanding the definition of `extend`,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 This completes the proof of `dt_integral_TCC8.1.1`.
`dt_integral_TCC8.1.2`:

{-1}	<code>uninterpreted_data_type?(dt_ushort 'uidt)</code>
{-2}	$\forall (d: ((\text{range_ushort})), a: \text{Address}):$ <code>valid?(uidt(dt_ushort))(to_byte(dt_ushort)(d, a), a)</code>
{-3}	$\forall (l: \text{list}[\text{Byte}], a: \text{Address}):$ <code>valid?(uidt(dt_ushort))(l, a) \equiv up?(from_byte(dt_ushort)(l, a))</code>
{-4}	$\forall (d: ((\text{range_ushort})), a: \text{Address}):$ <code>down(from_byte(dt_ushort)(to_byte(dt_ushort)(d, a), a)) = d</code>
{-5}	$\forall (l: \text{list}[\text{Byte}], a_1, a_2: \text{Address}):$ <code>valid?(uidt(dt_ushort))(l, a_1) \equiv valid?(uidt(dt_ushort))(l, a_2)</code>
{-6}	<code>size(uidt(dt_ushort)) > 0</code>
{-7}	<code>non_bool_integral?(ushort)</code>
{-8}	<code>ushort?(typ')</code>
{1}	$\forall (d: ((\text{range_integral}(\text{typ}')), a: \text{Address}):$ <code>down(from_byte(dt_ushort)(to_byte(dt_ushort)(d, a), a)) = d</code>

Repeatedly Skolemizing and flattening,
 Instantiating quantified variables,
 Expanding the definition of `range_integral`,
 Expanding the definition of `extend`,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 This completes the proof of `dt_integral_TCC8.1.2`.
`dt_integral_TCC8.2`:

{-1}	<code>non_bool_integral?(ushort)</code>
{-2}	<code>ushort?(typ')</code>
{1}	$\forall (x_1: [\text{list}[\text{Byte}], \text{Address}]):$ <code>every[(range_ushort)]((range_integral(ushort))(dt_ushort 'from_byte(x_1))</code>

Repeatedly Skolemizing and flattening,
 Expanding the definition of `every`,
 Expanding the definition of `range_integral`,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 Expanding the definition of `extend`,
 which is trivially true.
 This completes the proof of `dt_integral_TCC8.2`.
`dt_integral_TCC8.3`:

{-1}	<code>non_bool_integral?(ushort)</code>
{-2}	<code>ushort?(typ')</code>
{1}	$\forall (x: \text{int}):$ <code>$x \geq 0 \wedge (\text{range_ushort})(x) \equiv \text{IF } x \geq 0 \text{ THEN range_ushort}(x) \text{ ELSE FALSE ENDIF}$</code>

Repeatedly Skolemizing and flattening,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 This completes the proof of `dt_integral_TCC8.3`.
 Q.E.D.

C.43.16 Cpp_Integral_Types.dt_integral_TCC9

Terse proof for `dt_integral_TCC9`.

dt_integral_TCC9:

$\{1\} \quad \forall (\text{typ}: (\text{non_bool_integral?})): \\ \text{typ} = \text{uint} \supset \\ (\forall (x: \text{int}): x \geq 0 \wedge (\text{range_uint})(x) \equiv \text{range_integral}(\text{typ})(x)) \wedge \\ (\forall (x: \text{int}): x \geq 0 \wedge (\text{range_uint})(x) \equiv \text{range_integral}(\text{typ})(x)) \wedge \\ (\forall (x_1: [\text{list}[\text{Byte}], \text{Address}]): \\ \text{every}[(\text{range_uint})](\text{range_integral}(\text{typ}))(\text{dt_uint}'\text{from_byte}(x_1))) \\ \wedge \text{pod_data_type?}[(\text{range_integral}(\text{typ}))](\text{dt_uint})$

Repeatedly Skolemizing and flattening,

Replacing using formula -2,

Expanding the definition of range_integral,

Expanding the definition of extend,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 3 subgoals:

dt_integral_TCC9.1:

$\{-1\} \quad \text{non_bool_integral?}(\text{uint})$
$\{-2\} \quad \text{uint?}(\text{typ}')$
$\{1\} \quad \text{pod_data_type?}[(\text{range_integral}(\text{typ}'))](\text{dt_uint})$

Adding type constraints for dt_uint,

Expanding the definition of pod_data_type?,

Applying propositional simplification,

Expanding the definition of interpreted_data_type?,

Applying propositional simplification,

we get 2 subgoals:

dt_integral_TCC9.1.1:

$\{-1\} \quad \text{uninterpreted_data_type?}(\text{dt_uint}'\text{uidt})$
$\{-2\} \quad \forall (d: ((\text{range_uint})), a: \text{Address}): \\ \text{valid?}(\text{uidt}(\text{dt_uint}))(\text{to_byte}(\text{dt_uint})(d, a), a)$
$\{-3\} \quad \forall (l: \text{list}[\text{Byte}], a: \text{Address}): \\ \text{valid?}(\text{uidt}(\text{dt_uint}))(l, a) \equiv \text{up?}(\text{from_byte}(\text{dt_uint})(l, a))$
$\{-4\} \quad \forall (d: ((\text{range_uint})), a: \text{Address}): \\ \text{down}(\text{from_byte}(\text{dt_uint}))(\text{to_byte}(\text{dt_uint})(d, a), a) = d$
$\{-5\} \quad \forall (l: \text{list}[\text{Byte}], a_1, a_2: \text{Address}): \\ \text{valid?}(\text{uidt}(\text{dt_uint}))(l, a_1) \equiv \text{valid?}(\text{uidt}(\text{dt_uint}))(l, a_2)$
$\{-6\} \quad \text{size}(\text{uidt}(\text{dt_uint})) > 0$
$\{-7\} \quad \text{non_bool_integral?}(\text{uint})$
$\{-8\} \quad \text{uint?}(\text{typ}')$
$\{1\} \quad \forall (d: ((\text{range_integral}(\text{typ}'))), a: \text{Address}): \\ \text{valid?}(\text{uidt}(\text{dt_uint}))(\text{to_byte}(\text{dt_uint})(d, a), a)$

Repeatedly Skolemizing and flattening,

Instantiating quantified variables,

Expanding the definition of range_integral,

Expanding the definition of extend,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of dt_integral_TCC9.1.1.

dt_integral_TCC9.1.2:

{-1}	uninterpreted_data_type?(dt_uint 'uidt)
{-2}	$\forall (d: ((\text{range_uint})), a: \text{Address}):$ valid?(uidt(dt_uint))(to_byte(dt_uint)(d, a), a)
{-3}	$\forall (l: \text{list}[\text{Byte}], a: \text{Address}):$ valid?(uidt(dt_uint))(l, a) \equiv up?(from_byte(dt_uint)(l, a))
{-4}	$\forall (d: ((\text{range_uint})), a: \text{Address}):$ down(from_byte(dt_uint)(to_byte(dt_uint)(d, a), a)) = d
{-5}	$\forall (l: \text{list}[\text{Byte}], a_1, a_2: \text{Address}):$ valid?(uidt(dt_uint))(l, a_1) \equiv valid?(uidt(dt_uint))(l, a_2)
{-6}	size(uidt(dt_uint)) > 0
{-7}	non_bool_integral?(uint)
{-8}	uint?(typ')
{1}	
	$\forall (d: ((\text{range_integral}(\text{typ}')), a: \text{Address}):$ down(from_byte(dt_uint)(to_byte(dt_uint)(d, a), a)) = d

Repeatedly Skolemizing and flattening,
Instantiating quantified variables,
Expanding the definition of range_integral,
Expanding the definition of extend,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of dt_integral_TCC9.1.2.

dt_integral_TCC9.2:

{-1}	non_bool_integral?(uint)
{-2}	uint?(typ')
{1}	
	$\forall (x_1: [\text{list}[\text{Byte}], \text{Address}]):$ every[(range_uint)]((range_integral(uint))(dt_uint 'from_byte(x_1)))

Repeatedly Skolemizing and flattening,
Expanding the definition of every,
Expanding the definition of range_integral,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Expanding the definition of extend,
which is trivially true.
This completes the proof of dt_integral_TCC9.2.

dt_integral_TCC9.3:

{-1}	non_bool_integral?(uint)
{-2}	uint?(typ')
{1}	
	$\forall (x: \text{int}):$ $x \geq 0 \wedge (\text{range_uint})(x) \equiv \text{IF } x \geq 0 \text{ THEN range_uint}(x) \text{ ELSE FALSE ENDIF}$

Repeatedly Skolemizing and flattening,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of dt_integral_TCC9.3.
Q.E.D.

C.43.17 Cpp_Integral_Types.dt_integral_TCC10

Terse proof for dt_integral_TCC10.

dt_integral_TCC10:

$\{1\} \quad \forall (\text{typ}: (\text{non_bool_integral?})): \\ \text{typ} = \text{ulong} \supset \\ (\forall (x: \text{int}): x \geq 0 \wedge (\text{range_ulong})(x) \equiv \text{range_integral}(\text{typ})(x)) \wedge \\ (\forall (x: \text{int}): x \geq 0 \wedge (\text{range_ulong})(x) \equiv \text{range_integral}(\text{typ})(x)) \wedge \\ (\forall (x_1: [\text{list}[\text{Byte}], \text{Address}]): \\ \text{every}[(\text{range_ulong})](\text{range_integral}(\text{typ}))(\text{dt_ulong } \text{'from_byte}(x_1))) \\ \wedge \text{pod_data_type?}[(\text{range_integral}(\text{typ}))](\text{dt_ulong})$

Repeatedly Skolemizing and flattening,

Replacing using formula -2,

Expanding the definition of range_integral,

Expanding the definition of extend,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 3 subgoals:

dt_integral_TCC10.1:

$\{-1\} \quad \text{non_bool_integral?}(\text{ulong})$	$\{-2\} \quad \text{ulong?}(\text{typ}')$
$\{1\} \quad \text{pod_data_type?}[(\text{range_integral}(\text{typ}'))](\text{dt_ulong})$	

Adding type constraints for dt_ulong,

Expanding the definition of pod_data_type?,

Applying propositional simplification,

Expanding the definition of interpreted_data_type?,

Applying propositional simplification,

we get 2 subgoals:

dt_integral_TCC10.1.1:

$\{-1\} \quad \text{uninterpreted_data_type?}(\text{dt_ulong } \text{'uidt})$	$\{-2\} \quad \forall (d: ((\text{range_ulong})), a: \text{Address}): \\ \text{valid?}(\text{uidt}(\text{dt_ulong}))(\text{to_byte}(\text{dt_ulong})(d, a), a)$
$\{-3\} \quad \forall (l: \text{list}[\text{Byte}], a: \text{Address}): \\ \text{valid?}(\text{uidt}(\text{dt_ulong}))(l, a) \equiv \text{up?}(\text{from_byte}(\text{dt_ulong})(l, a))$	$\{-4\} \quad \forall (d: ((\text{range_ulong})), a: \text{Address}): \\ \text{down}(\text{from_byte}(\text{dt_ulong})(\text{to_byte}(\text{dt_ulong})(d, a), a)) = d$
$\{-5\} \quad \forall (l: \text{list}[\text{Byte}], a_1, a_2: \text{Address}): \\ \text{valid?}(\text{uidt}(\text{dt_ulong}))(l, a_1) \equiv \text{valid?}(\text{uidt}(\text{dt_ulong}))(l, a_2)$	$\{-6\} \quad \text{size}(\text{uidt}(\text{dt_ulong})) > 0$
$\{-7\} \quad \text{non_bool_integral?}(\text{ulong})$	$\{-8\} \quad \text{ulong?}(\text{typ}')$
$\{1\} \quad \forall (d: ((\text{range_integral}(\text{typ}'))), a: \text{Address}): \\ \text{valid?}(\text{uidt}(\text{dt_ulong}))(\text{to_byte}(\text{dt_ulong})(d, a), a)$	

Repeatedly Skolemizing and flattening,

Instantiating quantified variables,

Expanding the definition of range_integral,

Expanding the definition of extend,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of dt_integral_TCC10.1.1.

dt_integral_TCC10.1.2:

{-1}	uninterpreted_data_type?(dt_ulong 'uidt)
{-2}	$\forall (d: ((\text{range_ulong})), a: \text{Address}):$ valid?(uidt(dt_ulong))(to_byte(dt_ulong)(d, a), a)
{-3}	$\forall (l: \text{list}[\text{Byte}], a: \text{Address}):$ valid?(uidt(dt_ulong))(l, a) \equiv up?(from_byte(dt_ulong)(l, a))
{-4}	$\forall (d: ((\text{range_ulong})), a: \text{Address}):$ down(from_byte(dt_ulong)(to_byte(dt_ulong)(d, a), a)) = d
{-5}	$\forall (l: \text{list}[\text{Byte}], a_1, a_2: \text{Address}):$ valid?(uidt(dt_ulong))(l, a_1) \equiv valid?(uidt(dt_ulong))(l, a_2)
{-6}	size(uidt(dt_ulong)) > 0
{-7}	non_bool_integral?(ulong)
{-8}	ulong?(typ')
{1} $\forall (d: ((\text{range_integral}(\text{typ}')), a: \text{Address}):$ down(from_byte(dt_ulong)(to_byte(dt_ulong)(d, a), a)) = d	

Repeatedly Skolemizing and flattening,
Instantiating quantified variables,
Expanding the definition of range_integral,
Expanding the definition of extend,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of dt_integral_TCC10.1.2.

dt_integral_TCC10.2:

{-1}	non_bool_integral?(ulong)
{-2}	ulong?(typ')
{1} $\forall (x_1: [\text{list}[\text{Byte}], \text{Address}]):$ every[(range_ulong)]((range_integral(ulong))(dt_ulong 'from_byte(x_1)))	

Repeatedly Skolemizing and flattening,
Expanding the definition of every,
Expanding the definition of range_integral,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Expanding the definition of extend,
which is trivially true.
This completes the proof of dt_integral_TCC10.2.

dt_integral_TCC10.3:

{-1}	non_bool_integral?(ulong)
{-2}	ulong?(typ')
{1} $\forall (x: \text{int}):$ $x \geq 0 \wedge (\text{range_ulong})(x) \equiv \text{IF } x \geq 0 \text{ THEN range_ulong}(x) \text{ ELSE FALSE ENDIF}$	

Repeatedly Skolemizing and flattening,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of dt_integral_TCC10.3.
Q.E.D.

C.43.18 Cpp_Integral_Types.dt_integral_TCC11

Terse proof for dt_integral_TCC11.

dt_integral_TCC11:

<pre> {1} ∀ (typ: (non_bool_integral?): typ = ulonglong ⊃ (∀ (x: int): x ≥ 0 ∧ (range_ulonglong)(x) ≡ range_integral(typ)(x)) ∧ (∀ (x: int): x ≥ 0 ∧ (range_ulonglong)(x) ≡ range_integral(typ)(x)) ∧ (∀ (x₁: [list[Byte], Address]): every[(range_ulonglong)] ((range_integral(typ))(dt_ulonglong'from_byte(x₁))) ∧ pod_data_type?[(range_integral(typ))](dt_ulonglong) </pre>
--

Repeatedly Skolemizing and flattening,

Replacing using formula -2,

Expanding the definition of range_integral,

Expanding the definition of extend,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
we get 3 subgoals:

dt_integral_TCC11.1:

<pre> {-1} non_bool_integral?(ulonglong) {-2} ulonglong?(typ') </pre>	<pre> {1} pod_data_type?[(range_integral(typ'))](dt_ulonglong) </pre>
---	--

Adding type constraints for dt_ulonglong,

Expanding the definition of pod_data_type?,

Applying propositional simplification,

Expanding the definition of interpreted_data_type?,

Applying propositional simplification,

we get 2 subgoals:

dt_integral_TCC11.1.1:

<pre> {-1} uninterpreted_data_type?(dt_ulonglong'uidt) {-2} ∀ (d: ((range_ulonglong)), a: Address): valid?(uidt(dt_ulonglong))(to_byte(dt_ulonglong)(d, a), a) {-3} ∀ (l: list[Byte], a: Address): valid?(uidt(dt_ulonglong))(l, a) ≡ up?(from_byte(dt_ulonglong)(l, a)) {-4} ∀ (d: ((range_ulonglong)), a: Address): down(from_byte(dt_ulonglong)(to_byte(dt_ulonglong)(d, a), a)) = d {-5} ∀ (l: list[Byte], a₁, a₂: Address): valid?(uidt(dt_ulonglong))(l, a₁) ≡ valid?(uidt(dt_ulonglong))(l, a₂) {-6} size(uidt(dt_ulonglong)) > 0 {-7} non_bool_integral?(ulonglong) {-8} ulonglong?(typ') </pre>	<pre> {1} ∀ (d: ((range_integral(typ'))), a: Address): valid?(uidt(dt_ulonglong))(to_byte(dt_ulonglong)(d, a), a) </pre>
---	---

Repeatedly Skolemizing and flattening,

Instantiating quantified variables,

Expanding the definition of range_integral,

Expanding the definition of extend,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of dt_integral_TCC11.1.1.

dt_integral_TCC11.1.2:

{-1}	uninterpreted_data_type?(dt_ulonglong'uidt)
{-2}	$\forall (d: ((\text{range_ulonglong})), a: \text{Address}):$ valid?(uidt(dt_ulonglong))(to_byte(dt_ulonglong)(d, a), a)
{-3}	$\forall (l: \text{list}[\text{Byte}], a: \text{Address}):$ valid?(uidt(dt_ulonglong))(l, a) \equiv up?(from_byte(dt_ulonglong)(l, a))
{-4}	$\forall (d: ((\text{range_ulonglong})), a: \text{Address}):$ down(from_byte(dt_ulonglong)(to_byte(dt_ulonglong)(d, a), a)) = d
{-5}	$\forall (l: \text{list}[\text{Byte}], a_1, a_2: \text{Address}):$ valid?(uidt(dt_ulonglong))(l, a_1) \equiv valid?(uidt(dt_ulonglong))(l, a_2)
{-6}	size(uidt(dt_ulonglong)) > 0
{-7}	non_bool_integral?(ulonglong)
{-8}	ulonglong?(typ')
{1}	$\forall (d: ((\text{range_integral}(\text{typ}')), a: \text{Address}):$ down(from_byte(dt_ulonglong)(to_byte(dt_ulonglong)(d, a), a)) = d

Repeatedly Skolemizing and flattening,
Instantiating quantified variables,
Expanding the definition of range_integral,
Expanding the definition of extend,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of dt_integral_TCC11.1.2.

dt_integral_TCC11.2:

{-1}	non_bool_integral?(ulonglong)
{-2}	ulonglong?(typ')
{1}	$\forall (x_1: [\text{list}[\text{Byte}], \text{Address}]):$ every[(range_ulonglong)]((range_integral(ulonglong))(dt_ulonglong'from_byte(x_1))

Repeatedly Skolemizing and flattening,
Expanding the definition of every,
Expanding the definition of range_integral,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Expanding the definition of extend,
which is trivially true.
This completes the proof of dt_integral_TCC11.2.

dt_integral_TCC11.3:

{-1}	non_bool_integral?(ulonglong)
{-2}	ulonglong?(typ')
{1}	$\forall (x: \text{int}):$ $x \geq 0 \wedge (\text{range_ulonglong})(x) \equiv \text{IF } x \geq 0 \text{ THEN range_ulonglong}(x) \text{ ELSE FALSE EN-}$ DIF

Repeatedly Skolemizing and flattening,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of dt_integral_TCC11.3.
Q.E.D.

C.43.19 Cpp_Integral_Types.dt_integral_TCC12

Terse proof for dt_integral_TCC12.

dt_integral_TCC12:

$\{1\} \quad \forall (\text{typ}: (\text{non_bool_integral?})): \\ \text{typ} = \text{wchar_t} \supset \\ (\forall (x: \text{int}): (\text{range_wchar_t})(x) \equiv \text{range_integral}(\text{typ})(x)) \wedge \\ (\forall (x: \text{int}): (\text{range_wchar_t})(x) \equiv \text{range_integral}(\text{typ})(x)) \wedge \\ (\forall (x_1: [\text{list}[\text{Byte}], \text{Address}]): \\ \text{every}[(\text{range_wchar_t})((\text{range_integral}(\text{typ})))(\text{dt_wchar_t}'\text{from_byte}(x_1))] \\ \wedge \text{pod_data_type?}[(\text{range_integral}(\text{typ}))](\text{dt_wchar_t}))$

Repeatedly Skolemizing and flattening,

Replacing using formula -2,

Expanding the definition of range_integral,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 2 subgoals:

dt_integral_TCC12.1:

$\{-1\} \quad \text{non_bool_integral?}(\text{wchar_t})$
$\{-2\} \quad \text{wchar_t?}(\text{typ}')$
$\{1\} \quad \text{pod_data_type?}[(\text{range_integral}(\text{typ}'))](\text{dt_wchar_t})$

Adding type constraints for dt_wchar_t,

Expanding the definition of pod_data_type?,

Applying propositional simplification,

Expanding the definition of interpreted_data_type?,

Applying propositional simplification,

we get 2 subgoals:

dt_integral_TCC12.1.1:

$\{-1\} \quad \text{uninterpreted_data_type?}(\text{dt_wchar_t}'\text{uidt})$
$\{-2\} \quad \forall (d: ((\text{range_wchar_t})), a: \text{Address}): \\ \text{valid?}(\text{uidt}(\text{dt_wchar_t}))(\text{to_byte}(\text{dt_wchar_t})(d, a), a)$
$\{-3\} \quad \forall (l: \text{list}[\text{Byte}], a: \text{Address}): \\ \text{valid?}(\text{uidt}(\text{dt_wchar_t}))(l, a) \equiv \text{up?}(\text{from_byte}(\text{dt_wchar_t})(l, a))$
$\{-4\} \quad \forall (d: ((\text{range_wchar_t})), a: \text{Address}): \\ \text{down}(\text{from_byte}(\text{dt_wchar_t})(\text{to_byte}(\text{dt_wchar_t})(d, a), a)) = d$
$\{-5\} \quad \forall (l: \text{list}[\text{Byte}], a_1, a_2: \text{Address}): \\ \text{valid?}(\text{uidt}(\text{dt_wchar_t}))(l, a_1) \equiv \text{valid?}(\text{uidt}(\text{dt_wchar_t}))(l, a_2)$
$\{-6\} \quad \text{size}(\text{uidt}(\text{dt_wchar_t})) > 0$
$\{-7\} \quad \text{non_bool_integral?}(\text{wchar_t})$
$\{-8\} \quad \text{wchar_t?}(\text{typ}')$
$\{1\} \quad \forall (d: ((\text{range_integral}(\text{typ}'))), a: \text{Address}): \\ \text{valid?}(\text{uidt}(\text{dt_wchar_t}))(\text{to_byte}(\text{dt_wchar_t})(d, a), a)$

Repeatedly Skolemizing and flattening,

Instantiating quantified variables,

Expanding the definition of range_integral,

which is trivially true.

This completes the proof of dt_integral_TCC12.1.1.

dt_integral_TCC12.1.2:

{-1}	uninterpreted_data_type?(dt_wchar_t 'uidt)
{-2}	$\forall (d: ((\text{range_wchar_t})), a: \text{Address}):$ valid?(uidt(dt_wchar_t))(to_byte(dt_wchar_t)(d, a), a)
{-3}	$\forall (l: \text{list}[\text{Byte}], a: \text{Address}):$ valid?(uidt(dt_wchar_t))(l, a) \equiv up?(from_byte(dt_wchar_t)(l, a))
{-4}	$\forall (d: ((\text{range_wchar_t})), a: \text{Address}):$ down(from_byte(dt_wchar_t)(to_byte(dt_wchar_t)(d, a), a)) = d
{-5}	$\forall (l: \text{list}[\text{Byte}], a_1, a_2: \text{Address}):$ valid?(uidt(dt_wchar_t))(l, a_1) \equiv valid?(uidt(dt_wchar_t))(l, a_2)
{-6}	size(uidt(dt_wchar_t)) > 0
{-7}	non_bool_integral?(wchar_t)
{-8}	wchar_t?(typ')
{1}	
	$\forall (d: ((\text{range_integral}(\text{typ}')), a: \text{Address}):$ down(from_byte(dt_wchar_t)(to_byte(dt_wchar_t)(d, a), a)) = d

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Repeatedly Skolemizing and flattening,
Instantiating quantified variables,
Expanding the definition of range_integral,
which is trivially true.

This completes the proof of dt_integral_TCC12.1.2.

dt_integral_TCC12.2:

{-1}	non_bool_integral?(wchar_t)
{-2}	wchar_t?(typ')
{1}	
	$\forall (x_1: [\text{list}[\text{Byte}], \text{Address}]):$ every[(range_wchar_t)]((range_integral(wchar_t))(dt_wchar_t 'from_byte(x_1)))

Repeatedly Skolemizing and flattening,
Expanding the definition of every,
Expanding the definition of range_integral,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of dt_integral_TCC12.2.
Q.E.D.

C.43.20 Cpp_Integral_Types.dt_floating_point_TCC1

Terse proof for dt_floating_point_TCC1.

dt_floating_point_TCC1:

{1}	
	$\forall (\text{typ}: (\text{floating_point?})):$ typ = float \supset ($\forall (x: \text{extended_real}): (\text{range_float})(x) \equiv \text{range_floating_point}(\text{typ})(x)$) \wedge ($\forall (x: \text{extended_real}): (\text{range_float})(x) \equiv \text{range_floating_point}(\text{typ})(x)$) \wedge ($\forall (x_1: [\text{list}[\text{Byte}], \text{Address}]):$ every[(range_float)]((range_floating_point(typ))(dt_float 'from_byte(x_1))) \wedge pod_data_type?[(range_floating_point(typ))](dt_float))

Repeatedly Skolemizing and flattening,
Replacing using formula -2,
Expanding the definition of range_floating_point,
Applying propositional simplification,

we get 2 subgoals:

dt_floating_point_TCC1.1:

{-1} floating_point?(float) {-2} typ' = float	
{1} $\forall (x_1: [\text{list}[\text{Byte}], \text{Address}]$): every[(range_float)]((range_floating_point(float)))(dt_float' from_byte(x ₁))	

Repeatedly Skolemizing and flattening,

Expanding the definition of every,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Expanding the definition of range_floating_point,

which is trivially true.

This completes the proof of dt_floating_point_TCC1.1.

dt_floating_point_TCC1.2:

{-1} floating_point?(float) {-2} typ' = float	
{1} pod_data_type?[(range_floating_point(typ'))](dt_float)	

Adding type constraints for dt_float,

Expanding the definition of pod_data_type?,

Applying propositional simplification,

Expanding the definition of interpreted_data_type?,

Applying propositional simplification,

we get 2 subgoals:

dt_floating_point_TCC1.2.1:

{-1} uninterpreted_data_type?(dt_float' uidt) {-2} $\forall (d: (\text{range_float}), a: \text{Address})$: valid?(uidt(dt_float))(to_byte(dt_float)(d, a), a) {-3} $\forall (l: \text{list}[\text{Byte}], a: \text{Address})$: valid?(uidt(dt_float))(l, a) \equiv up?(from_byte(dt_float)(l, a)) {-4} $\forall (d: (\text{range_float}), a: \text{Address})$: down(from_byte(dt_float)(to_byte(dt_float)(d, a), a)) = d {-5} $\forall (l: \text{list}[\text{Byte}], a_1, a_2: \text{Address})$: valid?(uidt(dt_float))(l, a ₁) \equiv valid?(uidt(dt_float))(l, a ₂) {-6} size(uidt(dt_float)) > 0 {-7} floating_point?(float) {-8} typ' = float	
{1} $\forall (d: (\text{range_floating_point}(\text{typ}')), a: \text{Address})$: valid?(uidt(dt_float))(to_byte(dt_float)(d, a), a)	

Repeatedly Skolemizing and flattening,

Instantiating quantified variables,

Expanding the definition of range_floating_point,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of dt_floating_point_TCC1.2.1.

dt_floating_point_TCC1.2.2:

{-1}	uninterpreted_data_type?(dt_float'uidt)
{-2}	$\forall (d: ((range_float)), a: Address):$ valid?(uidt(dt_float))(to_byte(dt_float)(d, a), a)
{-3}	$\forall (l: list[Byte], a: Address):$ valid?(uidt(dt_float))(l, a) \equiv up?(from_byte(dt_float)(l, a))
{-4}	$\forall (d: ((range_float)), a: Address):$ down(from_byte(dt_float)(to_byte(dt_float)(d, a), a)) = d
{-5}	$\forall (l: list[Byte], a_1, a_2: Address):$ valid?(uidt(dt_float))(l, a_1) \equiv valid?(uidt(dt_float))(l, a_2)
{-6}	size(uidt(dt_float)) > 0
{-7}	floating_point?(float)
{-8}	typ' = float
{1}	$\forall (d: ((range_floating_point(typ'))), a: Address):$ down(from_byte(dt_float)(to_byte(dt_float)(d, a), a)) = d

Repeatedly Skolemizing and flattening,
Instantiating quantified variables,
Expanding the definition of range_floating_point,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of dt_floating_point_TCC1.2.2.
Q.E.D.

C.43.21 Cpp_Integral_Types.dt_floating_point_TCC2

Terse proof for dt_floating_point_TCC2.

dt_floating_point_TCC2:

{1}	$\forall (typ: (floating_point?)):$ typ = double \supset $(\forall (x: extended_real): (range_double)(x) \equiv range_floating_point(typ)(x)) \wedge$ $(\forall (x: extended_real): (range_double)(x) \equiv range_floating_point(typ)(x)) \wedge$ $(\forall (x_1: [list[Byte], Address]):$ every[(range_double)] ((range_floating_point(typ))(dt_double'from_byte(x_1))) $\wedge pod_data_type?[(range_floating_point(typ))](dt_double)$
-----	--

Repeatedly Skolemizing and flattening,
Replacing using formula -2,
Expanding the definition of range_floating_point,
Applying propositional simplification,
we get 2 subgoals:

dt_floating_point_TCC2.1:

{-1}	floating_point?(double)
{-2}	typ' = double
{1}	$\forall (x_1: [list[Byte], Address]):$ every[(range_double)]((range_floating_point(double))(dt_double'from_byte(x_1)))

Repeatedly Skolemizing and flattening,
Expanding the definition of every,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Expanding the definition of range_floating_point,

which is trivially true.

This completes the proof of `dt_floating_point_TCC2.1`.

`dt_floating_point_TCC2.2`:

{-1}	<code>floating_point?(double)</code>
{-2}	<code>typ' = double</code>
{1}	<code>pod_data_type?(((range_floating_point(typ')))(dt_double)</code>

Adding type constraints for `dt_double`,

Expanding the definition of `pod_data_type?`,

Applying propositional simplification,

Expanding the definition of `interpreted_data_type?`,

Applying propositional simplification,

we get 2 subgoals:

`dt_floating_point_TCC2.2.1`:

{-1}	<code>uninterpreted_data_type?(dt_double uidt)</code>
{-2}	$\forall (d: ((range_double)), a: Address):$ <code>valid?(uidt(dt_double))(to_byte(dt_double)(d, a), a)</code>
{-3}	$\forall (l: list[Byte], a: Address):$ <code>valid?(uidt(dt_double))(l, a) \equiv up?(from_byte(dt_double)(l, a))</code>
{-4}	$\forall (d: ((range_double)), a: Address):$ <code>down(from_byte(dt_double)(to_byte(dt_double)(d, a), a)) = d</code>
{-5}	$\forall (l: list[Byte], a_1, a_2: Address):$ <code>valid?(uidt(dt_double))(l, a_1) \equiv valid?(uidt(dt_double))(l, a_2)</code>
{-6}	<code>size(uidt(dt_double)) > 0</code>
{-7}	<code>floating_point?(double)</code>
{-8}	<code>typ' = double</code>
{1}	$\forall (d: ((range_floating_point(typ'))), a: Address):$ <code>valid?(uidt(dt_double))(to_byte(dt_double)(d, a), a)</code>

Repeatedly Skolemizing and flattening,

Instantiating quantified variables,

Expanding the definition of `range_floating_point`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `dt_floating_point_TCC2.2.1`.

`dt_floating_point_TCC2.2.2`:

{-1}	<code>uninterpreted_data_type?(dt_double uidt)</code>
{-2}	$\forall (d: ((range_double)), a: Address):$ <code>valid?(uidt(dt_double))(to_byte(dt_double)(d, a), a)</code>
{-3}	$\forall (l: list[Byte], a: Address):$ <code>valid?(uidt(dt_double))(l, a) \equiv up?(from_byte(dt_double)(l, a))</code>
{-4}	$\forall (d: ((range_double)), a: Address):$ <code>down(from_byte(dt_double)(to_byte(dt_double)(d, a), a)) = d</code>
{-5}	$\forall (l: list[Byte], a_1, a_2: Address):$ <code>valid?(uidt(dt_double))(l, a_1) \equiv valid?(uidt(dt_double))(l, a_2)</code>
{-6}	<code>size(uidt(dt_double)) > 0</code>
{-7}	<code>floating_point?(double)</code>
{-8}	<code>typ' = double</code>
{1}	$\forall (d: ((range_floating_point(typ'))), a: Address):$ <code>down(from_byte(dt_double)(to_byte(dt_double)(d, a), a)) = d</code>

Repeatedly Skolemizing and flattening,

Instantiating quantified variables,

Expanding the definition of `range_floating_point`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `dt_floating_point_TCC2.2.2`.
Q.E.D.

C.43.22 Cpp_Integral_Types.dt_floating_point_TCC3

Terse proof for `dt_floating_point_TCC3`.

`dt_floating_point_TCC3`:

<pre>{1} ∀ (typ: (floating_point?): typ = longdouble ⊃ (∀ (x: extended_real): (range_longdouble)(x) ≡ range_floating_point(typ)(x)) ∧ (∀ (x: extended_real): (range_longdouble)(x) ≡ range_floating_point(typ)(x)) ∧ (∀ (x₁: [list[Byte], Address]): every[(range_longdouble)] ((range_floating_point(typ))(dt_longdouble'from_byte(x₁))) ∧ pod_data_type?[(range_floating_point(typ))](dt_longdouble))</pre>
--

Repeatedly Skolemizing and flattening,
Replacing using formula -2,
Expanding the definition of `range_floating_point`,
Applying propositional simplification,
we get 2 subgoals:

`dt_floating_point_TCC3.1`:

<pre>{-1} floating_point?(longdouble) {-2} typ' = longdouble</pre> <hr/> <pre>{1} ∀ (x₁: [list[Byte], Address]): every[(range_longdouble)] ((range_floating_point(longdouble))(dt_longdouble'from_byte(x₁)))</pre>

Repeatedly Skolemizing and flattening,
Expanding the definition of `every`,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Expanding the definition of `range_floating_point`,
which is trivially true.

This completes the proof of `dt_floating_point_TCC3.1`.

`dt_floating_point_TCC3.2`:

<pre>{-1} floating_point?(longdouble) {-2} typ' = longdouble</pre> <hr/> <pre>{1} pod_data_type?[(range_floating_point(typ'))](dt_longdouble)</pre>
--

Adding type constraints for `dt_longdouble`,
Expanding the definition of `pod_data_type?`,
Applying propositional simplification,
Expanding the definition of `interpreted_data_type?`,
Applying propositional simplification,
we get 2 subgoals:

dt_floating_point_TCC3.2.1:

{-1}	uninterpreted_data_type?(dt_longdouble ' uidt)
{-2}	$\forall (d: ((\text{range_longdouble})), a: \text{Address}):$ valid?(uidt(dt_longdouble))(to_byte(dt_longdouble)(d, a), a)
{-3}	$\forall (l: \text{list}[\text{Byte}], a: \text{Address}):$ valid?(uidt(dt_longdouble))(l, a) \equiv up?(from_byte(dt_longdouble)(l, a))
{-4}	$\forall (d: ((\text{range_longdouble})), a: \text{Address}):$ down(from_byte(dt_longdouble)(to_byte(dt_longdouble)(d, a), a)) = d
{-5}	$\forall (l: \text{list}[\text{Byte}], a_1, a_2: \text{Address}):$ valid?(uidt(dt_longdouble))(l, a_1) \equiv valid?(uidt(dt_longdouble))(l, a_2)
{-6}	size(uidt(dt_longdouble)) > 0
{-7}	floating_point?(longdouble)
{-8}	typ' = longdouble
{1}	$\forall (d: ((\text{range_floating_point}(\text{typ}')), a: \text{Address}):$ valid?(uidt(dt_longdouble))(to_byte(dt_longdouble)(d, a), a)

Repeatedly Skolemizing and flattening,

Instantiating quantified variables,

Expanding the definition of range_floating_point,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of dt_floating_point_TCC3.2.1.

dt_floating_point_TCC3.2.2:

{-1}	uninterpreted_data_type?(dt_longdouble ' uidt)
{-2}	$\forall (d: ((\text{range_longdouble})), a: \text{Address}):$ valid?(uidt(dt_longdouble))(to_byte(dt_longdouble)(d, a), a)
{-3}	$\forall (l: \text{list}[\text{Byte}], a: \text{Address}):$ valid?(uidt(dt_longdouble))(l, a) \equiv up?(from_byte(dt_longdouble)(l, a))
{-4}	$\forall (d: ((\text{range_longdouble})), a: \text{Address}):$ down(from_byte(dt_longdouble)(to_byte(dt_longdouble)(d, a), a)) = d
{-5}	$\forall (l: \text{list}[\text{Byte}], a_1, a_2: \text{Address}):$ valid?(uidt(dt_longdouble))(l, a_1) \equiv valid?(uidt(dt_longdouble))(l, a_2)
{-6}	size(uidt(dt_longdouble)) > 0
{-7}	floating_point?(longdouble)
{-8}	typ' = longdouble
{1}	$\forall (d: ((\text{range_floating_point}(\text{typ}')), a: \text{Address}):$ down(from_byte(dt_longdouble)(to_byte(dt_longdouble)(d, a), a)) = d

Repeatedly Skolemizing and flattening,

Instantiating quantified variables,

Expanding the definition of range_floating_point,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of dt_floating_point_TCC3.2.2.

Q.E.D.

C.43.23 Cpp_Integral_Types.range_integral_unsigned_TCC1

Terse proof for range_integral_unsigned_TCC1.

range_integral_unsigned_TCC1:

{1}	$\forall (\text{typ}: (\text{unsigned_integer?})): \text{non_bool_integral?}(\text{typ})$
-----	--

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of range_integral_unsigned_TCC1.

Q.E.D.

C.43.24 Cpp_Integral_Types.range_integral_unsignedTerse proof for `range_integral_unsigned`.`range_integral_unsigned:`

{1}	$\forall (\text{typ}: (\text{unsigned_integer?})): \\ (\forall (x: \text{int}): \text{range_integral}(\text{typ})(x) \supset x \geq 0) \wedge \\ \text{is_finite}[\text{nat}](\text{range_integral}(\text{typ})) \wedge \neg \text{empty?}[\text{nat}](\text{range_integral}(\text{typ}))$
-----	---

Repeatedly Skolemizing and flattening,

Case splitting on `FORALL (x: int): range_integral(typ!1)(x) IMPLIES x >= 0`,

we get 3 subgoals:

`range_integral_unsigned.1:`

{-1}	$\forall (x: \text{int}): \text{range_integral}(\text{typ}') (x) \supset x \geq 0$
{-2}	<code>unsigned_integer?(typ')</code>
{1}	$(\forall (x: \text{int}): \text{range_integral}(\text{typ}') (x) \supset x \geq 0) \wedge \\ \text{is_finite}[\text{nat}](\text{range_integral}(\text{typ}')) \wedge \neg \text{empty?}[\text{nat}](\text{range_integral}(\text{typ}'))$

Adding type constraints for `range_integral(typ!1)`,

Applying propositional simplification,

we get 2 subgoals:

`range_integral_unsigned.1.1:`

{-1}	<code>is_finite[int](range_integral(typ'))</code>
{-2}	$\forall (x: \text{int}): \text{range_integral}(\text{typ}') (x) \supset x \geq 0$
{-3}	<code>unsigned_integer?(typ')</code>
{1}	<code>is_finite[nat](range_integral(typ'))</code>
{2}	<code>empty?[int](range_integral(typ'))</code>

Expanding the definition of `is_finite`,

which is trivially true.

This completes the proof of `range_integral_unsigned.1.1`.`range_integral_unsigned.1.2:`

{-1}	<code>empty?[nat](range_integral(typ'))</code>
{-2}	<code>is_finite[int](range_integral(typ'))</code>
{-3}	$\forall (x: \text{int}): \text{range_integral}(\text{typ}') (x) \supset x \geq 0$
{-4}	<code>unsigned_integer?(typ')</code>
{1}	<code>empty?[int](range_integral(typ'))</code>

Expanding the definition of `empty?`,

Repeatedly Skolemizing and flattening,

Instantiating quantified variables,

we get 2 subgoals:

`range_integral_unsigned.1.2.1:`

{-1}	<code>integer_pred(x')</code>
{-2}	<code>is_finite[int](range_integral(typ'))</code>
{-3}	$(x' \in \text{range_integral}(\text{typ}'))$
{-4}	$\forall (x: \text{int}): \text{range_integral}(\text{typ}') (x) \supset x \geq 0$
{-5}	<code>unsigned_integer?(typ')</code>
{1}	$(x' \in \text{range_integral}(\text{typ}'))$

Expanding the definition of member,
which is trivially true.

This completes the proof of `range_integral_unsigned.1.2.1`.
`range_integral_unsigned.1.2.2`:

{-1}	<code>integer_pred(x')</code>
{-2}	<code>is_finite[int](range_integral(typ'))</code>
{-3}	<code>(x' ∈ range_integral(typ'))</code>
{-4}	$\forall (x: \text{int}): \text{range_integral}(typ')(x) \supset x \geq 0$
{-5}	<code>unsigned_integer?(typ')</code>
{1}	$x' \geq 0$

Instantiating quantified variables,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Expanding the definition of member,
which is trivially true.

This completes the proof of `range_integral_unsigned.1.2.2`.
`range_integral_unsigned.2`:

{-1}	<code>unsigned_integer?(typ')</code>
{1}	$\forall (x: \text{int}): \text{range_integral}(typ')(x) \supset x \geq 0$
{2}	$(\forall (x: \text{int}): \text{range_integral}(typ')(x) \supset x \geq 0) \wedge$ $\text{is_finite}[\text{nat}](\text{range_integral}(typ')) \wedge \neg \text{empty?}[\text{nat}](\text{range_integral}(typ'))$

Hiding formulas: 2,
Repeatedly Skolemizing and flattening,
Expanding the definition of `unsigned_integer?`,
Expanding the definition of `range_integral`,
Adding type constraints for `range_uchar`,
Adding type constraints for `range_ulonglong`,
Adding type constraints for `range_ulong`,
Adding type constraints for `range_ushort`,
Adding type constraints for `range_uint`,
Expanding the definition of `extend`,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `range_integral_unsigned.2`.

`range_integral_unsigned.3`:

{-1}	<code>unsigned_integer?(typ')</code>
{1}	<code>non_bool_integral?(typ')</code>
{2}	$(\forall (x: \text{int}): \text{range_integral}(typ')(x) \supset x \geq 0) \wedge$ $\text{is_finite}[\text{nat}](\text{range_integral}(typ')) \wedge \neg \text{empty?}[\text{nat}](\text{range_integral}(typ'))$

Expanding the definition of `non_bool_integral?`,
which is trivially true.

This completes the proof of `range_integral_unsigned.3`.
Q.E.D.

C.43.25 Cpp_Integral_Types.max_value_TCC1

Terse proof for `max_value_TCC1`.

`max_value_TCC1`:

{1}	$\forall (\text{typ}: (\text{non_bool_integral?})):$ $\neg \text{empty?}[\text{real}](\text{extend}[\text{real}, \text{int}, \text{bool}, \text{FALSE}](\text{range_integral}(\text{typ})))$
-----	--

C Proof scripts

Repeatedly Skolemizing and flattening,
Adding type constraints for `range_integral(typ!1)`,
Expanding the definition of `empty?`,
Repeatedly Skolemizing and flattening,
Instantiating quantified variables,
Keeping `(-3 1)` and hiding `*`,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `max_value_TCC1`.
Q.E.D.

C.43.26 Cpp_Integral_Types.max_value_TCC2

Terse proof for `max_value_TCC2`.

`max_value_TCC2`:

$\{1\} \quad \forall (\text{typ}: (\text{non_bool_integral?})): \\ \text{rational_pred}(\text{max}[\text{real}, \leq](\text{extend}[\text{real}, \text{int}, \text{bool}, \text{FALSE}](\text{range_integral}(\text{typ})))) \wedge \\ \text{integer_pred}(\text{max}[\text{real}, \leq](\text{extend}[\text{real}, \text{int}, \text{bool}, \text{FALSE}](\text{range_integral}(\text{typ}))))$
--

Repeatedly Skolemizing and flattening,
Adding type constraints for `max[real, <=](extend[real, int, bool, FALSE](range_integral(typ!1)))`,
we get 2 subgoals:

`max_value_TCC2.1`:

$\{-1\} \quad \text{extend}[\text{real}, \text{int}, \text{bool}, \text{FALSE}] \\ (\text{range_integral}(\text{typ}')) \\ (\text{max}[\text{real}, \leq](\text{extend}[\text{real}, \text{int}, \text{bool}, \text{FALSE}](\text{range_integral}(\text{typ}'))))$
$\{-2\} \quad \forall (x: \text{real}): \\ \text{extend}[\text{real}, \text{int}, \text{bool}, \text{FALSE}](\text{range_integral}(\text{typ}'))(x) \supset \\ x \leq \text{max}[\text{real}, \leq](\text{extend}[\text{real}, \text{int}, \text{bool}, \text{FALSE}](\text{range_integral}(\text{typ}')))$
$\{-3\} \quad \text{non_bool_integral?}(\text{typ}')$
$\{1\} \quad \text{rational_pred}(\text{max}[\text{real}, \leq](\text{extend}[\text{real}, \text{int}, \text{bool}, \text{FALSE}](\text{range_integral}(\text{typ}')))) \wedge \\ \text{integer_pred}(\text{max}[\text{real}, \leq](\text{extend}[\text{real}, \text{int}, \text{bool}, \text{FALSE}](\text{range_integral}(\text{typ}'))))$

Expanding the definition of `extend`,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `max_value_TCC2.1`.

`max_value_TCC2.2`:

$\{-1\} \quad \text{empty?}[\text{real}](\text{extend}[\text{real}, \text{int}, \text{bool}, \text{FALSE}](\text{range_integral}(\text{typ}')))$
$\{-2\} \quad \text{non_bool_integral?}(\text{typ}')$
$\{1\} \quad \text{rational_pred}(\text{max}[\text{real}, \leq](\text{extend}[\text{real}, \text{int}, \text{bool}, \text{FALSE}](\text{range_integral}(\text{typ}')))) \wedge \\ \text{integer_pred}(\text{max}[\text{real}, \leq](\text{extend}[\text{real}, \text{int}, \text{bool}, \text{FALSE}](\text{range_integral}(\text{typ}'))))$

Using lemma `max_value_TCC1`,
This completes the proof of `max_value_TCC2.2`.
Q.E.D.

C.43.27 Cpp_Integral_Types.min_value_TCC1

Terse proof for `min_value_TCC1`.

min_value_TCC1:

$$\frac{}{\{1\} \quad \forall (\text{typ}: (\text{non_bool_integral?})): \\ \text{rational_pred}(\text{min}[\text{real}, \leq](\text{extend}[\text{real}, \text{int}, \text{bool}, \text{FALSE}](\text{range_integral}(\text{typ})))) \wedge \\ \text{integer_pred}(\text{min}[\text{real}, \leq](\text{extend}[\text{real}, \text{int}, \text{bool}, \text{FALSE}](\text{range_integral}(\text{typ}))))}$$

Repeatedly Skolemizing and flattening,

Adding type constraints for $\text{min}[\text{real}, \leq](\text{extend}[\text{real}, \text{int}, \text{bool}, \text{FALSE}](\text{range_integral}(\text{typ}!1)))$,

we get 2 subgoals:

min_value_TCC1.1:

$$\frac{\begin{array}{l} \{-1\} \quad \text{extend}[\text{real}, \text{int}, \text{bool}, \text{FALSE}] \\ \quad (\text{range_integral}(\text{typ}')) \\ \quad (\text{min}[\text{real}, \leq](\text{extend}[\text{real}, \text{int}, \text{bool}, \text{FALSE}](\text{range_integral}(\text{typ}')))) \\ \{-2\} \quad \forall (x: \text{real}): \\ \quad \text{extend}[\text{real}, \text{int}, \text{bool}, \text{FALSE}](\text{range_integral}(\text{typ}'))(x) \supset \\ \quad \text{min}[\text{real}, \leq](\text{extend}[\text{real}, \text{int}, \text{bool}, \text{FALSE}](\text{range_integral}(\text{typ}')))) \leq x \\ \{-3\} \quad \text{non_bool_integral?}(\text{typ}') \end{array}}{\{1\} \quad \text{rational_pred}(\text{min}[\text{real}, \leq](\text{extend}[\text{real}, \text{int}, \text{bool}, \text{FALSE}](\text{range_integral}(\text{typ}')))) \wedge \\ \text{integer_pred}(\text{min}[\text{real}, \leq](\text{extend}[\text{real}, \text{int}, \text{bool}, \text{FALSE}](\text{range_integral}(\text{typ}'))))}$$

Expanding the definition of extend,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of min_value_TCC1.1.

min_value_TCC1.2:

$$\frac{\begin{array}{l} \{-1\} \quad \text{empty?}[\text{real}](\text{extend}[\text{real}, \text{int}, \text{bool}, \text{FALSE}](\text{range_integral}(\text{typ}')))) \\ \{-2\} \quad \text{non_bool_integral?}(\text{typ}') \end{array}}{\{1\} \quad \text{rational_pred}(\text{min}[\text{real}, \leq](\text{extend}[\text{real}, \text{int}, \text{bool}, \text{FALSE}](\text{range_integral}(\text{typ}')))) \wedge \\ \text{integer_pred}(\text{min}[\text{real}, \leq](\text{extend}[\text{real}, \text{int}, \text{bool}, \text{FALSE}](\text{range_integral}(\text{typ}'))))}$$

Using lemma max_value_TCC1,

This completes the proof of min_value_TCC1.2.

Q.E.D.

C.43.28 Cpp_Integral_Types.max_value_unsigned

Terse proof for max_value_unsigned.

max_value_unsigned:

$$\frac{}{\{1\} \quad \forall (\text{typ}: (\text{unsigned_integer?})): \text{max_value}(\text{typ}) \geq 0}$$

Repeatedly Skolemizing and flattening,

Expanding the definition of max_value,

Using lemma range_integral_unsigned,

Applying disjunctive simplification to flatten sequent,

Instantiating quantified variables,

we get 2 subgoals:

`max_value_unsigned.1:`

$$\frac{\begin{array}{l} \{-1\} \text{ range_integral}(typ')(\max(\text{extend}[\text{real}, \text{int}, \text{bool}, \text{FALSE}](\text{range_integral}(typ')))) \supset \\ \quad \max(\text{extend}[\text{real}, \text{int}, \text{bool}, \text{FALSE}](\text{range_integral}(typ'))) \geq 0 \\ \{-2\} \text{ unsigned_integer?}(typ') \end{array}}{\begin{array}{l} \{1\} \text{ empty?}[\text{nat}](\text{range_integral}(typ')) \\ \{2\} \max(\text{extend}[\text{real}, \text{int}, \text{bool}, \text{FALSE}](\text{range_integral}(typ'))) \geq 0 \end{array}}$$

Adding type constraints for $\max(\text{extend}[\text{real}, \text{int}, \text{bool}, \text{FALSE}](\text{range_integral}(typ!1)))$,

Expanding the definition of `extend`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `max_value_unsigned.1`.

`max_value_unsigned.2:`

$$\frac{\begin{array}{l} \{-1\} \text{ unsigned_integer?}(typ') \end{array}}{\begin{array}{l} \{1\} \text{ rational_pred}(\max[\text{real}, \leq](\text{extend}[\text{real}, \text{int}, \text{bool}, \text{FALSE}](\text{range_integral}(typ')))) \wedge \\ \quad \text{integer_pred}(\max[\text{real}, \leq](\text{extend}[\text{real}, \text{int}, \text{bool}, \text{FALSE}](\text{range_integral}(typ')))) \\ \{2\} \text{ empty?}[\text{nat}](\text{range_integral}(typ')) \\ \{3\} \max(\text{extend}[\text{real}, \text{int}, \text{bool}, \text{FALSE}](\text{range_integral}(typ'))) \geq 0 \end{array}}$$

Using lemma `max_value_TCC2`,

This completes the proof of `max_value_unsigned.2`.

Q.E.D.

C.43.29 Cpp_Integral_Types.min_value_unsigned

Terse proof for `min_value_unsigned`.

`min_value_unsigned:`

$$\frac{}{\{1\} \forall (typ: (\text{unsigned_integer?})): \text{min_value}(typ) \geq 0}$$

Expanding the definition of `min_value`,

Repeatedly Skolemizing and flattening,

Using lemma `range_integral_unsigned`,

Applying disjunctive simplification to flatten sequent,

Instantiating quantified variables,

we get 2 subgoals:

`min_value_unsigned.1:`

$$\frac{\begin{array}{l} \{-1\} \text{ range_integral}(typ')(\min(\text{extend}[\text{real}, \text{int}, \text{bool}, \text{FALSE}](\text{range_integral}(typ')))) \supset \\ \quad \min(\text{extend}[\text{real}, \text{int}, \text{bool}, \text{FALSE}](\text{range_integral}(typ'))) \geq 0 \\ \{-2\} \text{ unsigned_integer?}(typ') \end{array}}{\begin{array}{l} \{1\} \text{ empty?}[\text{nat}](\text{range_integral}(typ')) \\ \{2\} \min(\text{extend}[\text{real}, \text{int}, \text{bool}, \text{FALSE}](\text{range_integral}(typ'))) \geq 0 \end{array}}$$

Adding type constraints for $\min(\text{extend}[\text{real}, \text{int}, \text{bool}, \text{FALSE}](\text{range_integral}(typ!1)))$,

Expanding the definition of `extend`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `min_value_unsigned.1`.

min_value_unsigned.2:

{-1}	unsigned_integer?(typ')
{1}	rational_pred(min[real, ≤](extend[real, int, bool, FALSE](range_integral(typ')))) ∧ integer_pred(min[real, ≤](extend[real, int, bool, FALSE](range_integral(typ'))))
{2}	empty?[nat](range_integral(typ'))
{3}	min(extend[real, int, bool, FALSE](range_integral(typ'))) ≥ 0

Using lemma min_value_TCC1,
This completes the proof of min_value_unsigned.2.
Q.E.D.

C.43.30 Cpp_Integral_Types.max_value_bits_TCC1

Terse proof for max_value_bits_TCC1.

max_value_bits_TCC1:

{1}	well_founded?(λ (x: (cv(unsigned_integer?)), y: (cv(unsigned_integer?))): restrict [[Cpp_Type_, Cpp_Type_], [(cv(unsigned_integer?)), (cv(unsigned_integer?))], boolean] (<<)(x, y))
-----	--

Using lemma well_founded_restrict[Cpp_Type_, (cv(unsigned_integer?))],
Expanding the definition of restrict,
which is trivially true.
This completes the proof of max_value_bits_TCC1.
Q.E.D.

C.43.31 Cpp_Integral_Types.max_value_bits_TCC2

Terse proof for max_value_bits_TCC2.

max_value_bits_TCC2:

{1}	∀ (typ: (cv(unsigned_integer?)), t: Cpp_Type_): typ = const(t) ⊃ cv(unsigned_integer?)(t)
-----	--

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of max_value_bits_TCC2.
Q.E.D.

C.43.32 Cpp_Integral_Types.max_value_bits_TCC3

Terse proof for max_value_bits_TCC3.

max_value_bits_TCC3:

{1}	∀ (typ: (cv(unsigned_integer?)), t: Cpp_Type_): typ = const(t) ⊃ restrict [[Cpp_Type_, Cpp_Type_], [(cv(unsigned_integer?)), (cv(unsigned_integer?))], boolean] (<<)(t, typ)
-----	---

C Proof scripts

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `max_value_bits_TCC3`.

Q.E.D.

C.43.33 Cpp_Integral_Types.max_value_bits_TCC4

Terse proof for `max_value_bits_TCC4`.

`max_value_bits_TCC4:`

$$\{1\} \quad \forall (\text{typ}: (\text{cv}(\text{unsigned_integer?})), t: \text{Cpp_Type_}):$$
$$\text{typ} = \text{volatile}(t) \supset \text{cv}(\text{unsigned_integer?})(t)$$

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `max_value_bits_TCC4`.

Q.E.D.

C.43.34 Cpp_Integral_Types.max_value_bits_TCC5

Terse proof for `max_value_bits_TCC5`.

`max_value_bits_TCC5:`

$$\{1\} \quad \forall (\text{typ}: (\text{cv}(\text{unsigned_integer?})), t: \text{Cpp_Type_}):$$
$$\text{typ} = \text{volatile}(t) \supset$$
$$\text{restrict}$$
$$\quad [[\text{Cpp_Type_}, \text{Cpp_Type_}], [(\text{cv}(\text{unsigned_integer?})), (\text{cv}(\text{unsigned_integer?}))],$$
$$\quad \text{boolean}]$$
$$(\ll)(t, \text{typ})$$

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `max_value_bits_TCC5`.

Q.E.D.

C.43.35 Cpp_Integral_Types.max_value_bits_TCC6

Terse proof for `max_value_bits_TCC6`.

max_value_bits_TCC6:

```
{1}  ∀ (typ: (cv(unsigned_integer?))):
      ¬ (schar?(typ) ∨
         char?(typ) ∨
         short?(typ) ∨
         int?(typ) ∨
         long?(typ) ∨
         longlong?(typ) ∨
         wchar_t?(typ) ∨
         bool?(typ) ∨
         float?(typ) ∨
         double?(typ) ∨
         longdouble?(typ) ∨
         void?(typ) ∨
         array?(typ) ∨
         function?(typ) ∨
         pointer?(typ) ∨
         reference?(typ) ∨
         class?(typ) ∨
         union?(typ) ∨
         bitfield?(typ) ∨ enum?(typ) ∨ pointer_to_member?(typ))
```

Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of max_value_bits_TCC6.
 Q.E.D.

C.44 Proofs for Cpp_Types (types.pvs)

C.44.1 Cpp_Types.alignment_TCC1

Terse proof for alignment_TCC1.

alignment_TCC1:

```
{1}  well_founded?(λ (x: Cpp_Type, y: Cpp_Type):
                  restrict[[Cpp_Type_, Cpp_Type_], [Cpp_Type, Cpp_Type], boolean]
                  (<<)(x, y))
```

Using lemma well_founded_restrict[Cpp_Type_, Cpp_Type],
 Expanding the definition of restrict,
 which is trivially true.
 This completes the proof of alignment_TCC1.
 Q.E.D.

C.44.2 Cpp_Types.alignment_TCC2

Terse proof for alignment_TCC2.

alignment_TCC2:

```
{1}  ∀ (typ: Cpp_Type, t: Cpp_Type_, s: nat): typ = array_type(t, s) ⊃ array?(typ)
```


Repeatedly Skolemizing and flattening,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `alignment_TCC2`.
Q.E.D.

C.44.3 Cpp_Types.alignment_TCC3

Terse proof for `alignment_TCC3`.

`alignment_TCC3`:

$$\{1\} \quad \forall (\text{typ}: \text{Cpp_Type}, t: \text{Cpp_Type_}): \text{typ} = \text{pointer}(t) \supset \text{pointer?}(\text{typ})$$

Repeatedly Skolemizing and flattening,
Replacing using formula -2,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `alignment_TCC3`.
Q.E.D.

C.44.4 Cpp_Types.alignment_TCC4

Terse proof for `alignment_TCC4`.

`alignment_TCC4`:

$$\{1\} \quad \forall (\text{typ}: \text{Cpp_Type}, t: \text{Cpp_Type_}): \text{typ} = \text{reference}(t) \supset \text{reference?}(\text{typ})$$

Repeatedly Skolemizing and flattening,
Replacing using formula -2,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `alignment_TCC4`.
Q.E.D.

C.44.5 Cpp_Types.alignment_TCC5

Terse proof for `alignment_TCC5`.

`alignment_TCC5`:

$$\{1\} \quad \forall (\text{typ}: \text{Cpp_Type}, n: \text{string}, a: \text{bool}): \text{typ} = \text{class}(n, a) \supset \text{class?}(\text{typ})$$

Repeatedly Skolemizing and flattening,
Replacing using formula -2,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `alignment_TCC5`.
Q.E.D.

C.44.6 Cpp_Types.alignment_TCC6

Terse proof for `alignment_TCC6`.

`alignment_TCC6`:

$$\{1\} \quad \forall (\text{typ}: \text{Cpp_Type}, n: \text{string}): \text{typ} = \text{union}(n) \supset \text{union?}(\text{typ})$$

Repeatedly Skolemizing and flattening,
 Replacing using formula -2,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 This completes the proof of `alignment_TCC6`.
 Q.E.D.

C.44.7 Cpp_Types.alignment_TCC7

Terse proof for `alignment_TCC7`.

`alignment_TCC7`:

$$\frac{}{\{1\} \quad \forall (\text{typ}: \text{Cpp_Type}, t: \text{Cpp_Type_}, \text{bo}: \text{nat}, s: \text{posnat}): \text{typ} = \text{bitfield}(t, \text{bo}, s) \supset \text{bitfield}?(typ)}$$

Repeatedly Skolemizing and flattening,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 This completes the proof of `alignment_TCC7`.
 Q.E.D.

C.44.8 Cpp_Types.alignment_TCC8

Terse proof for `alignment_TCC8`.

`alignment_TCC8`:

$$\frac{}{\{1\} \quad \forall (\text{typ}: \text{Cpp_Type}, n: \text{string}, t: \text{Cpp_Type_}, m: \text{posnat}, c: [\text{below}[m] \rightarrow \text{int}]): \text{typ} = \text{enum}(n, t, m, c) \supset \text{enum}?(typ)}$$

Repeatedly Skolemizing and flattening,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 This completes the proof of `alignment_TCC8`.
 Q.E.D.

C.44.9 Cpp_Types.alignment_TCC9

Terse proof for `alignment_TCC9`.

`alignment_TCC9`:

$$\frac{}{\{1\} \quad \forall (\text{typ}: \text{Cpp_Type}, t: (\text{class?}), m: \text{Cpp_Type_}): \text{typ} = \text{pointer_to_member}(t, m) \supset \text{pointer_to_member}?(typ)}$$

Repeatedly Skolemizing and flattening,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 This completes the proof of `alignment_TCC9`.
 Q.E.D.

C.44.10 Cpp_Types.alignment_TCC10

Terse proof for `alignment_TCC10`.

`alignment_TCC10`:

$$\frac{}{\{1\} \quad \forall (\text{typ}: \text{Cpp_Type}, t: \text{Cpp_Type_}): \text{typ} = \text{const}(t) \supset \text{Cpp_Type}?(t)}$$

Repeatedly Skolemizing and flattening,
 Replacing using formula -2,
 Expanding the definition of Cpp_Type?,
 Repeatedly Skolemizing and flattening,
 Instantiating quantified variables,
 Expanding the definition of subterm,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 This completes the proof of `alignment_TCC10`.
 Q.E.D.

C.44.11 Cpp_Types.alignment_TCC11

Terse proof for `alignment_TCC11`.

`alignment_TCC11`:

$\{1\}$	$\forall (\text{typ}: \text{Cpp_Type}, t: \text{Cpp_Type_}):$ $\text{typ} = \text{const}(t) \supset$ $\text{restrict}[[\text{Cpp_Type_}, \text{Cpp_Type_}], [\text{Cpp_Type}, \text{Cpp_Type}], \text{boolean}](\ll)(t, \text{typ})$
---------	---

Repeatedly Skolemizing and flattening,
 Expanding the definition of restrict,
 Replacing using formula -2,
 Expanding the definition of \blacksquare ,
 which is trivially true.
 This completes the proof of `alignment_TCC11`.
 Q.E.D.

C.44.12 Cpp_Types.alignment_TCC12

Terse proof for `alignment_TCC12`.

`alignment_TCC12`:

$\{1\}$	$\forall (\text{typ}: \text{Cpp_Type}, t: \text{Cpp_Type_}): \text{typ} = \text{volatile}(t) \supset \text{Cpp_Type?}(t)$
---------	---

Repeatedly Skolemizing and flattening,
 Expanding the definition of Cpp_Type?,
 Repeatedly Skolemizing and flattening,
 Instantiating the top quantifier in -1 with the terms: (t!2),
 Expanding the definition of subterm,
 Replacing using formula -2,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 This completes the proof of `alignment_TCC12`.
 Q.E.D.

C.44.13 Cpp_Types.alignment_TCC13

Terse proof for `alignment_TCC13`.

alignment_TCC13:

$$\{1\} \quad \forall (\text{typ}: \text{Cpp_Type}, t: \text{Cpp_Type_}):$$

$$\text{typ} = \text{volatile}(t) \supset$$

$$\text{restrict}[[\text{Cpp_Type_}, \text{Cpp_Type_}], [\text{Cpp_Type}, \text{Cpp_Type}], \text{boolean}](\ll)(t, \text{typ})$$

Repeatedly Skolemizing and flattening,

Expanding the definition of restrict,

Replacing using formula -2,

Expanding the definition of ■,

which is trivially true.

This completes the proof of alignment_TCC13.

Q.E.D.

C.44.14 Cpp_Types.pointer_to_aligned_object_TCC1

Terse proof for pointer_to_aligned_object_TCC1.

pointer_to_aligned_object_TCC1:

$$\{1\} \quad \forall (p: \text{Cpp_Subtype}(\text{cv}(\text{pointer?})), \text{ptr_val}: (\text{range_pointer}(\text{cv_base}(p))))):$$

$$\text{not_null?}(p)(\text{ptr_val}) \supset$$

$$\text{array?}(\text{cv_base}(p)) \vee$$

$$\text{pointer?}(\text{cv_base}(p)) \vee$$

$$\text{reference?}(\text{cv_base}(p)) \vee$$

$$\text{bitfield?}(\text{cv_base}(p)) \vee$$

$$\text{enum?}(\text{cv_base}(p)) \vee$$

$$\text{pointer_to_member?}(\text{cv_base}(p)) \vee \text{const?}(\text{cv_base}(p)) \vee \text{volatile?}(\text{cv_base}(p))$$

Repeatedly Skolemizing and flattening,

Keeping (-2 2) and hiding *,

Rewriting using cv_base_result, matching in *,

Keeping (1) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of pointer_to_aligned_object_TCC1.

Q.E.D.

C.44.15 Cpp_Types.pointer_to_aligned_object_TCC2

Terse proof for pointer_to_aligned_object_TCC2.

pointer_to_aligned_object_TCC2:

$$\{1\} \quad \forall (p: \text{Cpp_Subtype}(\text{cv}(\text{pointer?})), \text{ptr_val}: (\text{range_pointer}(\text{cv_base}(p))))):$$

$$\text{not_null?}(p)(\text{ptr_val}) \supset \text{Cpp_Type?}(\text{typ}(\text{cv_base}(p)))$$

Repeatedly Skolemizing and flattening,

Expanding the definition of Cpp_Type?,

Repeatedly Skolemizing and flattening,

Instantiating the top quantifier in -1 with the terms: (t!1),

Splitting conjunctions,

we get 2 subgoals:

pointer_to_aligned_object_TCC2.1:

```

{-1} no_pointers_to_bitfield?(t') ∧
     no_pointers_to_references?(t') ∧
     no_cv_references?(t') ∧
     no_reference_to_reference?(t') ∧
     no_reference_to_bitfields?(t') ∧
     no_pointer_to_member_to_reference?(t') ∧
     no_pointer_to_member_to_cv_void?(t') ∧
     no_cv_void_parameter?(t') ∧
     no_array_of_references?(t') ∧
     no_array_of_cv_void?(t') ∧
     no_array_of_function?(t') ∧
     no_array_of_abstract_class?(t') ∧
     no_pointer_or_ref_to_incomplete_array_parameter?(t') ∧
     no_array_return_type?(t') ∧
     no_function_return_type?(t') ∧
     bitfield_underlying_integral_or_enum_type?(t') ∧
     cv_array?(t') ∧
     enum_underlying_integral?(t') ∧
     enum_constants?(t') ∧
     const_volatile?(t') ∧
     const_stutter?(t') ∧
     volatile_stutter?(t') ∧
     no_cv_class?(t') ∧
     no_cv_union?(t') ∧
     no_cv_function?(t') ∧
     no_reference_to_void?(t') ∧
     no_cv_void?(t') ∧ no_array_of_bitfields?(t')
{-2} cv(pointer?)(p')
{-3} range_pointer(cv_base(p'))(ptr_val')
{-4} not_null?(p')(ptr_val')
{-5} subterm(t', typ(cv_base(p')))
-----
{1} no_pointers_to_bitfield?(t') ∧
     no_pointers_to_references?(t') ∧
     no_cv_references?(t') ∧
     no_reference_to_reference?(t') ∧
     no_reference_to_bitfields?(t') ∧
     no_pointer_to_member_to_reference?(t') ∧
     no_pointer_to_member_to_cv_void?(t') ∧
     no_cv_void_parameter?(t') ∧
     no_array_of_references?(t') ∧
     no_array_of_cv_void?(t') ∧
     no_array_of_function?(t') ∧
     no_array_of_abstract_class?(t') ∧
     no_pointer_or_ref_to_incomplete_array_parameter?(t') ∧
     no_array_return_type?(t') ∧
     no_function_return_type?(t') ∧
     bitfield_underlying_integral_or_enum_type?(t') ∧
     cv_array?(t') ∧
     enum_underlying_integral?(t') ∧
     enum_constants?(t') ∧
     const_volatile?(t') ∧
     const_stutter?(t') ∧
     volatile_stutter?(t') ∧
     no_cv_class?(t') ∧
     no_cv_union?(t') ∧
     no_cv_function?(t') ∧
     no_reference_to_void?(t') ∧
     no_cv_void?(t') ∧ no_array_of_bitfields?(t')

```

which is trivially true.

This completes the proof of `pointer_to_aligned_object_TCC2.1`.

`pointer_to_aligned_object_TCC2.2`:

{-1}	<code>cv(pointer?)(p')</code>
{-2}	<code>range_pointer(cv_base(p'))(ptr_val')</code>
{-3}	<code>not_null?(p')(ptr_val')</code>
{-4}	<code>subterm(t', typ(cv_base(p')))</code>
{1}	<code>subterm(t', p')</code>
{2}	$\begin{aligned} &\text{no_pointers_to_bitfield?}(t') \wedge \\ &\text{no_pointers_to_references?}(t') \wedge \\ &\text{no_cv_references?}(t') \wedge \\ &\text{no_reference_to_reference?}(t') \wedge \\ &\text{no_reference_to_bitfields?}(t') \wedge \\ &\text{no_pointer_to_member_to_reference?}(t') \wedge \\ &\text{no_pointer_to_member_to_cv_void?}(t') \wedge \\ &\text{no_cv_void_parameter?}(t') \wedge \\ &\text{no_array_of_references?}(t') \wedge \\ &\text{no_array_of_cv_void?}(t') \wedge \\ &\text{no_array_of_function?}(t') \wedge \\ &\text{no_array_of_abstract_class?}(t') \wedge \\ &\text{no_pointer_or_ref_to_incomplete_array_parameter?}(t') \wedge \\ &\text{no_array_return_type?}(t') \wedge \\ &\text{no_function_return_type?}(t') \wedge \\ &\text{bitfield_underlying_integral_or_enum_type?}(t') \wedge \\ &\text{cv_array?}(t') \wedge \\ &\text{enum_underlying_integral?}(t') \wedge \\ &\text{enum_constants?}(t') \wedge \\ &\text{const_volatile?}(t') \wedge \\ &\text{const_stutter?}(t') \wedge \\ &\text{volatile_stutter?}(t') \wedge \\ &\text{no_cv_class?}(t') \wedge \\ &\text{no_cv_union?}(t') \wedge \\ &\text{no_cv_function?}(t') \wedge \\ &\text{no_reference_to_void?}(t') \wedge \\ &\text{no_cv_void?}(t') \wedge \text{no_array_of_bitfields?}(t') \end{aligned}$

Hiding formulas: 2,

Using lemma `subterm_cv_base`,

Using lemma `subterm_transitive`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Using lemma `cv_base_result`,

we get 2 subgoals:

`pointer_to_aligned_object_TCC2.2.1`:

{-1}	<code>pointer?(cv_base(p'))</code>
{-2}	<code>subterm(cv_base(p'), p')</code>
{-3}	<code>cv(pointer?)(p')</code>
{-4}	<code>range_pointer(cv_base(p'))(ptr_val')</code>
{-5}	<code>not_null?(p')(ptr_val')</code>
{-6}	<code>subterm(t', typ(cv_base(p')))</code>
{1}	<code>subterm(t', cv_base(p'))</code>
{2}	<code>subterm(t', p')</code>

Expanding the definition of `subterm`,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 This completes the proof of `pointer_to_aligned_object_TCC2.2.1`.

`pointer_to_aligned_object_TCC2.2.2`:

{-1}	<code>subterm(cv_base(p'), p')</code>
{-2}	<code>cv(pointer?)(p')</code>
{-3}	<code>range_pointer(cv_base(p'))(ptr_val')</code>
{-4}	<code>not_null?(p')(ptr_val')</code>
{-5}	<code>subterm(t', typ(cv_base(p')))</code>
{1}	<code>(pointer? \subseteq interpreted?)</code>
{2}	<code>subterm(t', cv_base(p'))</code>
{3}	<code>subterm(t', p')</code>

Keeping (1) and hiding *,
 Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of `pointer_to_aligned_object_TCC2.2.2`.
 Q.E.D.

C.44.16 Cpp_Types.pointer_to_aligned_object_TCC3

Terse proof for `pointer_to_aligned_object_TCC3`.

`pointer_to_aligned_object_TCC3`:

{1}	$\forall (p: \text{Cpp_Subtype}(\text{cv}(\text{pointer?})), \text{ptr_val}: (\text{range_pointer}(\text{cv_base}(p)))):$ $\text{not_null?}(p)(\text{ptr_val}) \supset \text{up?}[\text{Memory_Address}](\text{address_of}(p)(\text{ptr_val}))$
-----	---

Repeatedly Skolemizing and flattening,
 Rewriting using `address_of_spec`, matching in *,
 This completes the proof of `pointer_to_aligned_object_TCC3`.
 Q.E.D.

C.44.17 Cpp_Types.align_array_of_type_TCC1

Terse proof for `align_array_of_type_TCC1`.

`align_array_of_type_TCC1`:

{1}	$\forall (a: \text{Cpp_Subtype}(\text{array?})):$ $\text{array?}(a) \vee$ $\text{pointer?}(a) \vee$ $\text{reference?}(a) \vee$ $\text{bitfield?}(a) \vee \text{enum?}(a) \vee \text{pointer_to_member?}(a) \vee \text{const?}(a) \vee \text{volatile?}(a)$
-----	--

Repeatedly Skolemizing and flattening,
 This completes the proof of `align_array_of_type_TCC1`.
 Q.E.D.

C.44.18 Cpp_Types.align_array_of_type_TCC2

Terse proof for `align_array_of_type_TCC2`.

align_array_of_type_TCC2:

{1} $\forall (a: \text{Cpp_Subtype}(\text{array?})): \text{Cpp_Type?}(\text{typ}(a))$

Repeatedly Skolemizing and flattening,
 Expanding the definition of Cpp_Type?,
 Repeatedly Skolemizing and flattening,
 Instantiating the top quantifier in -1 with the terms: (t!1),
 Expanding the definition of subterm,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 This completes the proof of align_array_of_type_TCC2.
 Q.E.D.

C.44.19 Cpp_Types.range_TCC1

Terse proof for range_TCC1.

range_TCC1:

{1} well_founded? $(\lambda (x: \text{Cpp_Subtype}(\text{cv}(\text{non_bool_integral_enum?})),$
 $y: \text{Cpp_Subtype}(\text{cv}(\text{non_bool_integral_enum?})))$):
 restrict
 $[[\text{Cpp_Type_}, \text{Cpp_Type_}],$
 $[\text{Cpp_Subtype}(\text{cv}(\text{non_bool_integral_enum?})),$
 $\text{Cpp_Subtype}(\text{cv}(\text{non_bool_integral_enum?}))],$
 boolean]
 $(\ll)(x, y)$

Using lemma well_founded_restrict[Cpp_Type_, Cpp_Subtype(cv(non_bool_integral_enum?))],
 Expanding the definition of restrict,
 which is trivially true.
 This completes the proof of range_TCC1.
 Q.E.D.

C.44.20 Cpp_Types.range_TCC2

Terse proof for range_TCC2.

range_TCC2:

{1} $\forall (\text{typ}: \text{Cpp_Subtype}(\text{cv}(\text{non_bool_integral_enum?})), n: \text{string}, t: \text{Cpp_Type_}, m: \text{pos-}$
 $\text{nat},$
 $c: [\text{below}[m] \rightarrow \text{int}]$):
 $\text{typ} = \text{enum}(n, t, m, c) \supset \text{enum?}(\text{typ})$

Repeatedly Skolemizing and flattening,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 This completes the proof of range_TCC2.
 Q.E.D.

C.44.21 Cpp_Types.range_TCC3

Terse proof for range_TCC3.

range_TCC3:

{1}	$\forall (\text{typ}: \text{Cpp_Subtype}(\text{cv}(\text{non_bool_integral_enum?})), t: \text{Cpp_Type_}):$ $\text{typ} = \text{const}(t) \supset \text{Cpp_Type?}(t) \wedge \text{cv}(\text{non_bool_integral_enum?})(t)$
-----	---

Repeatedly Skolemizing and flattening,
 Applying propositional simplification,
 we get 2 subgoals:

range_TCC3.1:

{-1}	Cpp_Type?(typ')
{-2}	cv(non_bool_integral_enum?)(typ')
{-3}	typ' = const(t')
{1}	Cpp_Type?(t')

Expanding the definition of Cpp_Type?,
 Repeatedly Skolemizing and flattening,
 Instantiating the top quantifier in -2 with the terms: (t!2),
 Replacing using formula -4,
 Expanding the definition of subterm,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 This completes the proof of range_TCC3.1.

range_TCC3.2:

{-1}	Cpp_Type?(typ')
{-2}	cv(non_bool_integral_enum?)(typ')
{-3}	typ' = const(t')
{1}	cv(non_bool_integral_enum?)(t')

Replacing using formula -3,
 Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of range_TCC3.2.
 Q.E.D.

C.44.22 Cpp_Types.range_TCC4

Terse proof for range_TCC4.

range_TCC4:

{1}	$\forall (\text{typ}: \text{Cpp_Subtype}(\text{cv}(\text{non_bool_integral_enum?})), t: \text{Cpp_Type_}):$ $\text{typ} = \text{const}(t) \supset$ restrict $\quad [[\text{Cpp_Type_}, \text{Cpp_Type_}],$ $\quad \quad [\text{Cpp_Subtype}(\text{cv}(\text{non_bool_integral_enum?})),$ $\quad \quad \quad \text{Cpp_Subtype}(\text{cv}(\text{non_bool_integral_enum?}))],$ $\quad \quad \text{boolean}]$ $\quad (\ll)(t, \text{typ})$
-----	---

Expanding the definition of restrict,
 Repeatedly Skolemizing and flattening,
 Replacing using formula -3,
 Expanding the definition of ■,
 which is trivially true.
 This completes the proof of range_TCC4.

Q.E.D.

C.44.23 Cpp_Types.range_TCC5

Terse proof for range_TCC5.

range_TCC5:

$$\frac{}{\{1\} \quad \forall (\text{typ}: \text{Cpp_Subtype}(\text{cv}(\text{non_bool_integral_enum?})), t: \text{Cpp_Type_}): \\ \text{typ} = \text{volatile}(t) \supset \text{Cpp_Type?}(t) \wedge \text{cv}(\text{non_bool_integral_enum?})(t)}$$

Repeatedly Skolemizing and flattening,

Applying propositional simplification,

we get 2 subgoals:

range_TCC5.1:

$$\frac{\begin{array}{l} \{-1\} \quad \text{Cpp_Type?}(\text{typ}') \\ \{-2\} \quad \text{cv}(\text{non_bool_integral_enum?})(\text{typ}') \\ \{-3\} \quad \text{typ}' = \text{volatile}(t') \end{array}}{\{1\} \quad \text{Cpp_Type?}(t')}$$

Expanding the definition of Cpp_Type?,

Repeatedly Skolemizing and flattening,

Instantiating the top quantifier in -2 with the terms: (t!2),

Replacing using formula -4,

Expanding the definition of subterm,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of range_TCC5.1.

range_TCC5.2:

$$\frac{\begin{array}{l} \{-1\} \quad \text{Cpp_Type?}(\text{typ}') \\ \{-2\} \quad \text{cv}(\text{non_bool_integral_enum?})(\text{typ}') \\ \{-3\} \quad \text{typ}' = \text{volatile}(t') \end{array}}{\{1\} \quad \text{cv}(\text{non_bool_integral_enum?})(t')}$$

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of range_TCC5.2.

Q.E.D.

C.44.24 Cpp_Types.range_TCC6

Terse proof for range_TCC6.

range_TCC6:

$$\frac{}{\{1\} \quad \forall (\text{typ}: \text{Cpp_Subtype}(\text{cv}(\text{non_bool_integral_enum?})), t: \text{Cpp_Type_}): \\ \text{typ} = \text{volatile}(t) \supset \\ \text{restrict} \\ \quad [[\text{Cpp_Type_}, \text{Cpp_Type_}], \\ \quad [\text{Cpp_Subtype}(\text{cv}(\text{non_bool_integral_enum?})), \\ \quad \quad \text{Cpp_Subtype}(\text{cv}(\text{non_bool_integral_enum?}))], \\ \quad \text{boolean}] \\ \quad (\ll)(t, \text{typ})}$$

Repeatedly Skolemizing and flattening,

Replacing using formula -3,
 Keeping (1) and hiding *,
 Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of `range_TCC6`.
 Q.E.D.

C.44.25 Cpp_Types.range_TCC7

Terse proof for `range_TCC7`.

`range_TCC7`:

{1}	$\forall (\text{typ: Cpp_Subtype}(\text{cv}(\text{non_bool_integral_enum?})))$ $\neg \text{volatile?}(\text{typ}) \wedge \neg \text{const?}(\text{typ}) \wedge \neg \text{enum?}(\text{typ}) \supset \text{non_bool_integral?}(\text{typ})$
-----	--

Repeatedly Skolemizing and flattening,
 Expanding the definition of `cv`,
 Expanding the definition of `v`,
 Expanding the definition of `c`,
 Expanding the definition of `non_bool_integral_enum?`,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 This completes the proof of `range_TCC7`.
 Q.E.D.

C.44.26 Cpp_Types.range_cv_base_TCC1

Terse proof for `range_cv_base_TCC1`.

`range_cv_base_TCC1`:

{1}	$\forall (\text{typ: Cpp_Subtype}(\text{cv}(\text{non_bool_integral_enum?})))$ $\text{cv}(\text{non_bool_integral_enum?})(\text{cv_base}(\text{typ}))$
-----	---

Repeatedly Skolemizing and flattening,
 Hiding formulas: -1,
 Expanding the definition of `cv`,
 Expanding the definition of `c`,
 Applying disjunctive simplification to flatten sequent,
 Expanding the definition of `v`,
 Hiding formulas: (2 3 4),
 Applying propositional simplification,
 we get 5 subgoals:
`range_cv_base_TCC1.1`:

{-1}	<code>non_bool_integral_enum?(typ')</code>
{1}	<code>non_bool_integral_enum?(cv_base(typ'))</code>

Expanding the definition of `cv_base`,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 we get 2 subgoals:
`range_cv_base_TCC1.1.1`:

{-1}	<code>non_bool_integral_enum?(typ')</code>
{-2}	<code>const?(typ')</code>
{1}	<code>non_bool_integral_enum?(cv_base(typ'))</code>

Hiding formulas: 1,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `range_cv_base_TCC1.1.1`.

`range_cv_base_TCC1.1.2`:

{-1}	non_bool_integral_enum?(typ')
{-2}	volatile?(typ')
{1}	non_bool_integral_enum?(cv_base(typ(typ')))

Hiding formulas: 1,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `range_cv_base_TCC1.1.2`.

`range_cv_base_TCC1.2`:

{-1}	const?(typ')
{-2}	non_bool_integral_enum?(typ(typ'))
{1}	non_bool_integral_enum?(cv_base(typ'))

Expanding the definition of `cv_base`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Expanding the definition of `cv_base`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 2 subgoals:

`range_cv_base_TCC1.2.1`:

{-1}	const?(typ')
{-2}	non_bool_integral_enum?(typ(typ'))
{-3}	const?(typ(typ'))
{1}	non_bool_integral_enum?(cv_base(typ(typ(typ'))))

Keeping (-2 -3) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `range_cv_base_TCC1.2.1`.

`range_cv_base_TCC1.2.2`:

{-1}	const?(typ')
{-2}	non_bool_integral_enum?(typ(typ'))
{-3}	volatile?(typ(typ'))
{1}	non_bool_integral_enum?(cv_base(typ(typ(typ'))))

Keeping (-2 -3) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `range_cv_base_TCC1.2.2`.

`range_cv_base_TCC1.3`:

{-1}	non_bool_integral_enum?(typ')
{1}	non_bool_integral_enum?(cv_base(typ'))

Expanding the definition of `cv_base`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 2 subgoals:

`range_cv_base_TCC1.3.1`:

{-1}	non_bool_integral_enum?(typ')
{-2}	const?(typ')
{1}	non_bool_integral_enum?(cv_base(typ(typ')))

C Proof scripts

Keeping (-2 -1) and hiding *,
 Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of `range_cv_base_TCC1.3.1`.
`range_cv_base_TCC1.3.2`:

$$\frac{\begin{array}{l} \{-1\} \text{ non_bool_integral_enum?}(typ') \\ \{-2\} \text{ volatile?}(typ') \end{array}}{\{1\} \text{ non_bool_integral_enum?}(cv_base(typ(typ')))}$$

Keeping (-2 -1) and hiding *,
 Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of `range_cv_base_TCC1.3.2`.
`range_cv_base_TCC1.4`:

$$\frac{\begin{array}{l} \{-1\} \text{ volatile?}(typ') \\ \{-2\} \text{ non_bool_integral_enum?}(typ(typ')) \end{array}}{\{1\} \text{ non_bool_integral_enum?}(cv_base(typ'))}$$

Expanding the definition of `cv_base`,
 Expanding the definition of `cv_base`,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 we get 2 subgoals:

`range_cv_base_TCC1.4.1`:

$$\frac{\begin{array}{l} \{-1\} \text{ volatile?}(typ') \\ \{-2\} \text{ non_bool_integral_enum?}(typ(typ')) \\ \{-3\} \text{ const?}(typ(typ')) \end{array}}{\{1\} \text{ non_bool_integral_enum?}(cv_base(typ(typ(typ'))))}$$

Keeping (-2 -3) and hiding *,
 Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of `range_cv_base_TCC1.4.1`.
`range_cv_base_TCC1.4.2`:

$$\frac{\begin{array}{l} \{-1\} \text{ volatile?}(typ') \\ \{-2\} \text{ non_bool_integral_enum?}(typ(typ')) \\ \{-3\} \text{ volatile?}(typ(typ')) \end{array}}{\{1\} \text{ non_bool_integral_enum?}(cv_base(typ(typ(typ'))))}$$

Keeping (-2 -3) and hiding *,
 Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of `range_cv_base_TCC1.4.2`.
`range_cv_base_TCC1.5`:

$$\frac{\begin{array}{l} \{-1\} \text{ const?}(typ') \\ \{-2\} \text{ volatile?}(typ(typ')) \\ \{-3\} \text{ non_bool_integral_enum?}(typ(typ(typ')) \end{array}}{\{1\} \text{ non_bool_integral_enum?}(cv_base(typ'))}$$

Expanding the definition of `cv_base`,
 Expanding the definition of `cv_base`,
 Expanding the definition of `cv_base`,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 we get 2 subgoals:

range_cv_base_TCC1.5.1:

{-1}	const?(typ')	
{-2}	volatile?(typ(typ'))	
{-3}	non_bool_integral_enum?(typ(typ(typ')))	
{-4}	const?(typ(typ(typ')))	
{1}		non_bool_integral_enum?(cv_base(typ(typ')))

Keeping (-4 -3) and hiding *,
 Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of range_cv_base_TCC1.5.1.

range_cv_base_TCC1.5.2:

{-1}	const?(typ')	
{-2}	volatile?(typ(typ'))	
{-3}	non_bool_integral_enum?(typ(typ(typ')))	
{-4}	volatile?(typ(typ(typ')))	
{1}		non_bool_integral_enum?(cv_base(typ(typ(typ'))))

Keeping (-4 -3) and hiding *,
 Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of range_cv_base_TCC1.5.2.
 Q.E.D.

C.44.27 Cpp_Types.range_cv_base

Terse proof for range_cv_base.

range_cv_base:

{1}		$\forall (typ: \text{Cpp_Subtype}(cv(\text{non_bool_integral_enum?}))) : \text{range}(typ) = \text{range}(cv_base(typ))$
-----	--	---

Repeatedly Skolemizing and flattening,
 Expanding the definition of cv,
 Expanding the definition of c,
 Expanding the definition of v,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 we get 4 subgoals:

range_cv_base.1:

{-1}	Cpp_Type?(typ')	
{-2}	non_bool_integral_enum?(typ')	
{1}		range(typ') = range(cv_base(typ'))

Expanding the definition of cv_base,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 we get 2 subgoals:

range_cv_base.1.1:

{-1}	Cpp_Type?(typ')	
{-2}	non_bool_integral_enum?(typ')	
{-3}	const?(typ')	
{1}		range(typ') = range(cv_base(typ(typ')))

Keeping (-2 -3) and hiding *,
 Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `range_cv_base.1.1`.

`range_cv_base.1.2`:

{-1}	Cpp_Type?(typ')
{-2}	non_bool_integral_enum?(typ')
{-3}	volatile?(typ')
{1}	range(typ') = range(cv_base(typ(typ')))

Keeping (-2 -3) and hiding *,
 Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of `range_cv_base.1.2`.

`range_cv_base.2`:

{-1}	Cpp_Type?(typ')
{-2}	const?(typ')
{-3}	non_bool_integral_enum?(typ(typ'))
{1}	range(typ') = range(cv_base(typ'))

Expanding the definition of `cv_base`,
 Expanding the definition of `range`,
 Expanding the definition of `cv_base`,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 we get 2 subgoals:

`range_cv_base.2.1`:

{-1}	Cpp_Type?(typ')
{-2}	const?(typ')
{-3}	non_bool_integral_enum?(typ(typ'))
{-4}	const?(typ(typ'))
{1}	range(typ(typ')) = range(cv_base(typ(typ')))

Keeping (-3 -4) and hiding *,
 Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of `range_cv_base.2.1`.

`range_cv_base.2.2`:

{-1}	Cpp_Type?(typ')
{-2}	const?(typ')
{-3}	non_bool_integral_enum?(typ(typ'))
{-4}	volatile?(typ(typ'))
{1}	range(typ(typ')) = range(cv_base(typ(typ')))

Keeping (-3 -4) and hiding *,
 Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of `range_cv_base.2.2`.

`range_cv_base.3`:

{-1}	Cpp_Type?(typ')
{-2}	const?(typ')
{-3}	volatile?(typ(typ'))
{-4}	non_bool_integral_enum?(typ(typ(typ')))
{1}	range(typ') = range(cv_base(typ'))

Expanding the definition of `cv_base`,
 Expanding the definition of `cv_base`,
 Expanding the definition of `range`,
 Expanding the definition of `range`,

Expanding the definition of `cv_base`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
we get 2 subgoals:

`range_cv_base.3.1:`

{-1}	<code>Cpp_Type?(typ')</code>
{-2}	<code>const?(typ')</code>
{-3}	<code>volatile?(typ(typ'))</code>
{-4}	<code>non_bool_integral_enum?(typ(typ(typ')))</code>
{-5}	<code>const?(typ(typ(typ')))</code>
{1}	
<code>range(typ(typ(typ'))) = range(cv_base(typ(typ(typ))))</code>	

Keeping (-4 -5) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `range_cv_base.3.1`.

`range_cv_base.3.2:`

{-1}	<code>Cpp_Type?(typ')</code>
{-2}	<code>const?(typ')</code>
{-3}	<code>volatile?(typ(typ'))</code>
{-4}	<code>non_bool_integral_enum?(typ(typ(typ')))</code>
{-5}	<code>volatile?(typ(typ(typ')))</code>
{1}	
<code>range(typ(typ(typ'))) = range(cv_base(typ(typ(typ))))</code>	

Keeping (-4 -5) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `range_cv_base.3.2`.

`range_cv_base.4:`

{-1}	<code>Cpp_Type?(typ')</code>
{-2}	<code>non_bool_integral_enum?(typ(typ'))</code>
{-3}	<code>volatile?(typ')</code>
{1}	
<code>range(typ') = range(cv_base(typ'))</code>	

Expanding the definition of `cv_base`,

Expanding the definition of `cv_base`,

Expanding the definition of `range`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 2 subgoals:

`range_cv_base.4.1:`

{-1}	<code>Cpp_Type?(typ')</code>
{-2}	<code>non_bool_integral_enum?(typ(typ'))</code>
{-3}	<code>volatile?(typ')</code>
{-4}	<code>const?(typ(typ'))</code>
{1}	
<code>range(typ(typ')) = range(cv_base(typ(typ)))</code>	

Keeping (-2 -4) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `range_cv_base.4.1`.

range_cv_base.4.2:

{-1}	Cpp_Type?(typ')
{-2}	non_bool_integral_enum?(typ(typ'))
{-3}	volatile?(typ')
{-4}	volatile?(typ(typ'))
{1}	range(typ(typ')) = range(cv_base(typ(typ(typ'))))

Keeping (-2 -4) and hiding *,
 Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of range_cv_base.4.2.
 Q.E.D.

C.44.28 Cpp_Types.dt_non_bool_integral_enum_TCC1

Terse proof for dt_non_bool_integral_enum_TCC1.

dt_non_bool_integral_enum_TCC1:

{1}	\forall (typ: Cpp_Subtype(non_bool_integral_enum?), n: string, t: Cpp_Type_, m: pos-nat, c: [below[m] \rightarrow int]): typ = enum(n, t, m, c) \supset enum?(typ)
-----	--

Repeatedly Skolemizing and flattening,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 This completes the proof of dt_non_bool_integral_enum_TCC1.
 Q.E.D.

C.44.29 Cpp_Types.dt_non_bool_integral_enum_TCC2

Terse proof for dt_non_bool_integral_enum_TCC2.

dt_non_bool_integral_enum_TCC2:

{1}	\forall (typ: Cpp_Subtype(non_bool_integral_enum?), n: string, t: Cpp_Type_, m: pos-nat, c: [below[m] \rightarrow int]): typ = enum(n, t, m, c) \supset (\forall (x: int): range_enum(typ)(x) \equiv range(typ)(x)) \wedge (\forall (x: int): range_enum(typ)(x) \equiv range(typ)(x)) \wedge (\forall (x ₁ : [list[Byte], Address]): every[(range_enum(typ))](range(typ))(dt_enum(typ)'from_byte(x ₁))) \wedge interpreted_data_type?(range(typ)](dt_enum(typ))
-----	---

Repeatedly Skolemizing and flattening,
 Replacing using formula -4,
 Expanding the definition of range,
 Adding type constraints for dt_enum(enum(n!1, t!1, m!1, c!1)),
 we get 2 subgoals:

dt_non_bool_integral_enum_TCC2.1:

{-1}	pod_data_type?[((range_enum(enum(n', t', m', c')))) (dt_enum(enum(n', t', m', c')))]
{-2}	Cpp_Type?(enum(n', t', m', c'))
{-3}	non_bool_integral_enum?(enum(n', t', m', c'))
{-4}	m' > 0
{-5}	typ' = enum(n', t', m', c')
{1}	($\forall (x_1: \text{list}[\text{Byte}], \text{Address}])$: every[((range_enum(typ'))] (range_enum(enum(n', t', m', c')) (dt_enum(enum(n', t', m', c'))'from_byte(x_1)) \wedge interpreted_data_type?[((range(typ')))](dt_enum(enum(n', t', m', c'))))

Applying propositional simplification,

we get 2 subgoals:

dt_non_bool_integral_enum_TCC2.1.1:

{-1}	pod_data_type?[((range_enum(enum(n', t', m', c')))) (dt_enum(enum(n', t', m', c')))]
{-2}	Cpp_Type?(enum(n', t', m', c'))
{-3}	non_bool_integral_enum?(enum(n', t', m', c'))
{-4}	m' > 0
{-5}	typ' = enum(n', t', m', c')
{1}	$\forall (x_1: \text{list}[\text{Byte}], \text{Address}])$: every[((range_enum(typ'))] (range_enum(enum(n', t', m', c')) (dt_enum(enum(n', t', m', c'))'from_byte(x_1))

Repeatedly Skolemizing and flattening,

Expanding the definition of every,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of dt_non_bool_integral_enum_TCC2.1.1.

dt_non_bool_integral_enum_TCC2.1.2:

{-1}	pod_data_type?[((range_enum(enum(n', t', m', c')))) (dt_enum(enum(n', t', m', c')))]
{-2}	Cpp_Type?(enum(n', t', m', c'))
{-3}	non_bool_integral_enum?(enum(n', t', m', c'))
{-4}	m' > 0
{-5}	typ' = enum(n', t', m', c')
{1}	interpreted_data_type?[((range(typ')))](dt_enum(enum(n', t', m', c')))

Expanding the definition of pod_data_type?,

Applying disjunctive simplification to flatten sequent,

Keeping (-1 -7 1) and hiding *,

Expanding the definition of interpreted_data_type?,

Applying propositional simplification,

we get 2 subgoals:

dt_non_bool_integral_enum_TCC2.1.2.1:

{-1}	uninterpreted_data_type?(dt_enum(enum(n' , t' , m' , c'))'uidt)
{-2}	$\forall (d: ((\text{range_enum}(\text{enum}(n', t', m', c')))), a: \text{Address}):$ $\text{valid?}(\text{uidt}(\text{dt_enum}(\text{enum}(n', t', m', c'))))$ $(\text{to_byte}(\text{dt_enum}(\text{enum}(n', t', m', c')))(d, a), a)$
{-3}	$\forall (l: \text{list}[\text{Byte}], a: \text{Address}):$ $\text{valid?}(\text{uidt}(\text{dt_enum}(\text{enum}(n', t', m', c'))))(l, a) \equiv$ $\text{up?}(\text{from_byte}(\text{dt_enum}(\text{enum}(n', t', m', c')))(l, a))$
{-4}	$\forall (d: ((\text{range_enum}(\text{enum}(n', t', m', c')))), a: \text{Address}):$ $\text{down}(\text{from_byte}(\text{dt_enum}(\text{enum}(n', t', m', c'))))$ $(\text{to_byte}(\text{dt_enum}(\text{enum}(n', t', m', c')))(d, a), a))$ $= d$
{-5}	$\text{typ}' = \text{enum}(n', t', m', c')$
{1}	$\forall (d: ((\text{range}(\text{typ}'))), a: \text{Address}):$ $\text{valid?}(\text{uidt}(\text{dt_enum}(\text{enum}(n', t', m', c'))))$ $(\text{to_byte}(\text{dt_enum}(\text{enum}(n', t', m', c')))(d, a), a)$

Repeatedly Skolemizing and flattening,

Instantiating quantified variables,

Expanding the definition of range,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of dt_non_bool_integral_enum_TCC2.1.2.1.

dt_non_bool_integral_enum_TCC2.1.2.2:

{-1}	uninterpreted_data_type?(dt_enum(enum(n' , t' , m' , c'))'uidt)
{-2}	$\forall (d: ((\text{range_enum}(\text{enum}(n', t', m', c')))), a: \text{Address}):$ $\text{valid?}(\text{uidt}(\text{dt_enum}(\text{enum}(n', t', m', c'))))$ $(\text{to_byte}(\text{dt_enum}(\text{enum}(n', t', m', c')))(d, a), a)$
{-3}	$\forall (l: \text{list}[\text{Byte}], a: \text{Address}):$ $\text{valid?}(\text{uidt}(\text{dt_enum}(\text{enum}(n', t', m', c'))))(l, a) \equiv$ $\text{up?}(\text{from_byte}(\text{dt_enum}(\text{enum}(n', t', m', c')))(l, a))$
{-4}	$\forall (d: ((\text{range_enum}(\text{enum}(n', t', m', c')))), a: \text{Address}):$ $\text{down}(\text{from_byte}(\text{dt_enum}(\text{enum}(n', t', m', c'))))$ $(\text{to_byte}(\text{dt_enum}(\text{enum}(n', t', m', c')))(d, a), a))$ $= d$
{-5}	$\text{typ}' = \text{enum}(n', t', m', c')$
{1}	$\forall (d: ((\text{range}(\text{typ}'))), a: \text{Address}):$ $\text{down}(\text{from_byte}(\text{dt_enum}(\text{enum}(n', t', m', c'))))$ $(\text{to_byte}(\text{dt_enum}(\text{enum}(n', t', m', c')))(d, a), a))$ $= d$

Repeatedly Skolemizing and flattening,

Instantiating quantified variables,

Expanding the definition of range,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of dt_non_bool_integral_enum_TCC2.1.2.2.

dt_non_bool_integral_enum_TCC2.2:

{-1}	Cpp_Type?(enum(n' , t' , m' , c'))
{-2}	non_bool_integral_enum?(enum(n' , t' , m' , c'))
{-3}	$m' > 0$
{-4}	$typ' = \text{enum}(n', t', m', c')$
{1}	Cpp_Type?(enum(n' , t' , m' , c'))
{2}	$(\forall (x_1: [\text{list}[\text{Byte}], \text{Address}]):$ every[[$(\text{range_enum}(typ'))$] $(\text{range_enum}(\text{enum}(n', t', m', c'))$ $(\text{dt_enum}(\text{enum}(n', t', m', c')) \text{from_byte}(x_1)))$ $\wedge \text{interpreted_data_type?}[(\text{range}(typ'))](\text{dt_enum}(\text{enum}(n', t', m', c')))]$

which is trivially true.

This completes the proof of dt_non_bool_integral_enum_TCC2.2.

Q.E.D.

C.44.30 Cpp_Types.dt_non_bool_integral_enum_TCC3

Terse proof for dt_non_bool_integral_enum_TCC3.

dt_non_bool_integral_enum_TCC3:

{1}	$\forall (typ: \text{Cpp_Subtype}(\text{non_bool_integral_enum?})): \neg \text{enum?}(typ) \supset \text{non_bool_integral?}(typ)$
-----	--

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of dt_non_bool_integral_enum_TCC3.

Q.E.D.

C.44.31 Cpp_Types.dt_non_bool_integral_enum_TCC4

Terse proof for dt_non_bool_integral_enum_TCC4.

dt_non_bool_integral_enum_TCC4:

{1}	$\forall (typ: \text{Cpp_Subtype}(\text{non_bool_integral_enum?})): $ $\neg \text{enum?}(typ) \supset $ $(\forall (x: \text{int}): \text{range_integral}(typ)(x) \equiv \text{range}(typ)(x)) \wedge $ $(\forall (x: \text{int}): \text{range_integral}(typ)(x) \equiv \text{range}(typ)(x)) \wedge $ $(\forall (x_1: [\text{list}[\text{Byte}], \text{Address}]): $ every[[$(\text{range_integral}(typ))$] $(\text{range}(typ))(\text{dt_integral}(typ) \text{from_byte}(x_1))$ $\wedge \text{interpreted_data_type?}[(\text{range}(typ))](\text{dt_integral}(typ))$
-----	--

Repeatedly Skolemizing and flattening,

Expanding the definition of non_bool_integral_enum?,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 3 subgoals:

dt_non_bool_integral_enum_TCC4.1:

{-1}	Cpp_Type?(typ')
{-2}	non_bool_integral?(typ')
{1}	enum?(typ')
{2}	interpreted_data_type?[(range(typ'))](dt_integral(typ'))

C Proof scripts

Adding type constraints for `dt_integral(typ!1)`,
Using lemma `pod_is_interpreted_data` $[(\text{range_integral}(\text{typ}'))]$,
Expanding the definition of `interpreted_data_type?`,
Keeping (-1 -4 2) and hiding *,
Applying propositional simplification,
we get 2 subgoals:

`dt_non_bool_integral_enum_TCC4.1.1:`

{-1}	<code>uninterpreted_data_type?(dt_integral(typ)'uidt)</code>
{-2}	$\forall (d: ((\text{range_integral}(\text{typ}'))), a: \text{Address}):$ <code>valid?(uidt(dt_integral(typ')))(to_byte(dt_integral(typ'))(d, a), a)</code>
{-3}	$\forall (l: \text{list}[\text{Byte}], a: \text{Address}):$ <code>valid?(uidt(dt_integral(typ')))(l, a) \equiv</code> <code>up?(from_byte(dt_integral(typ'))(l, a))</code>
{-4}	$\forall (d: ((\text{range_integral}(\text{typ}'))), a: \text{Address}):$ <code>down(from_byte(dt_integral(typ')))(to_byte(dt_integral(typ'))(d, a), a) = d</code>
{-5}	<code>non_bool_integral?(typ')</code>
{1}	$\forall (d: ((\text{range}(\text{typ}'))), a: \text{Address}):$ <code>valid?(uidt(dt_integral(typ')))(to_byte(dt_integral(typ'))(d, a), a)</code>

Repeatedly Skolemizing and flattening,
Instantiating quantified variables,
Keeping (-1 -5 1) and hiding *,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `dt_non_bool_integral_enum_TCC4.1.1`.
`dt_non_bool_integral_enum_TCC4.1.2:`

{-1}	<code>uninterpreted_data_type?(dt_integral(typ)'uidt)</code>
{-2}	$\forall (d: ((\text{range_integral}(\text{typ}'))), a: \text{Address}):$ <code>valid?(uidt(dt_integral(typ')))(to_byte(dt_integral(typ'))(d, a), a)</code>
{-3}	$\forall (l: \text{list}[\text{Byte}], a: \text{Address}):$ <code>valid?(uidt(dt_integral(typ')))(l, a) \equiv</code> <code>up?(from_byte(dt_integral(typ'))(l, a))</code>
{-4}	$\forall (d: ((\text{range_integral}(\text{typ}'))), a: \text{Address}):$ <code>down(from_byte(dt_integral(typ')))(to_byte(dt_integral(typ'))(d, a), a) = d</code>
{-5}	<code>non_bool_integral?(typ')</code>
{1}	$\forall (d: ((\text{range}(\text{typ}'))), a: \text{Address}):$ <code>down(from_byte(dt_integral(typ')))(to_byte(dt_integral(typ'))(d, a), a) = d</code>

Repeatedly Skolemizing and flattening,
Instantiating quantified variables,
Keeping (-1 -5 1) and hiding *,
Expanding the definition of `range`,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `dt_non_bool_integral_enum_TCC4.1.2`.
`dt_non_bool_integral_enum_TCC4.2:`

{-1}	<code>Cpp_Type?(typ')</code>
{-2}	<code>non_bool_integral?(typ')</code>
{1}	<code>enum?(typ')</code>
{2}	$\forall (x_1: [\text{list}[\text{Byte}], \text{Address}]):$ <code>every</code> $[(\text{range_integral}(\text{typ}'))]$ <code>((range(typ'))(dt_integral(typ)'from_byte(x₁))</code>

Repeatedly Skolemizing and flattening,
Expanding the definition of `every`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 Expanding the definition of range,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 we get 2 subgoals:

dt_non_bool_integral_enum_TCC4.2.1:

{-1}	Cpp_Type?(typ')
{-2}	non_bool_integral?(typ')
{-3}	const?(typ')
{1}	bottom?(dt_integral(typ)'from_byte(x ₁))
{2}	range(typ(typ'))(down(dt_integral(typ)'from_byte(x ₁)))

Keeping (-2 -3) and hiding *,
 Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of dt_non_bool_integral_enum_TCC4.2.1.
 dt_non_bool_integral_enum_TCC4.2.2:

{-1}	Cpp_Type?(typ')
{-2}	non_bool_integral?(typ')
{-3}	volatile?(typ')
{1}	bottom?(dt_integral(typ)'from_byte(x ₁))
{2}	range(typ(typ'))(down(dt_integral(typ)'from_byte(x ₁)))

Keeping (-2 -3) and hiding *,
 Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of dt_non_bool_integral_enum_TCC4.2.2.
 dt_non_bool_integral_enum_TCC4.3:

{-1}	Cpp_Type?(typ')
{-2}	non_bool_integral?(typ')
{1}	enum?(typ')
{2}	$\forall (x: \text{int}): \text{range_integral}(typ')(x) \equiv \text{range}(typ')(x)$

Repeatedly Skolemizing and flattening,
 Expanding the definition of range,
 Keeping (-3 2) and hiding *,
 Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of dt_non_bool_integral_enum_TCC4.3.
 Q.E.D.

C.44.32 Cpp_Types.dt_non_bool_integral_enum_pod

Terse proof for dt_non_bool_integral_enum_pod.

dt_non_bool_integral_enum_pod:

{1}	$\forall (typ: \text{Cpp_Subtype}(\text{non_bool_integral_enum?})): \text{pod_data_type?}[(\text{range}(typ))](\text{dt_non_bool_integral_enum}(typ))$

Repeatedly Skolemizing and flattening,
 Expanding the definition of dt_non_bool_integral_enum,
 Expanding the definition of non_bool_integral_enum?,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 we get 3 subgoals:

C Proof scripts

dt_non_bool_integral_enum_pod.1:

{-1}	Cpp_Type?(typ')
{-2}	non_bool_integral?(typ')
{-3}	enum?(typ')
{1}	pod_data_type?[((range(typ')))](dt_enum(typ'))

Keeping (-2 -3) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of dt_non_bool_integral_enum_pod.1.

dt_non_bool_integral_enum_pod.2:

{-1}	Cpp_Type?(typ')
{-2}	non_bool_integral?(typ')
{1}	enum?(typ')
{2}	pod_data_type?[((range(typ')))](dt_integral(typ'))

Adding type constraints for dt_integral(typ!1),

Expanding the definition of pod_data_type?,

Expanding the definition of interpreted_data_type?,

Applying propositional simplification,

we get 2 subgoals:

dt_non_bool_integral_enum_pod.2.1:

{-1}	uninterpreted_data_type?(dt_integral(typ')' uidt)
{-2}	$\forall (d: ((range_integral(typ'))), a: Address):$ valid?(uidt(dt_integral(typ')))(to_byte(dt_integral(typ'))(d, a), a)
{-3}	$\forall (l: list[Byte], a: Address):$ valid?(uidt(dt_integral(typ')))(l, a) \equiv up?(from_byte(dt_integral(typ'))(l, a))
{-4}	$\forall (d: ((range_integral(typ'))), a: Address):$ down(from_byte(dt_integral(typ'))(to_byte(dt_integral(typ'))(d, a), a)) = d
{-5}	$\forall (l: list[Byte], a_1, a_2: Address):$ valid?(uidt(dt_integral(typ')))(l, a_1) \equiv valid?(uidt(dt_integral(typ')))(l, a_2)
{-6}	size(uidt(dt_integral(typ'))) > 0
{-7}	Cpp_Type?(typ')
{-8}	non_bool_integral?(typ')
{1}	$\forall (d: ((range(typ'))), a: Address):$ valid?(uidt(dt_integral(typ')))(to_byte(dt_integral(typ'))(d, a), a)
{2}	enum?(typ')

Repeatedly Skolemizing and flattening,

Instantiating quantified variables,

Expanding the definition of range,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 2 subgoals:

dt_non_bool_integral_enum_pod.2.1.1:

{-1}	const?(typ')
{-2}	range(typ(typ'))(d')
{-3}	uninterpreted_data_type?(dt_integral(typ') uidt)
{-4}	$\forall (l: \text{list}[\text{Byte}], a: \text{Address}):$ valid?(uidt(dt_integral(typ')))(l, a) \equiv up?(from_byte(dt_integral(typ')))(l, a)
{-5}	$\forall (d: ((\text{range_integral}(\text{typ}'))), a: \text{Address}):$ down(from_byte(dt_integral(typ')))(to_byte(dt_integral(typ'))(d, a), a) = d
{-6}	$\forall (l: \text{list}[\text{Byte}], a_1, a_2: \text{Address}):$ valid?(uidt(dt_integral(typ')))(l, a_1) \equiv valid?(uidt(dt_integral(typ')))(l, a_2)
{-7}	size(uidt(dt_integral(typ'))) > 0
{-8}	Cpp_Type?(typ')
{-9}	non_bool_integral?(typ')
{1}	range_integral(typ')(d')
{2}	valid?(uidt(dt_integral(typ')))(to_byte(dt_integral(typ'))(d', a'), a')

Keeping (-1 -9) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of dt_non_bool_integral_enum_pod.2.1.1.

dt_non_bool_integral_enum_pod.2.1.2:

{-1}	range(typ(typ'))(d')
{-2}	volatile?(typ')
{-3}	uninterpreted_data_type?(dt_integral(typ') uidt)
{-4}	$\forall (l: \text{list}[\text{Byte}], a: \text{Address}):$ valid?(uidt(dt_integral(typ')))(l, a) \equiv up?(from_byte(dt_integral(typ')))(l, a)
{-5}	$\forall (d: ((\text{range_integral}(\text{typ}'))), a: \text{Address}):$ down(from_byte(dt_integral(typ')))(to_byte(dt_integral(typ'))(d, a), a) = d
{-6}	$\forall (l: \text{list}[\text{Byte}], a_1, a_2: \text{Address}):$ valid?(uidt(dt_integral(typ')))(l, a_1) \equiv valid?(uidt(dt_integral(typ')))(l, a_2)
{-7}	size(uidt(dt_integral(typ'))) > 0
{-8}	Cpp_Type?(typ')
{-9}	non_bool_integral?(typ')
{1}	range_integral(typ')(d')
{2}	valid?(uidt(dt_integral(typ')))(to_byte(dt_integral(typ'))(d', a'), a')

Keeping (-2 -9) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of dt_non_bool_integral_enum_pod.2.1.2.

dt_non_bool_integral_enum_pod.2.2:

{-1}	uninterpreted_data_type?(dt_integral(typ') 'uidt)
{-2}	$\forall (d: ((\text{range_integral}(\text{typ}')), a: \text{Address}):$ valid?(uidt(dt_integral(typ')))(to_byte(dt_integral(typ'))(d, a), a)
{-3}	$\forall (l: \text{list}[\text{Byte}], a: \text{Address}):$ valid?(uidt(dt_integral(typ')))(l, a) \equiv up?(from_byte(dt_integral(typ')))(l, a)
{-4}	$\forall (d: ((\text{range_integral}(\text{typ}')), a: \text{Address}):$ down(from_byte(dt_integral(typ')))(to_byte(dt_integral(typ'))(d, a), a) = d
{-5}	$\forall (l: \text{list}[\text{Byte}], a_1, a_2: \text{Address}):$ valid?(uidt(dt_integral(typ')))(l, a_1) \equiv valid?(uidt(dt_integral(typ')))(l, a_2)
{-6}	size(uidt(dt_integral(typ'))) > 0
{-7}	Cpp_Type?(typ')
{-8}	non_bool_integral?(typ')
{1}	$\forall (d: ((\text{range}(\text{typ}')), a: \text{Address}):$ down(from_byte(dt_integral(typ')))(to_byte(dt_integral(typ'))(d, a), a) = d
{2}	enum?(typ')

Repeatedly Skolemizing and flattening,

Instantiating quantified variables,

Expanding the definition of range,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 2 subgoals:

dt_non_bool_integral_enum_pod.2.2.1:

{-1}	const?(typ')
{-2}	range(typ(typ'))(d')
{-3}	uninterpreted_data_type?(dt_integral(typ') 'uidt)
{-4}	$\forall (d: ((\text{range_integral}(\text{typ}')), a: \text{Address}):$ valid?(uidt(dt_integral(typ')))(to_byte(dt_integral(typ'))(d, a), a)
{-5}	$\forall (l: \text{list}[\text{Byte}], a: \text{Address}):$ valid?(uidt(dt_integral(typ')))(l, a) \equiv up?(from_byte(dt_integral(typ')))(l, a)
{-6}	$\forall (l: \text{list}[\text{Byte}], a_1, a_2: \text{Address}):$ valid?(uidt(dt_integral(typ')))(l, a_1) \equiv valid?(uidt(dt_integral(typ')))(l, a_2)
{-7}	size(uidt(dt_integral(typ'))) > 0
{-8}	Cpp_Type?(typ')
{-9}	non_bool_integral?(typ')
{1}	range_integral(typ')(d')
{2}	down(from_byte(dt_integral(typ')))(to_byte(dt_integral(typ'))(d', a'), a') = d'

Keeping (-1 -9) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of dt_non_bool_integral_enum_pod.2.2.1.

dt_non_bool_integral_enum_pod.2.2.2:

{-1}	range(typ(typ'))(d')
{-2}	volatile?(typ')
{-3}	uninterpreted_data_type?(dt_integral(typ')'uidt)
{-4}	$\forall (d: ((\text{range_integral}(\text{typ}'))), a: \text{Address}):$ $\text{valid?}(\text{uidt}(\text{dt_integral}(\text{typ}')))(\text{to_byte}(\text{dt_integral}(\text{typ}'))(d, a), a)$
{-5}	$\forall (l: \text{list}[\text{Byte}], a: \text{Address}):$ $\text{valid?}(\text{uidt}(\text{dt_integral}(\text{typ}')))(l, a) \equiv$ $\text{up?}(\text{from_byte}(\text{dt_integral}(\text{typ}')))(l, a)$
{-6}	$\forall (l: \text{list}[\text{Byte}], a_1, a_2: \text{Address}):$ $\text{valid?}(\text{uidt}(\text{dt_integral}(\text{typ}')))(l, a_1) \equiv$ $\text{valid?}(\text{uidt}(\text{dt_integral}(\text{typ}')))(l, a_2)$
{-7}	$\text{size}(\text{uidt}(\text{dt_integral}(\text{typ}'))) > 0$
{-8}	Cpp_Type?(typ')
{-9}	non_bool_integral?(typ')
{1}	range_integral(typ')(d')
{2}	$\text{down}(\text{from_byte}(\text{dt_integral}(\text{typ}')))(\text{to_byte}(\text{dt_integral}(\text{typ}'))(d', a'), a') = d'$

Keeping (-2 -9) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of dt_non_bool_integral_enum_pod.2.2.2.

dt_non_bool_integral_enum_pod.3:

{-1}	Cpp_Type?(typ')
{-2}	enum?(typ')
{1}	$\text{pod_data_type?}[(\text{range}(\text{typ}'))](\text{dt_enum}(\text{typ}'))$

Adding type constraints for dt_enum(typ!1),

Expanding the definition of pod_data_type?,

Expanding the definition of interpreted_data_type?,

Applying propositional simplification,

we get 2 subgoals:

dt_non_bool_integral_enum_pod.3.1:

{-1}	uninterpreted_data_type?(dt_enum(typ')'uidt)
{-2}	$\forall (d: ((\text{range_enum}(\text{typ}'))), a: \text{Address}):$ $\text{valid?}(\text{uidt}(\text{dt_enum}(\text{typ}')))(\text{to_byte}(\text{dt_enum}(\text{typ}'))(d, a), a)$
{-3}	$\forall (l: \text{list}[\text{Byte}], a: \text{Address}):$ $\text{valid?}(\text{uidt}(\text{dt_enum}(\text{typ}')))(l, a) \equiv \text{up?}(\text{from_byte}(\text{dt_enum}(\text{typ}')))(l, a)$
{-4}	$\forall (d: ((\text{range_enum}(\text{typ}'))), a: \text{Address}):$ $\text{down}(\text{from_byte}(\text{dt_enum}(\text{typ}')))(\text{to_byte}(\text{dt_enum}(\text{typ}'))(d, a), a) = d$
{-5}	$\forall (l: \text{list}[\text{Byte}], a_1, a_2: \text{Address}):$ $\text{valid?}(\text{uidt}(\text{dt_enum}(\text{typ}')))(l, a_1) \equiv \text{valid?}(\text{uidt}(\text{dt_enum}(\text{typ}')))(l, a_2)$
{-6}	$\text{size}(\text{uidt}(\text{dt_enum}(\text{typ}'))) > 0$
{-7}	Cpp_Type?(typ')
{-8}	enum?(typ')
{1}	$\forall (d: ((\text{range}(\text{typ}'))), a: \text{Address}):$ $\text{valid?}(\text{uidt}(\text{dt_enum}(\text{typ}')))(\text{to_byte}(\text{dt_enum}(\text{typ}'))(d, a), a)$

Repeatedly Skolemizing and flattening,

Instantiating quantified variables,

Expanding the definition of range,

which is trivially true.

This completes the proof of dt_non_bool_integral_enum_pod.3.1.

dt_non_bool_integral_enum_pod.3.2:

{-1}	uninterpreted_data_type?(dt_enum(typ'))'uidt)
{-2}	$\forall (d: ((\text{range_enum}(\text{typ}'))), a: \text{Address}):$ valid?(uidt(dt_enum(typ')))(to_byte(dt_enum(typ'))(d, a), a)
{-3}	$\forall (l: \text{list}[\text{Byte}], a: \text{Address}):$ valid?(uidt(dt_enum(typ')))(l, a) \equiv up?(from_byte(dt_enum(typ'))(l, a))
{-4}	$\forall (d: ((\text{range_enum}(\text{typ}'))), a: \text{Address}):$ down(from_byte(dt_enum(typ'))(to_byte(dt_enum(typ'))(d, a), a)) = d
{-5}	$\forall (l: \text{list}[\text{Byte}], a_1, a_2: \text{Address}):$ valid?(uidt(dt_enum(typ')))(l, a_1) \equiv valid?(uidt(dt_enum(typ')))(l, a_2)
{-6}	size(uidt(dt_enum(typ')))) > 0
{-7}	Cpp_Type?(typ')
{-8}	enum?(typ')
{1}	
	$\forall (d: ((\text{range}(\text{typ}'))), a: \text{Address}):$ down(from_byte(dt_enum(typ'))(to_byte(dt_enum(typ'))(d, a), a)) = d

Repeatedly Skolemizing and flattening,
Instantiating quantified variables,
Expanding the definition of range,
which is trivially true.

This completes the proof of dt_non_bool_integral_enum_pod.3.2.

Q.E.D.

C.44.33 Cpp_Types.dt_TCC1

Terse proof for dt_TCC1.

dt_TCC1:

{1}	$\forall (\text{typ}: \text{Cpp_Subtype}(\text{cv}(\text{non_bool_integral_enum?})), t: \text{Cpp_Type_}):$ typ = const(t) \supset ($\exists (x_1: (\text{interpreted_data_type?}[\text{((range(t))]}))): \text{TRUE}$)
-----	--

Repeatedly Skolemizing and flattening,
Expanding the definition of cv,
Expanding the definition of v,
Expanding the definition of c,
Replacing using formula -3,
Simplifying, rewriting, and recording with decision procedures,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
we get 3 subgoals:

dt_TCC1.1:

{-1}	Cpp_Type?(const(t'))
{-2}	non_bool_integral_enum?(const(t'))
{-3}	typ' = const(t')
{1}	
	$\exists (x_1: (\text{interpreted_data_type?}[\text{((range(t'))]}))): \text{TRUE}$

Keeping (-2) and hiding *,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of dt_TCC1.1.

dt_TCC1.2:

{-1} Cpp_Type?(const(t')) {-2} non_bool_integral_enum?(t') {-3} $\text{typ}' = \text{const}(t')$	<hr style="border: 0.5px solid black;"/> {1} $\exists (x_1: (\text{interpreted_data_type?}[(\text{range}(t'))])): \text{TRUE}$
--	--

Expanding the definition of non_bool_integral_enum?,

Applying propositional simplification,

we get 2 subgoals:

dt_TCC1.2.1:

{-1} non_bool_integral?(t') {-2} Cpp_Type?(const(t')) {-3} $\text{typ}' = \text{const}(t')$	<hr style="border: 0.5px solid black;"/> {1} $\exists (x_1: (\text{interpreted_data_type?}[(\text{range}(t'))])): \text{TRUE}$
---	--

Instantiating the top quantifier in 1 with the terms: (dt_integral($t!1$)),

we get 3 subgoals:

dt_TCC1.2.1.1:

{-1} non_bool_integral?(t') {-2} Cpp_Type?(const(t')) {-3} $\text{typ}' = \text{const}(t')$	<hr style="border: 0.5px solid black;"/> {1} TRUE
---	---

which is trivially true.

This completes the proof of dt_TCC1.2.1.1.

dt_TCC1.2.1.2:

{-1} non_bool_integral?(t') {-2} Cpp_Type?(const(t')) {-3} $\text{typ}' = \text{const}(t')$	<hr style="border: 0.5px solid black;"/> {1} $(\forall (x: \text{int}): \text{range_integral}(t')(x) \equiv \text{range}(t')(x)) \wedge$ $(\forall (x: \text{int}): \text{range_integral}(t')(x) \equiv \text{range}(t')(x)) \wedge$ $(\forall (x_1: [\text{list}[\text{Byte}], \text{Address}]):$ $\quad \text{every}[(\text{range_integral}(t'))]$ $\quad (\text{range}(t'))(\text{dt_integral}(t)' \text{from_byte}(x_1)))$ $\wedge \text{interpreted_data_type?}[(\text{range}(t'))](\text{dt_integral}(t'))$
---	--

Adding type constraints for dt_integral($t!1$),

we get 2 subgoals:

dt_TCC1.2.1.2.1:

{-1} pod_data_type?[(range_integral(t'))](dt_integral(t')) {-2} non_bool_integral?(t') {-3} Cpp_Type?(const(t')) {-4} $\text{typ}' = \text{const}(t')$	<hr style="border: 0.5px solid black;"/> {1} $(\forall (x: \text{int}): \text{range_integral}(t')(x) \equiv \text{range}(t')(x)) \wedge$ $(\forall (x: \text{int}): \text{range_integral}(t')(x) \equiv \text{range}(t')(x)) \wedge$ $(\forall (x_1: [\text{list}[\text{Byte}], \text{Address}]):$ $\quad \text{every}[(\text{range_integral}(t'))]$ $\quad (\text{range}(t'))(\text{dt_integral}(t)' \text{from_byte}(x_1)))$ $\wedge \text{interpreted_data_type?}[(\text{range}(t'))](\text{dt_integral}(t'))$
---	--

Applying propositional simplification,

we get 4 subgoals:

C Proof scripts

dt_TCC1.2.1.2.1.1:

{-1}	pod_data_type?[(range_integral(t'))](dt_integral(t'))
{-2}	non_bool_integral?(t')
{-3}	Cpp_Type?(const(t'))
{-4}	typ' = const(t')
{1}	$\forall (x: \text{int}): \text{range_integral}(t')(x) \equiv \text{range}(t')(x)$

Repeatedly Skolemizing and flattening,
 Expanding the definition of range,
 Keeping (-3 1) and hiding *,
 Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of dt_TCC1.2.1.2.1.1.

dt_TCC1.2.1.2.1.2:

{-1}	pod_data_type?[(range_integral(t'))](dt_integral(t'))
{-2}	non_bool_integral?(t')
{-3}	Cpp_Type?(const(t'))
{-4}	typ' = const(t')
{1}	$\forall (x: \text{int}): \text{range_integral}(t')(x) \equiv \text{range}(t')(x)$

Keeping (-2 1) and hiding *,
 Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of dt_TCC1.2.1.2.1.2.

dt_TCC1.2.1.2.1.3:

{-1}	pod_data_type?[(range_integral(t'))](dt_integral(t'))
{-2}	non_bool_integral?(t')
{-3}	Cpp_Type?(const(t'))
{-4}	typ' = const(t')
{1}	$\forall (x_1: [\text{list}[\text{Byte}], \text{Address}]):$ $\text{every}[(\text{range_integral}(t'))](\text{range}(t'))(\text{dt_integral}(t)' \text{from_byte}(x_1))$

Repeatedly Skolemizing and flattening,
 Expanding the definition of every,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 Expanding the definition of range,
 Keeping (-2 2) and hiding *,
 Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of dt_TCC1.2.1.2.1.3.

dt_TCC1.2.1.2.1.4:

{-1}	pod_data_type?[(range_integral(t'))](dt_integral(t'))
{-2}	non_bool_integral?(t')
{-3}	Cpp_Type?(const(t'))
{-4}	typ' = const(t')
{1}	interpreted_data_type?[(range(t'))](dt_integral(t'))

Using lemma pod_is_interpreted_data[(range_integral(t'))],
 Keeping (-1 -3 1) and hiding *,
 Expanding the definition of interpreted_data_type?,
 Applying propositional simplification,
 we get 2 subgoals:

dt_TCC1.2.1.2.1.4.1:

{-1}	uninterpreted_data_type?(dt_integral(t)' uidt)
{-2}	$\forall (d: ((\text{range_integral}(t))), a: \text{Address}):$ valid?(uidt(dt_integral(t'))(to_byte(dt_integral(t'))(d, a), a)
{-3}	$\forall (l: \text{list}[\text{Byte}], a: \text{Address}):$ valid?(uidt(dt_integral(t'))(l, a) \equiv up?(from_byte(dt_integral(t'))(l, a))
{-4}	$\forall (d: ((\text{range_integral}(t))), a: \text{Address}):$ down(from_byte(dt_integral(t'))(to_byte(dt_integral(t'))(d, a), a)) = d
{-5}	non_bool_integral?(t')
{1}	$\forall (d: ((\text{range}(t))), a: \text{Address}):$ valid?(uidt(dt_integral(t'))(to_byte(dt_integral(t'))(d, a), a)

Repeatedly Skolemizing and flattening,

Instantiating quantified variables,

Expanding the definition of range,

Keeping (-1 -5 1) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of dt_TCC1.2.1.2.1.4.1.

dt_TCC1.2.1.2.1.4.2:

{-1}	uninterpreted_data_type?(dt_integral(t)' uidt)
{-2}	$\forall (d: ((\text{range_integral}(t))), a: \text{Address}):$ valid?(uidt(dt_integral(t'))(to_byte(dt_integral(t'))(d, a), a)
{-3}	$\forall (l: \text{list}[\text{Byte}], a: \text{Address}):$ valid?(uidt(dt_integral(t'))(l, a) \equiv up?(from_byte(dt_integral(t'))(l, a))
{-4}	$\forall (d: ((\text{range_integral}(t))), a: \text{Address}):$ down(from_byte(dt_integral(t'))(to_byte(dt_integral(t'))(d, a), a)) = d
{-5}	non_bool_integral?(t')
{1}	$\forall (d: ((\text{range}(t))), a: \text{Address}):$ down(from_byte(dt_integral(t'))(to_byte(dt_integral(t'))(d, a), a)) = d

Repeatedly Skolemizing and flattening,

Instantiating quantified variables,

Keeping (-1 -5 1) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of dt_TCC1.2.1.2.1.4.2.

dt_TCC1.2.1.2.2:

{-1}	non_bool_integral?(t')
{-2}	Cpp_Type?(const(t'))
{-3}	typ' = const(t')
{1}	non_bool_integral?(t')
{2}	$(\forall (x: \text{int}): \text{range_integral}(t')(x) \equiv \text{range}(t')(x)) \wedge$ $(\forall (x: \text{int}): \text{range_integral}(t')(x) \equiv \text{range}(t')(x)) \wedge$ $(\forall (x_1: [\text{list}[\text{Byte}], \text{Address}]):$ every[[range_integral(t')]] ((range(t'))(dt_integral(t)' from_byte(x ₁))) $\wedge \text{interpreted_data_type?}[[\text{range}(t')]](\text{dt_integral}(t'))$

which is trivially true.

This completes the proof of dt_TCC1.2.1.2.2.

C Proof scripts

dt_TCC1.2.1.3:

{-1}	non_bool_integral?(t')
{-2}	Cpp_Type?(const(t'))
{-3}	typ' = const(t')
{1}	non_bool_integral?(t')

which is trivially true.

This completes the proof of dt_TCC1.2.1.3.

dt_TCC1.2.2:

{-1}	enum?(t')
{-2}	Cpp_Type?(const(t'))
{-3}	typ' = const(t')
{1}	$\exists (x_1: (\text{interpreted_data_type?}[(\text{range}(t'))])): \text{TRUE}$

Instantiating the top quantifier in 1 with the terms: (dt_enum(t!1)),
we get 3 subgoals:

dt_TCC1.2.2.1:

{-1}	enum?(t')
{-2}	Cpp_Type?(const(t'))
{-3}	typ' = const(t')
{1}	TRUE

which is trivially true.

This completes the proof of dt_TCC1.2.2.1.

dt_TCC1.2.2.2:

{-1}	enum?(t')
{-2}	Cpp_Type?(const(t'))
{-3}	typ' = const(t')
{1}	$(\forall (x: \text{int}): \text{range_enum}(t')(x) \equiv \text{range}(t')(x)) \wedge$ $(\forall (x: \text{int}): \text{range_enum}(t')(x) \equiv \text{range}(t')(x)) \wedge$ $(\forall (x_1: [\text{list}[\text{Byte}], \text{Address}]):$ $\text{every}[(\text{range_enum}(t'))](\text{range}(t'))(\text{dt_enum}(t)' \text{from_byte}(x_1)))$ $\wedge \text{interpreted_data_type?}[(\text{range}(t'))](\text{dt_enum}(t'))$

Expanding the definition of range,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 2 subgoals:

dt_TCC1.2.2.2.1:

{-1}	enum?(t')
{-2}	Cpp_Type?(const(t'))
{-3}	typ' = const(t')
{1}	interpreted_data_type?[(range(t'))](dt_enum(t'))

Adding type constraints for dt_enum(t!1),

Using lemma pod_is_interpreted_data[(range_enum(t'))],

Keeping (-1 -3 1) and hiding *,

Expanding the definition of interpreted_data_type?,

Applying propositional simplification,

we get 2 subgoals:

dt_TCC1.2.2.2.1.1:

{-1}	uninterpreted_data_type?(dt_enum(t)'uidt)
{-2}	$\forall (d: ((\text{range_enum}(t))), a: \text{Address}):$ valid?(uidt(dt_enum(t'))(to_byte(dt_enum(t'))(d, a), a)
{-3}	$\forall (l: \text{list}[\text{Byte}], a: \text{Address}):$ valid?(uidt(dt_enum(t'))(l, a) \equiv up?(from_byte(dt_enum(t'))(l, a))
{-4}	$\forall (d: ((\text{range_enum}(t))), a: \text{Address}):$ down(from_byte(dt_enum(t'))(to_byte(dt_enum(t'))(d, a), a)) = d
{-5}	enum?(t')
{1}	$\forall (d: ((\text{range}(t))), a: \text{Address}):$ valid?(uidt(dt_enum(t'))(to_byte(dt_enum(t'))(d, a), a)

Repeatedly Skolemizing and flattening,

Instantiating quantified variables,

Keeping (-1 -5 1) and hiding *,

Expanding the definition of range,

which is trivially true.

This completes the proof of dt_TCC1.2.2.2.1.1.

dt_TCC1.2.2.2.1.2:

{-1}	uninterpreted_data_type?(dt_enum(t)'uidt)
{-2}	$\forall (d: ((\text{range_enum}(t))), a: \text{Address}):$ valid?(uidt(dt_enum(t'))(to_byte(dt_enum(t'))(d, a), a)
{-3}	$\forall (l: \text{list}[\text{Byte}], a: \text{Address}):$ valid?(uidt(dt_enum(t'))(l, a) \equiv up?(from_byte(dt_enum(t'))(l, a))
{-4}	$\forall (d: ((\text{range_enum}(t))), a: \text{Address}):$ down(from_byte(dt_enum(t'))(to_byte(dt_enum(t'))(d, a), a)) = d
{-5}	enum?(t')
{1}	$\forall (d: ((\text{range}(t))), a: \text{Address}):$ down(from_byte(dt_enum(t'))(to_byte(dt_enum(t'))(d, a), a)) = d

Repeatedly Skolemizing and flattening,

Instantiating quantified variables,

Expanding the definition of range,

which is trivially true.

This completes the proof of dt_TCC1.2.2.2.1.2.

dt_TCC1.2.2.2.2:

{-1}	enum?(t')
{-2}	Cpp_Type?(const(t'))
{-3}	typ' = const(t')
{1}	$\forall (x_1: [\text{list}[\text{Byte}], \text{Address}]):$ every[$((\text{range_enum}(t')))](\text{range_enum}(t'))(\text{dt_enum}(t)' \text{from_byte}(x_1))$

Expanding the definition of every,

Repeatedly Skolemizing and flattening,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of dt_TCC1.2.2.2.2.

dt_TCC1.2.2.3:

{-1}	enum?(t')
{-2}	Cpp_Type?(const(t'))
{-3}	typ' = const(t')
{1}	Cpp_Type?(t') \wedge enum?(t')

Simplifying, rewriting, and recording with decision procedures,

C Proof scripts

Expanding the definition of Cpp_Type?,
 Repeatedly Skolemizing and flattening,
 Instantiating the top quantifier in -3 with the terms: (t!2),
 Expanding the definition of subterm,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 This completes the proof of dt_TCC1.2.2.3.

dt_TCC1.3:

{-1}	Cpp_Type?(const(t'))
{-2}	volatile?(t')
{-3}	non_bool_integral_enum?(typ(t'))
{-4}	typ' = const(t')
{1} $\exists (x_1: (\text{interpreted_data_type?}[(\text{range}(t'))])): \text{TRUE}$	

Expanding the definition of non_bool_integral_enum?,
 Applying propositional simplification,
 we get 2 subgoals:

dt_TCC1.3.1:

{-1}	non_bool_integral?(typ(t'))
{-2}	Cpp_Type?(const(t'))
{-3}	volatile?(t')
{-4}	typ' = const(t')
{1} $\exists (x_1: (\text{interpreted_data_type?}[(\text{range}(t'))])): \text{TRUE}$	

Instantiating the top quantifier in 1 with the terms: (dt_integral(typ(t!1))),
 we get 3 subgoals:

dt_TCC1.3.1.1:

{-1}	non_bool_integral?(typ(t'))
{-2}	Cpp_Type?(const(t'))
{-3}	volatile?(t')
{-4}	typ' = const(t')
{1} TRUE	

which is trivially true.

This completes the proof of dt_TCC1.3.1.1.

dt_TCC1.3.1.2:

{-1}	non_bool_integral?(typ(t'))
{-2}	Cpp_Type?(const(t'))
{-3}	volatile?(t')
{-4}	typ' = const(t')
{1} $(\forall (x: \text{int}): \text{range_integral}(\text{typ}(t'))(x) \equiv \text{range}(t')(x)) \wedge$ $(\forall (x: \text{int}): \text{range_integral}(\text{typ}(t'))(x) \equiv \text{range}(t')(x)) \wedge$ $(\forall (x_1: [\text{list}[\text{Byte}], \text{Address}]):$ $\quad \text{every}[(\text{range_integral}(\text{typ}(t')))]$ $\quad \quad ((\text{range}(t'))(\text{dt_integral}(\text{typ}(t')) \text{ 'from_byte}(x_1)))$ $\quad \wedge \text{interpreted_data_type?}[(\text{range}(t'))](\text{dt_integral}(\text{typ}(t'))))$	

Adding type constraints for dt_integral(typ(t!1)),
 we get 2 subgoals:

dt_TCC1.3.1.2.1:

{-1}	pod_data_type?[((range_integral(typ(t')))](dt_integral(typ(t')))
{-2}	non_bool_integral?(typ(t'))
{-3}	Cpp_Type?(const(t'))
{-4}	volatile?(t')
{-5}	typ' = const(t')
{1}	$(\forall (x: \text{int}): \text{range_integral}(\text{typ}(t'))(x) \equiv \text{range}(t')(x)) \wedge$ $(\forall (x: \text{int}): \text{range_integral}(\text{typ}(t'))(x) \equiv \text{range}(t')(x)) \wedge$ $(\forall (x_1: [\text{list}[\text{Byte}], \text{Address}]):$ $\quad \text{every}[\text{range_integral}(\text{typ}(t'))]$ $\quad \quad ((\text{range}(t'))(\text{dt_integral}(\text{typ}(t')) \text{'from_byte}(x_1)))$ $\quad \wedge \text{interpreted_data_type?}[\text{range}(t')](\text{dt_integral}(\text{typ}(t')))$

Applying propositional simplification,

we get 4 subgoals:

dt_TCC1.3.1.2.1.1:

{-1}	pod_data_type?[((range_integral(typ(t')))](dt_integral(typ(t')))
{-2}	non_bool_integral?(typ(t'))
{-3}	Cpp_Type?(const(t'))
{-4}	volatile?(t')
{-5}	typ' = const(t')
{1}	$\forall (x: \text{int}): \text{range_integral}(\text{typ}(t'))(x) \equiv \text{range}(t')(x)$

Repeatedly Skolemizing and flattening,

Expanding the definition of range,

Keeping (-3 1) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of dt_TCC1.3.1.2.1.1.

dt_TCC1.3.1.2.1.2:

{-1}	pod_data_type?[((range_integral(typ(t')))](dt_integral(typ(t')))
{-2}	non_bool_integral?(typ(t'))
{-3}	Cpp_Type?(const(t'))
{-4}	volatile?(t')
{-5}	typ' = const(t')
{1}	$\forall (x: \text{int}): \text{range_integral}(\text{typ}(t'))(x) \equiv \text{range}(t')(x)$

Keeping (-2 1) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of dt_TCC1.3.1.2.1.2.

dt_TCC1.3.1.2.1.3:

{-1}	pod_data_type?[((range_integral(typ(t')))](dt_integral(typ(t')))
{-2}	non_bool_integral?(typ(t'))
{-3}	Cpp_Type?(const(t'))
{-4}	volatile?(t')
{-5}	typ' = const(t')
{1}	$\forall (x_1: [\text{list}[\text{Byte}], \text{Address}]):$ $\quad \text{every}[\text{range_integral}(\text{typ}(t'))]$ $\quad \quad ((\text{range}(t'))(\text{dt_integral}(\text{typ}(t')) \text{'from_byte}(x_1)))$

Repeatedly Skolemizing and flattening,

Expanding the definition of every,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

C Proof scripts

Expanding the definition of range,

Keeping (-2 2) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `dt_TCC1.3.1.2.1.3`.

`dt_TCC1.3.1.2.1.4`:

{-1}	pod_data_type?[((range_integral(typ(t')))](dt_integral(typ(t')))
{-2}	non_bool_integral?(typ(t'))
{-3}	Cpp_Type?(const(t'))
{-4}	volatile?(t')
{-5}	typ' = const(t')
{1}	interpreted_data_type?[((range(t')))](dt_integral(typ(t')))

Using lemma `pod_is_interpreted_data[(range_integral(typ(t')))]`,

Keeping (-1 -3 1) and hiding *,

Expanding the definition of `interpreted_data_type?`,

Applying propositional simplification,

we get 2 subgoals:

`dt_TCC1.3.1.2.1.4.1`:

{-1}	<code>uninterpreted_data_type?(dt_integral(typ(t')) 'uidt)</code>
{-2}	$\forall (d: ((range_integral(typ(t')))), a: Address):$ <code>valid?(uidt(dt_integral(typ(t')))(to_byte(dt_integral(typ(t')))(d, a), a)</code>
{-3}	$\forall (l: list[Byte], a: Address):$ <code>valid?(uidt(dt_integral(typ(t')))(l, a) \equiv</code> <code>up?(from_byte(dt_integral(typ(t')))(l, a)</code>
{-4}	$\forall (d: ((range_integral(typ(t')))), a: Address):$ <code>down(from_byte(dt_integral(typ(t'))</code> <code>(to_byte(dt_integral(typ(t')))(d, a), a)</code> <code>= d</code>
{-5}	<code>non_bool_integral?(typ(t'))</code>
{1}	$\forall (d: ((range(t')), a: Address):$ <code>valid?(uidt(dt_integral(typ(t')))(to_byte(dt_integral(typ(t')))(d, a), a)</code>

Repeatedly Skolemizing and flattening,

Instantiating quantified variables,

Expanding the definition of range,

Keeping (-1 -5 1) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `dt_TCC1.3.1.2.1.4.1`.

dt_TCC1.3.1.2.1.4.2:

{-1}	uninterpreted_data_type?(dt_integral(typ(t'))'uidt)
{-2}	$\forall (d: ((\text{range_integral}(\text{typ}(t')))), a: \text{Address}):$ valid?(uidt(dt_integral(typ(t')))(to_byte(dt_integral(typ(t')))(d, a), a)
{-3}	$\forall (l: \text{list}[\text{Byte}], a: \text{Address}):$ valid?(uidt(dt_integral(typ(t')))(l, a) \equiv up?(from_byte(dt_integral(typ(t')))(l, a))
{-4}	$\forall (d: ((\text{range_integral}(\text{typ}(t')))), a: \text{Address}):$ down(from_byte(dt_integral(typ(t'))) (to_byte(dt_integral(typ(t')))(d, a), a)) $= d$
{-5}	non_bool_integral?(typ(t'))
{1}	$\forall (d: ((\text{range}(t'))), a: \text{Address}):$ down(from_byte(dt_integral(typ(t')) (to_byte(dt_integral(typ(t')))(d, a), a)) $= d$

Repeatedly Skolemizing and flattening,

Instantiating quantified variables,

Keeping (-1 -5 1) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of dt_TCC1.3.1.2.1.4.2.

dt_TCC1.3.1.2.2:

{-1}	non_bool_integral?(typ(t'))
{-2}	Cpp_Type?(const(t'))
{-3}	volatile?(t')
{-4}	typ' = const(t')
{1}	non_bool_integral?(typ(t'))
{2}	$(\forall (x: \text{int}): \text{range_integral}(\text{typ}(t'))(x) \equiv \text{range}(t')(x)) \wedge$ $(\forall (x: \text{int}): \text{range_integral}(\text{typ}(t'))(x) \equiv \text{range}(t')(x)) \wedge$ $(\forall (x_1: [\text{list}[\text{Byte}], \text{Address}]):$ every[[range_integral(typ(t'))]] (range(t'))(dt_integral(typ(t'))'from_byte(x ₁)) $\wedge \text{interpreted_data_type?}[[\text{range}(t')]](\text{dt_integral}(\text{typ}(t')))$

which is trivially true.

This completes the proof of dt_TCC1.3.1.2.2.

dt_TCC1.3.1.3:

{-1}	non_bool_integral?(typ(t'))
{-2}	Cpp_Type?(const(t'))
{-3}	volatile?(t')
{-4}	typ' = const(t')
{1}	non_bool_integral?(typ(t'))

which is trivially true.

This completes the proof of dt_TCC1.3.1.3.

dt_TCC1.3.2:

{-1}	enum?(typ(t'))
{-2}	Cpp_Type?(const(t'))
{-3}	volatile?(t')
{-4}	typ' = const(t')
{1}	$\exists (x_1: (\text{interpreted_data_type?}[[\text{range}(t')]])): \text{TRUE}$

Instantiating the top quantifier in 1 with the terms: $(\text{dt_enum}(\text{typ}(t!1)))$,
we get 3 subgoals:
dt_TCC1.3.2.1:

{-1}	enum?(typ(t'))
{-2}	CppType?(const(t'))
{-3}	volatile?(t')
{-4}	typ' = const(t')
{1}	TRUE

which is trivially true.

This completes the proof of dt_TCC1.3.2.1.

dt_TCC1.3.2.2:

{-1}	enum?(typ(t'))
{-2}	CppType?(const(t'))
{-3}	volatile?(t')
{-4}	typ' = const(t')
{1}	$(\forall (x: \text{int}): \text{range_enum}(\text{typ}(t'))(x) \equiv \text{range}(t')(x)) \wedge$ $(\forall (x: \text{int}): \text{range_enum}(\text{typ}(t'))(x) \equiv \text{range}(t')(x)) \wedge$ $(\forall (x_1: [\text{list}[\text{Byte}], \text{Address}]):$ $\quad \text{every}[(\text{range_enum}(\text{typ}(t')))]$ $\quad \quad ((\text{range}(t'))(\text{dt_enum}(\text{typ}(t')) \text{'from_byte}(x_1)))$ $\quad \wedge \text{interpreted_data_type?}[(\text{range}(t'))](\text{dt_enum}(\text{typ}(t')))$

Expanding the definition of range,

Expanding the definition of range,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 2 subgoals:

dt_TCC1.3.2.2.1:

{-1}	enum?(typ(t'))
{-2}	CppType?(const(t'))
{-3}	volatile?(t')
{-4}	typ' = const(t')
{1}	interpreted_data_type?[(range(t'))](dt_enum(typ(t')))

Adding type constraints for dt_enum(typ(t!1)),

Using lemma pod_is_interpreted_data[(range_enum(typ(t')))],

Keeping (-1 -3 1) and hiding *,

Expanding the definition of interpreted_data_type?,

Applying propositional simplification,

we get 2 subgoals:

dt_TCC1.3.2.2.1.1:

{-1}	uninterpreted_data_type?(dt_enum(typ(t'))'uidt)
{-2}	$\forall (d: ((\text{range_enum}(\text{typ}(t')))), a: \text{Address}):$ $\quad \text{valid?}(\text{uidt}(\text{dt_enum}(\text{typ}(t')))(\text{to_byte}(\text{dt_enum}(\text{typ}(t')))(d, a), a)$
{-3}	$\forall (l: \text{list}[\text{Byte}], a: \text{Address}):$ $\quad \text{valid?}(\text{uidt}(\text{dt_enum}(\text{typ}(t')))(l, a) \equiv$ $\quad \quad \text{up?}(\text{from_byte}(\text{dt_enum}(\text{typ}(t')))(l, a))$
{-4}	$\forall (d: ((\text{range_enum}(\text{typ}(t')))), a: \text{Address}):$ $\quad \text{down}(\text{from_byte}(\text{dt_enum}(\text{typ}(t')))(\text{to_byte}(\text{dt_enum}(\text{typ}(t')))(d, a), a) = d$
{-5}	enum?(typ(t'))
{1}	$\forall (d: ((\text{range}(t'))), a: \text{Address}):$ $\quad \text{valid?}(\text{uidt}(\text{dt_enum}(\text{typ}(t')))(\text{to_byte}(\text{dt_enum}(\text{typ}(t')))(d, a), a)$

Repeatedly Skolemizing and flattening,
 Instantiating quantified variables,
 Keeping (-1 -5 1) and hiding *,
 Expanding the definition of range,
 Expanding the definition of range,
 which is trivially true.
 This completes the proof of dt_TCC1.3.2.2.1.1.
 dt_TCC1.3.2.2.1.2:

{-1}	uninterpreted_data_type?(dt_enum(typ(t')) 'uidt)
{-2}	$\forall (d: ((\text{range_enum}(\text{typ}(t')))), a: \text{Address}):$ $\text{valid?}(\text{uidt}(\text{dt_enum}(\text{typ}(t'))))(\text{to_byte}(\text{dt_enum}(\text{typ}(t')))(d, a), a)$
{-3}	$\forall (l: \text{list}[\text{Byte}], a: \text{Address}):$ $\text{valid?}(\text{uidt}(\text{dt_enum}(\text{typ}(t'))))(l, a) \equiv$ $\text{up?}(\text{from_byte}(\text{dt_enum}(\text{typ}(t'))))(l, a)$
{-4}	$\forall (d: ((\text{range_enum}(\text{typ}(t')))), a: \text{Address}):$ $\text{down}(\text{from_byte}(\text{dt_enum}(\text{typ}(t'))))(\text{to_byte}(\text{dt_enum}(\text{typ}(t')))(d, a), a) = d$
{-5}	$\text{enum?}(\text{typ}(t'))$
{1}	$\forall (d: ((\text{range}(t'))), a: \text{Address}):$ $\text{down}(\text{from_byte}(\text{dt_enum}(\text{typ}(t'))))(\text{to_byte}(\text{dt_enum}(\text{typ}(t')))(d, a), a) = d$

Repeatedly Skolemizing and flattening,
 Instantiating quantified variables,
 Expanding the definition of range,
 Expanding the definition of range,
 which is trivially true.
 This completes the proof of dt_TCC1.3.2.2.1.2.
 dt_TCC1.3.2.2.2:

{-1}	$\text{enum?}(\text{typ}(t'))$
{-2}	$\text{Cpp_Type?}(\text{const}(t'))$
{-3}	$\text{volatile?}(t')$
{-4}	$\text{typ}' = \text{const}(t')$
{1}	$\forall (x_1: [\text{list}[\text{Byte}], \text{Address}]):$ $\text{every}[[((\text{range_enum}(\text{typ}(t'))))] (\text{range_enum}(\text{typ}(t')))(\text{dt_enum}(\text{typ}(t')) 'from_byte(x_1))$

Expanding the definition of every,
 Repeatedly Skolemizing and flattening,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 This completes the proof of dt_TCC1.3.2.2.2.
 dt_TCC1.3.2.3:

{-1}	$\text{enum?}(\text{typ}(t'))$
{-2}	$\text{Cpp_Type?}(\text{const}(t'))$
{-3}	$\text{volatile?}(t')$
{-4}	$\text{typ}' = \text{const}(t')$
{1}	$\text{Cpp_Type?}(\text{typ}(t')) \wedge \text{enum?}(\text{typ}(t'))$

Simplifying, rewriting, and recording with decision procedures,
 Expanding the definition of Cpp_Type?,
 Repeatedly Skolemizing and flattening,
 Instantiating the top quantifier in -3 with the terms: (t!2),
 Expanding the definition of subterm,
 Expanding the definition of subterm,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of dt_TCC1.3.2.3.

Q.E.D.

C.44.34 Cpp_Types.dt_TCC2

Terse proof for dt_TCC2.

dt_TCC2:

$ \begin{array}{l} \{1\} \quad \forall (\text{typ}: \text{Cpp_Subtype}(\text{cv}(\text{non_bool_integral_enum?})), \\ \quad \quad \quad v: \\ \quad \quad \quad [\text{typ1}: \\ \quad \quad \quad \quad \{z: \text{Cpp_Subtype}(\text{cv}(\text{non_bool_integral_enum?})) \mid \\ \quad \quad \quad \quad \quad \text{restrict} \\ \quad \quad \quad \quad \quad \quad [[\text{Cpp_Type_}, \text{Cpp_Type_}], \\ \quad \quad \quad \quad \quad \quad \quad [\text{Cpp_Subtype}(\text{cv}(\text{non_bool_integral_enum?})), \\ \quad \quad \quad \quad \quad \quad \quad \quad \text{Cpp_Subtype}(\text{cv}(\text{non_bool_integral_enum?}))], \\ \quad \quad \quad \quad \quad \quad \quad \quad \text{boolean}] \\ \quad \quad \quad \quad \quad \quad \quad \quad (\ll)(z, \text{typ})\} \rightarrow \\ \quad \quad \quad \quad \quad \quad \quad \quad (\text{interpreted_data_type?}[(\text{range}(\text{typ1}))]), \\ \quad \quad \quad \quad \quad \quad \quad \quad t: \text{Cpp_Type_}: \\ \quad \quad \quad \quad \quad \quad \quad \quad \text{typ} = \text{const}(t) \supset \\ \quad \quad \quad \quad \quad \quad \quad \quad (\forall (x: \text{int}): \text{range}(t)(x) \equiv \text{range}(\text{typ})(x)) \wedge \\ \quad \quad \quad \quad \quad \quad \quad \quad (\forall (x: \text{int}): \text{range}(t)(x) \equiv \text{range}(\text{typ})(x)) \wedge \\ \quad \quad \quad \quad \quad \quad \quad \quad (\forall (x_1: [\text{list}[\text{Byte}], \text{Address}]): \\ \quad \quad \quad \quad \quad \quad \quad \quad \quad \text{every}[(\text{range}(t))] \\ \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad (\text{range}(\text{typ}))(\text{dt_const}[(\text{range}(t))](v(t)) \text{'from_byte}(x_1)) \\ \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \wedge \text{interpreted_data_type?}[(\text{range}(\text{typ}))](\text{dt_const}[(\text{range}(t))](v(t))) \end{array} $

Repeatedly Skolemizing and flattening,

Replacing using formula -3,

Expanding the definition of range,

Applying propositional simplification,

we get 2 subgoals:

dt_TCC2.1:

$ \begin{array}{l} \{-1\} \quad \text{Cpp_Type?}(\text{const}(t')) \\ \{-2\} \quad \text{cv}(\text{non_bool_integral_enum?})(\text{const}(t')) \\ \{-3\} \quad \text{typ}' = \text{const}(t') \\ \hline \{1\} \quad \forall (x_1: [\text{list}[\text{Byte}], \text{Address}]): \\ \quad \quad \quad \text{every}[(\text{range}(t'))] \\ \quad \quad \quad \quad (\text{range}(\text{const}(t'))) \\ \quad \quad \quad \quad (\text{dt_const}[(\text{range}(t'))](v'(t')) \text{'from_byte}(x_1)) \end{array} $

Repeatedly Skolemizing and flattening,

Expanding the definition of every,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Expanding the definition of range,

which is trivially true.

This completes the proof of dt_TCC2.1.

dt_TCC2.2:

{-1}	Cpp_Type?(const(t'))
{-2}	cv(non_bool_integral_enum?)(const(t'))
{-3}	$\text{typ}' = \text{const}(t')$
{1}	interpreted_data_type?[(range(typ'))](dt_const[(range(t'))]($v'(t')$))

Adding type constraints for dt_const($v!1(t!1)$),

we get 2 subgoals:

dt_TCC2.2.1:

{-1}	interpreted_data_type?[(range(t'))](dt_const($v'(t')$))
{-2}	Cpp_Type?(const(t'))
{-3}	cv(non_bool_integral_enum?)(const(t'))
{-4}	$\text{typ}' = \text{const}(t')$
{1}	interpreted_data_type?[(range(typ'))](dt_const[(range(t'))]($v'(t')$))

Expanding the definition of interpreted_data_type?,

Applying propositional simplification,

we get 2 subgoals:

dt_TCC2.2.1.1:

{-1}	uninterpreted_data_type?(dt_const($v'(t')$)'uidt)
{-2}	$\forall (d: ((\text{range}(t')), a: \text{Address}):$ valid?(uidt(dt_const($v'(t')$)))(to_byte(dt_const($v'(t')$))(d, a, a))
{-3}	$\forall (l: \text{list}[\text{Byte}], a: \text{Address}):$ valid?(uidt(dt_const($v'(t')$)))(l, a) \equiv up?(from_byte(dt_const($v'(t')$)))(l, a)
{-4}	$\forall (d: ((\text{range}(t')), a: \text{Address}):$ down(from_byte(dt_const($v'(t')$)))(to_byte(dt_const($v'(t')$))(d, a, a)) = d
{-5}	Cpp_Type?(const(t'))
{-6}	cv(non_bool_integral_enum?)(const(t'))
{-7}	$\text{typ}' = \text{const}(t')$
{1}	$\forall (d: ((\text{range}(\text{typ}')), a: \text{Address}):$ valid?(uidt(dt_const[(range(t'))]($v'(t')$)))(to_byte(dt_const[(range(t'))]($v'(t')$))(d, a, a))

Repeatedly Skolemizing and flattening,

Instantiating quantified variables,

Expanding the definition of range,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Replacing using formula -7,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of dt_TCC2.2.1.1.

dt_TCC2.2.1.2:

{-1}	uninterpreted_data_type?(dt_const(v'(t'))'uidt)
{-2}	$\forall (d: ((\text{range}(t'))), a: \text{Address}):$ valid?(uidt(dt_const(v'(t')))(to_byte(dt_const(v'(t')))(d, a), a)
{-3}	$\forall (l: \text{list}[\text{Byte}], a: \text{Address}):$ valid?(uidt(dt_const(v'(t')))(l, a) \equiv up?(from_byte(dt_const(v'(t')))(l, a))
{-4}	$\forall (d: ((\text{range}(t'))), a: \text{Address}):$ down(from_byte(dt_const(v'(t')))(to_byte(dt_const(v'(t')))(d, a), a) =
{-5}	$\overset{d}{\text{Cpp_Type?}}(\text{const}(t'))$
{-6}	cv(non_bool_integral_enum?)(const(t'))
{-7}	typ' = const(t')
{1}	$\forall (d: ((\text{range}(\text{typ}')), a: \text{Address}):$ down(from_byte(dt_const[((range(t')))](v'(t')))(to_byte(dt_const[((range(t')))](v'(t')))(d, a), a) = d

Repeatedly Skolemizing and flattening,
Instantiating quantified variables,
Replacing using formula -7,
Expanding the definition of range,
which is trivially true.

This completes the proof of dt_TCC2.2.1.2.

dt_TCC2.2.2:

{-1}	Cpp_Type?(const(t'))
{-2}	cv(non_bool_integral_enum?)(const(t'))
{-3}	typ' = const(t')
{1}	$\exists (x_1: (\text{interpreted_data_type?}[\text{range}(t'))]): \text{TRUE}$
{2}	interpreted_data_type?[\text{range}(typ')](dt_const[\text{range}(t')](v'(t')))

Using lemma dt_TCC1,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of dt_TCC2.2.2.

Q.E.D.

C.44.35 Cpp_Types.dt_TCC3

Terse proof for dt_TCC3.

dt_TCC3:

{1}	$\forall (\text{typ}: \text{Cpp_Subtype}(\text{cv}(\text{non_bool_integral_enum?})), t: \text{Cpp_Type_}):$ typ = volatile(t) $\supset (\exists (x_1: (\text{interpreted_data_type?}[\text{range}(t)])): \text{TRUE})$
-----	---

Repeatedly Skolemizing and flattening,

Replacing using formula -3,

Expanding the definition of cv,

Expanding the definition of c,

Expanding the definition of v,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 2 subgoals:

dt_TCC3.1:

{-1}	Cpp_Type?(volatile(t'))
{-2}	non_bool_integral_enum?(volatile(t'))
{-3}	typ' = volatile(t')
{1}	$\exists (x_1: (\text{interpreted_data_type?}[\text{range}(t')])): \text{TRUE}$

Keeping (-2) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of dt_TCC3.1.

dt_TCC3.2:

{-1}	Cpp_Type?(volatile(t'))
{-2}	non_bool_integral_enum?(t')
{-3}	typ' = volatile(t')
{1}	$\exists (x_1: (\text{interpreted_data_type?}[\text{range}(t')])): \text{TRUE}$

Expanding the definition of non_bool_integral_enum?,

Applying propositional simplification,

we get 2 subgoals:

dt_TCC3.2.1:

{-1}	non_bool_integral?(t')
{-2}	Cpp_Type?(volatile(t'))
{-3}	typ' = volatile(t')
{1}	$\exists (x_1: (\text{interpreted_data_type?}[\text{range}(t')])): \text{TRUE}$

Instantiating the top quantifier in 1 with the terms: (dt_integral(t!1)),

we get 3 subgoals:

dt_TCC3.2.1.1:

{-1}	non_bool_integral?(t')
{-2}	Cpp_Type?(volatile(t'))
{-3}	typ' = volatile(t')
{1}	TRUE

which is trivially true.

This completes the proof of dt_TCC3.2.1.1.

dt_TCC3.2.1.2:

{-1}	non_bool_integral?(t')
{-2}	Cpp_Type?(volatile(t'))
{-3}	typ' = volatile(t')
{1}	$(\forall (x: \text{int}): \text{range_integral}(t')(x) \equiv \text{range}(t')(x)) \wedge$ $(\forall (x: \text{int}): \text{range_integral}(t')(x) \equiv \text{range}(t')(x)) \wedge$ $(\forall (x_1: [\text{list}[\text{Byte}], \text{Address}]):$ $\quad \text{every}[\text{range_integral}(t')]$ $\quad \quad ((\text{range}(t'))(\text{dt_integral}(t)' \text{from_byte}(x_1)))$ $\quad \wedge \text{interpreted_data_type?}[\text{range}(t')](\text{dt_integral}(t'))$

Applying propositional simplification,

we get 4 subgoals:

dt_TCC3.2.1.2.1:

{-1}	non_bool_integral?(t')
{-2}	Cpp_Type?(volatile(t'))
{-3}	typ' = volatile(t')
{1}	$\forall (x: \text{int}): \text{range_integral}(t')(x) \equiv \text{range}(t')(x)$

C Proof scripts

Keeping (-1 1) and hiding *,
 Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of dt_TCC3.2.1.2.1.
 dt_TCC3.2.1.2.2:

{-1}	non_boolIntegral?(t')
{-2}	Cpp_Type?(volatile(t'))
{-3}	typ' = volatile(t')
{1}	$\forall (x: \text{int}): \text{range_integral}(t')(x) \equiv \text{range}(t')(x)$

Keeping (-1 1) and hiding *,
 Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of dt_TCC3.2.1.2.2.
 dt_TCC3.2.1.2.3:

{-1}	non_boolIntegral?(t')
{-2}	Cpp_Type?(volatile(t'))
{-3}	typ' = volatile(t')
{1}	$\forall (x_1: [\text{list}[\text{Byte}], \text{Address}]):$ $\text{every}[(\text{range_integral}(t'))](\text{range}(t'))(\text{dt_integral}(t)' \text{from_byte}(x_1))$

Repeatedly Skolemizing and flattening,
 Expanding the definition of every,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 Adding type constraints for down(dt_integral(t!1)'from_byte(x1!1)),
 Expanding the definition of range,
 Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of dt_TCC3.2.1.2.3.
 dt_TCC3.2.1.2.4:

{-1}	non_boolIntegral?(t')
{-2}	Cpp_Type?(volatile(t'))
{-3}	typ' = volatile(t')
{1}	interpreted_data_type?[(range(t'))](dt_integral(t'))

Adding type constraints for dt_integral(t!1),
 Using lemma pod_is_interpreted_data[(range_integral(t'))],
 Expanding the definition of interpreted_data_type?,
 Applying propositional simplification,
 we get 2 subgoals:
 dt_TCC3.2.1.2.4.1:

{-1}	uninterpreted_data_type?(dt_integral(t)'uidt)
{-2}	$\forall (d: (\text{range_integral}(t')), a: \text{Address}):$ $\text{valid?}(\text{uidt}(\text{dt_integral}(t')))(\text{to_byte}(\text{dt_integral}(t'))(d, a), a)$
{-3}	$\forall (l: \text{list}[\text{Byte}], a: \text{Address}):$ $\text{valid?}(\text{uidt}(\text{dt_integral}(t')))(l, a) \equiv \text{up?}(\text{from_byte}(\text{dt_integral}(t'))(l, a))$
{-4}	$\forall (d: (\text{range_integral}(t')), a: \text{Address}):$ $\text{down}(\text{from_byte}(\text{dt_integral}(t'))(\text{to_byte}(\text{dt_integral}(t'))(d, a), a)) = d$
{-5}	pod_data_type?[(range_integral(t'))](dt_integral(t'))
{-6}	non_boolIntegral?(t')
{-7}	Cpp_Type?(volatile(t'))
{-8}	typ' = volatile(t')
{1}	$\forall (d: (\text{range}(t')), a: \text{Address}):$ $\text{valid?}(\text{uidt}(\text{dt_integral}(t')))(\text{to_byte}(\text{dt_integral}(t'))(d, a), a)$

Repeatedly Skolemizing and flattening,
 Instantiating quantified variables,
 Keeping (-1 -6 1) and hiding *,
 Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of dt_TCC3.2.1.2.4.1.
 dt_TCC3.2.1.2.4.2:

{-1}	uninterpreted_data_type?(dt_integral(t)' uidt)
{-2}	$\forall (d: ((\text{range_integral}(t))), a: \text{Address}):$ valid?(uidt(dt_integral(t'))(to_byte(dt_integral(t'))(d, a), a)
{-3}	$\forall (l: \text{list}[\text{Byte}], a: \text{Address}):$ valid?(uidt(dt_integral(t'))(l, a) \equiv up?(from_byte(dt_integral(t'))(l, a))
{-4}	$\forall (d: ((\text{range_integral}(t))), a: \text{Address}):$ down(from_byte(dt_integral(t'))(to_byte(dt_integral(t'))(d, a), a)) = d
{-5}	pod_data_type?[(range_integral(t'))](dt_integral(t'))
{-6}	non_bool_integral?(t')
{-7}	Cpp_Type?(volatile(t'))
{-8}	typ' = volatile(t')
{1}	$\forall (d: ((\text{range}(t))), a: \text{Address}):$ down(from_byte(dt_integral(t'))(to_byte(dt_integral(t'))(d, a), a)) = d

Repeatedly Skolemizing and flattening,
 Instantiating quantified variables,
 Keeping (-1 -6 1) and hiding *,
 Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of dt_TCC3.2.1.2.4.2.
 dt_TCC3.2.1.3:

{-1}	non_bool_integral?(t')
{-2}	Cpp_Type?(volatile(t'))
{-3}	typ' = volatile(t')
{1}	non_bool_integral?(t')

which is trivially true.

This completes the proof of dt_TCC3.2.1.3.

dt_TCC3.2.2:

{-1}	enum?(t')
{-2}	Cpp_Type?(volatile(t'))
{-3}	typ' = volatile(t')
{1}	$\exists (x_1: (\text{interpreted_data_type?}[(\text{range}(t'))])): \text{TRUE}$

Instantiating the top quantifier in 1 with the terms: (dt_enum(t!1)),
 we get 3 subgoals:

dt_TCC3.2.2.1:

{-1}	enum?(t')
{-2}	Cpp_Type?(volatile(t'))
{-3}	typ' = volatile(t')
{1}	TRUE

which is trivially true.

This completes the proof of dt_TCC3.2.2.1.

dt_TCC3.2.2.2:

{-1}	enum?(t')
{-2}	CppType?(volatile(t'))
{-3}	typ' = volatile(t')
{1}	$(\forall (x: \text{int}): \text{range_enum}(t')(x) \equiv \text{range}(t')(x)) \wedge$ $(\forall (x: \text{int}): \text{range_enum}(t')(x) \equiv \text{range}(t')(x)) \wedge$ $(\forall (x_1: [\text{list}[\text{Byte}], \text{Address}]):$ $\quad \text{every}[[\text{range_enum}(t')]]((\text{range}(t'))(\text{dt_enum}(t)' \text{from_byte}(x_1)))$ $\quad \wedge \text{interpreted_data_type?}[[\text{range}(t')]](\text{dt_enum}(t'))$

Expanding the definition of range,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 2 subgoals:

dt_TCC3.2.2.2.1:

{-1}	enum?(t')
{-2}	CppType?(volatile(t'))
{-3}	typ' = volatile(t')
{1}	interpreted_data_type?[[range(t')]](dt_enum(t'))

Adding type constraints for dt_enum(t'),

Using lemma pod_is_interpreted_data[[range_enum(t')],

Hiding formulas: (-2 -4 -5),

Expanding the definition of interpreted_data_type?,

Applying propositional simplification,

we get 2 subgoals:

dt_TCC3.2.2.2.1.1:

{-1}	uninterpreted_data_type?(dt_enum(t')'uidt)
{-2}	$\forall (d: ((\text{range_enum}(t'))), a: \text{Address}):$ $\quad \text{valid?}(\text{uidt}(\text{dt_enum}(t')))(\text{to_byte}(\text{dt_enum}(t'))(d, a), a)$
{-3}	$\forall (l: \text{list}[\text{Byte}], a: \text{Address}):$ $\quad \text{valid?}(\text{uidt}(\text{dt_enum}(t')))(l, a) \equiv \text{up?}(\text{from_byte}(\text{dt_enum}(t')))(l, a)$
{-4}	$\forall (d: ((\text{range_enum}(t'))), a: \text{Address}):$ $\quad \text{down}(\text{from_byte}(\text{dt_enum}(t')))(\text{to_byte}(\text{dt_enum}(t'))(d, a), a) = d$
{-5}	enum?(t')
{1}	$\forall (d: ((\text{range}(t'))), a: \text{Address}):$ $\quad \text{valid?}(\text{uidt}(\text{dt_enum}(t')))(\text{to_byte}(\text{dt_enum}(t'))(d, a), a)$

Repeatedly Skolemizing and flattening,

Instantiating quantified variables,

Expanding the definition of range,

which is trivially true.

This completes the proof of dt_TCC3.2.2.2.1.1.

dt_TCC3.2.2.2.1.2:

{-1}	uninterpreted_data_type?(dt_enum(t') 'uidt)
{-2}	$\forall (d: ((\text{range_enum}(t')), a: \text{Address}):$ valid?(uidt(dt_enum(t'))(to_byte(dt_enum(t'))(d, a), a)
{-3}	$\forall (l: \text{list}[\text{Byte}], a: \text{Address}):$ valid?(uidt(dt_enum(t'))(l, a) \equiv up?(from_byte(dt_enum(t'))(l, a))
{-4}	$\forall (d: ((\text{range_enum}(t')), a: \text{Address}):$ down(from_byte(dt_enum(t'))(to_byte(dt_enum(t'))(d, a), a)) = d
{-5}	enum?(t')
{1}	$\forall (d: ((\text{range}(t')), a: \text{Address}):$ down(from_byte(dt_enum(t'))(to_byte(dt_enum(t'))(d, a), a)) = d

Repeatedly Skolemizing and flattening,

Instantiating quantified variables,

Expanding the definition of range,

which is trivially true.

This completes the proof of dt_TCC3.2.2.2.1.2.

dt_TCC3.2.2.2.2:

{-1}	enum?(t')
{-2}	Cpp_Type?(volatile(t'))
{-3}	typ' = volatile(t')
{1}	$\forall (x_1: [\text{list}[\text{Byte}], \text{Address}]):$ every[[(range_enum(t'))]](range_enum(t')(dt_enum(t') 'from_byte(x_1))

Expanding the definition of every,

Repeatedly Skolemizing and flattening,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of dt_TCC3.2.2.2.2.

dt_TCC3.2.2.3:

{-1}	enum?(t')
{-2}	Cpp_Type?(volatile(t'))
{-3}	typ' = volatile(t')
{1}	Cpp_Type?(t') \wedge enum?(t')

Simplifying, rewriting, and recording with decision procedures,

Expanding the definition of Cpp_Type?,

Repeatedly Skolemizing and flattening,

Instantiating the top quantifier in -3 with the terms: (t!2),

Expanding the definition of subterm,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of dt_TCC3.2.2.3.

Q.E.D.

C.44.36 Cpp_Types.dt_TCC4

Terse proof for dt_TCC4.

dt_TCC4:

$ \begin{aligned} \{1\} \quad & \forall (\text{typ}: \text{Cpp_Subtype}(\text{cv}(\text{non_bool_integral_enum?})), \\ & v: \\ & \quad [\text{typ1}: \\ & \quad \quad \{z: \text{Cpp_Subtype}(\text{cv}(\text{non_bool_integral_enum?})) \mid \\ & \quad \quad \quad \text{restrict} \\ & \quad \quad \quad \quad [[\text{Cpp_Type_}, \text{Cpp_Type_}], \\ & \quad \quad \quad \quad [\text{Cpp_Subtype}(\text{cv}(\text{non_bool_integral_enum?})), \\ & \quad \quad \quad \quad \text{Cpp_Subtype}(\text{cv}(\text{non_bool_integral_enum?}))], \\ & \quad \quad \quad \quad \text{boolean}] \\ & \quad \quad \quad (\ll)(z, \text{typ})\} \rightarrow \\ & \quad \quad \quad (\text{interpreted_data_type?}[\text{((range(typ1)))]}), \\ & t: \text{Cpp_Type_}: \\ & \text{typ} = \text{volatile}(t) \supset \\ & (\forall (x: \text{int}): \text{range}(t)(x) \equiv \text{range}(\text{typ})(x)) \wedge \\ & (\forall (x: \text{int}): \text{range}(t)(x) \equiv \text{range}(\text{typ})(x)) \wedge \\ & (\forall (x_1: [\text{list}[\text{Byte}], \text{Address}]): \\ & \quad \text{every}[\text{((range}(t)))] \\ & \quad \quad (\text{range}(\text{typ})) \\ & \quad \quad (\text{dt_volatile}[\text{((range}(t)))](v(t)) \text{'from_byte}(x_1))) \\ & \wedge \\ & \text{interpreted_data_type?}[\text{range}(\text{typ})](\text{dt_volatile}[\text{((range}(t)))](v(t))) \end{aligned} $
--

Repeatedly Skolemizing and flattening,
 Replacing using formula -3,
 Expanding the definition of range,
 Applying propositional simplification,
 we get 2 subgoals:

dt_TCC4.1:

$ \begin{aligned} \{-1\} \quad & \text{Cpp_Type?}(\text{volatile}(t')) \\ \{-2\} \quad & \text{cv}(\text{non_bool_integral_enum?})(\text{volatile}(t')) \\ \{-3\} \quad & \text{typ}' = \text{volatile}(t') \end{aligned} $
$ \{1\} \quad \forall (x_1: [\text{list}[\text{Byte}], \text{Address}]): \\ \quad \text{every}[\text{((range}(t')))] \\ \quad \quad (\text{range}(\text{volatile}(t'))) \\ \quad \quad (\text{dt_volatile}[\text{((range}(t')))](v'(t')) \text{'from_byte}(x_1)) $

Repeatedly Skolemizing and flattening,
 Expanding the definition of every,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 Expanding the definition of range,
 which is trivially true.

This completes the proof of dt_TCC4.1.

dt_TCC4.2:

$ \begin{aligned} \{-1\} \quad & \text{Cpp_Type?}(\text{volatile}(t')) \\ \{-2\} \quad & \text{cv}(\text{non_bool_integral_enum?})(\text{volatile}(t')) \\ \{-3\} \quad & \text{typ}' = \text{volatile}(t') \end{aligned} $
$ \{1\} \quad \text{interpreted_data_type?}[\text{((range}(\text{typ}')\text{))}] \\ \quad \quad (\text{dt_volatile}[\text{((range}(t')))](v'(t'))) $

Adding type constraints for dt_volatile(v!1(t!1)),

we get 2 subgoals:

dt_TCC4.2.1.1:

{-1}	interpreted_data_type?[((range(t')))](dt_volatile(v'(t')))
{-2}	Cpp_Type?(volatile(t'))
{-3}	cv(non_bool_integral_enum?)(volatile(t'))
{-4}	typ' = volatile(t')
{1}	interpreted_data_type?[((range(typ')))] (dt_volatile[((range(t')))](v'(t')))

Expanding the definition of interpreted_data_type?,

Applying propositional simplification,

we get 2 subgoals:

dt_TCC4.2.1.1.1:

{-1}	uninterpreted_data_type?(dt_volatile(v'(t'))'uidt)
{-2}	$\forall (d: ((range(t'))), a: Address):$ valid?(uidt(dt_volatile(v'(t')))(to_byte(dt_volatile(v'(t')))(d, a), a)
{-3}	$\forall (l: list[Byte], a: Address):$ valid?(uidt(dt_volatile(v'(t')))(l, a) \equiv up?(from_byte(dt_volatile(v'(t')))(l, a))
{-4}	$\forall (d: ((range(t'))), a: Address):$ down(from_byte(dt_volatile(v'(t'))) (to_byte(dt_volatile(v'(t')))(d, a), a)) $= d$
{-5}	Cpp_Type?(volatile(t'))
{-6}	cv(non_bool_integral_enum?)(volatile(t'))
{-7}	typ' = volatile(t')
{1}	$\forall (d: ((range(typ'))), a: Address):$ valid?(uidt(dt_volatile[((range(t')))](v'(t'))) (to_byte(dt_volatile[((range(t')))](v'(t')))(d, a), a)

Repeatedly Skolemizing and flattening,

Instantiating quantified variables,

Expanding the definition of range,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Replacing using formula -7,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of dt_TCC4.2.1.1.

dt_TCC4.2.1.2:

{-1}	uninterpreted_data_type?(dt_volatile(v'(t'))'uidt)
{-2}	$\forall (d: ((\text{range}(t'))), a: \text{Address}):$ valid?(uidt(dt_volatile(v'(t')))(to_byte(dt_volatile(v'(t')))(d, a), a)
{-3}	$\forall (l: \text{list}[\text{Byte}], a: \text{Address}):$ valid?(uidt(dt_volatile(v'(t')))(l, a) \equiv up?(from_byte(dt_volatile(v'(t')))(l, a)
{-4}	$\forall (d: ((\text{range}(t'))), a: \text{Address}):$ down(from_byte(dt_volatile(v'(t')))(to_byte(dt_volatile(v'(t')))(d, a), a)) $= d$
{-5}	Cpp_Type?(volatile(t'))
{-6}	cv(non_bool_integral_enum?)(volatile(t'))
{-7}	typ' = volatile(t')
{1}	$\forall (d: ((\text{range}(\text{typ}')), a: \text{Address}):$ down(from_byte(dt_volatile[[(range(t'))]](v'(t')))(to_byte(dt_volatile[[(range(t'))]](v'(t')))(d, a), a)) $= d$

Repeatedly Skolemizing and flattening,
Instantiating quantified variables,
Replacing using formula -7,
Expanding the definition of range,
which is trivially true.

This completes the proof of dt_TCC4.2.1.2.

dt_TCC4.2.2:

{-1}	Cpp_Type?(volatile(t'))
{-2}	cv(non_bool_integral_enum?)(volatile(t'))
{-3}	typ' = volatile(t')
{1}	$\exists (x_1: (\text{interpreted_data_type?}[\text{[(range}(t'))]]))): \text{TRUE}$
{2}	interpreted_data_type?[\text{[(range}(typ'))]] (dt_volatile[\text{[(range}(t'))]](v'(t')))

Using lemma dt_TCC3,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of dt_TCC4.2.2.

Q.E.D.

C.44.37 Cpp_Types.dt_TCC5

Terse proof for dt_TCC5.

dt_TCC5:

{1}	$\forall (\text{typ}: \text{Cpp_Subtype}(\text{cv}(\text{non_bool_integral_enum?}))):$ $\neg \text{volatile?}(\text{typ}) \wedge \neg \text{const?}(\text{typ}) \supset \text{non_bool_integral_enum?}(\text{typ})$
-----	---

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of dt_TCC5.

Q.E.D.

C.44.38 Cpp_Types.dt_pod

Terse proof for dt_pod.

dt_pod:

$$\frac{}{\{1\} \quad \forall (\text{typ}: \text{Cpp_Subtype}(\text{cv}(\text{non_bool_integral_enum}?))) : \text{pod_data_type?}[(\text{range}(\text{typ}))](\text{dt}(\text{typ}))}$$

Repeatedly Skolemizing and flattening,

Installing automatic rewrites from: cv c v

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
we get 4 subgoals:

dt_pod.1:

$$\frac{\begin{array}{l} \{-1\} \quad \text{Cpp_Type?}(\text{typ}') \\ \{-2\} \quad \text{non_bool_integral_enum?}(\text{typ}') \end{array}}{\{1\} \quad \text{pod_data_type?}[(\text{range}(\text{typ}'))](\text{dt}(\text{typ}'))}$$

Expanding the definition of dt,

Adding type constraints for dt_non_bool_integral_enum(typ!1),

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
we get 2 subgoals:

dt_pod.1.1:

$$\frac{\begin{array}{l} \{-1\} \quad \text{pod_data_type?}[(\text{range}(\text{typ}'))](\text{dt_non_bool_integral_enum}(\text{typ}')) \\ \{-2\} \quad \text{Cpp_Type?}(\text{typ}') \\ \{-3\} \quad \text{non_bool_integral_enum?}(\text{typ}') \\ \{-4\} \quad \text{const?}(\text{typ}') \end{array}}{\{1\} \quad \text{pod_data_type?}[(\text{range}(\text{typ}'))](\text{dt_const}(\text{dt}(\text{typ}(\text{typ}'))))}$$

Keeping (-3 -4) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of dt_pod.1.1.

dt_pod.1.2:

$$\frac{\begin{array}{l} \{-1\} \quad \text{pod_data_type?}[(\text{range}(\text{typ}'))](\text{dt_non_bool_integral_enum}(\text{typ}')) \\ \{-2\} \quad \text{Cpp_Type?}(\text{typ}') \\ \{-3\} \quad \text{non_bool_integral_enum?}(\text{typ}') \\ \{-4\} \quad \text{volatile?}(\text{typ}') \end{array}}{\{1\} \quad \text{pod_data_type?}[(\text{range}(\text{typ}'))](\text{dt_volatile}(\text{dt}(\text{typ}(\text{typ}'))))}$$

Keeping (-3 -4) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of dt_pod.1.2.

dt_pod.2:

$$\frac{\begin{array}{l} \{-1\} \quad \text{Cpp_Type?}(\text{typ}') \\ \{-2\} \quad \text{const?}(\text{typ}') \\ \{-3\} \quad \text{non_bool_integral_enum?}(\text{typ}(\text{typ}')) \end{array}}{\{1\} \quad \text{pod_data_type?}[(\text{range}(\text{typ}'))](\text{dt}(\text{typ}'))}$$

Expanding the definition of dt,

Using lemma dt_const_pod[(range(typ(typ')))],

we get 3 subgoals:

C Proof scripts

dt_pod.2.1:

{-1}	pod_data_type?[(range(typ(typ')))](dt_const(dt(typ(typ'))))
{-2}	Cpp_Type?(typ')
{-3}	const?(typ')
{-4}	non_bool_integral_enum?(typ(typ'))
{1}	pod_data_type?[(range(typ'))](dt_const(dt(typ(typ'))))

Expanding the definition of pod_data_type?,
 Adding type constraints for dt_const(dt(typ(typ!1))),
 we get 2 subgoals:

dt_pod.2.1.1:

{-1}	interpreted_data_type?[(range(typ(typ')))](dt_const(dt(typ(typ'))))
{-2}	($\forall (l: \text{list}[\text{Byte}], a_1, a_2: \text{Address}):$ $\text{valid?}(\text{uidt}(\text{dt_const}(\text{dt}(\text{typ}(\text{typ}')))))(l, a_1) \equiv$ $\text{valid?}(\text{uidt}(\text{dt_const}(\text{dt}(\text{typ}(\text{typ}')))))(l, a_2))$ $\wedge \text{size}(\text{uidt}(\text{dt_const}(\text{dt}(\text{typ}(\text{typ}'))))) > 0$)
{-3}	Cpp_Type?(typ')
{-4}	const?(typ')
{-5}	non_bool_integral_enum?(typ(typ'))
{1}	interpreted_data_type?(dt_const(dt(typ(typ')))) \wedge $(\forall (l: \text{list}[\text{Byte}], a_1, a_2: \text{Address}):$ $\text{valid?}(\text{uidt}(\text{dt_const}(\text{dt}(\text{typ}(\text{typ}')))))(l, a_1) \equiv$ $\text{valid?}(\text{uidt}(\text{dt_const}(\text{dt}(\text{typ}(\text{typ}')))))(l, a_2))$ $\wedge \text{size}(\text{uidt}(\text{dt_const}(\text{dt}(\text{typ}(\text{typ}'))))) > 0$)

Expanding the definition of interpreted_data_type?,
 Applying propositional simplification,
 we get 2 subgoals:

dt_pod.2.1.1.1:

{-1}	uninterpreted_data_type?(dt_const(dt(typ(typ')))'uidt)
{-2}	$\forall (d: ((\text{range}(\text{typ}(\text{typ}')))), a: \text{Address}):$ $\text{valid?}(\text{uidt}(\text{dt_const}(\text{dt}(\text{typ}(\text{typ}')))))($ $\text{to_byte}(\text{dt_const}(\text{dt}(\text{typ}(\text{typ}'))))(d, a), a)$
{-3}	$\forall (l: \text{list}[\text{Byte}], a: \text{Address}):$ $\text{valid?}(\text{uidt}(\text{dt_const}(\text{dt}(\text{typ}(\text{typ}')))))(l, a) \equiv$ $\text{up?}(\text{from_byte}(\text{dt_const}(\text{dt}(\text{typ}(\text{typ}')))))(l, a)$
{-4}	$\forall (d: ((\text{range}(\text{typ}(\text{typ}')))), a: \text{Address}):$ $\text{down}(\text{from_byte}(\text{dt_const}(\text{dt}(\text{typ}(\text{typ}')))))($ $\text{to_byte}(\text{dt_const}(\text{dt}(\text{typ}(\text{typ}'))))(d, a), a)$ $= d$
{-5}	$\forall (l: \text{list}[\text{Byte}], a_1, a_2: \text{Address}):$ $\text{valid?}(\text{uidt}(\text{dt_const}(\text{dt}(\text{typ}(\text{typ}')))))(l, a_1) \equiv$ $\text{valid?}(\text{uidt}(\text{dt_const}(\text{dt}(\text{typ}(\text{typ}')))))(l, a_2)$
{-6}	$\text{size}(\text{uidt}(\text{dt_const}(\text{dt}(\text{typ}(\text{typ}'))))) > 0$
{-7}	Cpp_Type?(typ')
{-8}	const?(typ')
{-9}	non_bool_integral_enum?(typ(typ'))
{1}	$\forall (d: ((\text{range}(\text{typ}(\text{typ}')))), a: \text{Address}):$ $\text{valid?}(\text{uidt}(\text{dt_const}(\text{dt}(\text{typ}(\text{typ}')))))($ $\text{to_byte}(\text{dt_const}(\text{dt}(\text{typ}(\text{typ}'))))(d, a), a)$

Repeatedly Skolemizing and flattening,
 Instantiating quantified variables,

Expanding the definition of range,

which is trivially true.

This completes the proof of `dt_pod.2.1.1.1`.

`dt_pod.2.1.1.2`:

<pre> {-1} uninterpreted_data_type?(dt_const(dt(typ(typ')))'uidt) {-2} ∀ (d: ((range(typ(typ')))), a: Address): valid?(uidt(dt_const(dt(typ(typ'))))) (to_byte(dt_const(dt(typ(typ'))))(d, a), a) {-3} ∀ (l: list[Byte], a: Address): valid?(uidt(dt_const(dt(typ(typ'))))(l, a) ≡ up?(from_byte(dt_const(dt(typ(typ'))))(l, a)) {-4} ∀ (d: ((range(typ(typ')))), a: Address): down(from_byte(dt_const(dt(typ(typ'))))) (to_byte(dt_const(dt(typ(typ'))))(d, a), a) = d {-5} ∀ (l: list[Byte], a1, a2: Address): valid?(uidt(dt_const(dt(typ(typ'))))(l, a1) ≡ valid?(uidt(dt_const(dt(typ(typ'))))(l, a2)) {-6} size(uidt(dt_const(dt(typ(typ'))))) > 0 {-7} Cpp_Type?(typ') {-8} const?(typ') {-9} non_bool_integral_enum?(typ(typ')) </pre>	<hr style="border: 0.5px solid black;"/> <pre> {1} ∀ (d: ((range(typ')), a: Address): down(from_byte(dt_const(dt(typ(typ'))))) (to_byte(dt_const(dt(typ(typ'))))(d, a), a) = d </pre>
--	---

Repeatedly Skolemizing and flattening,

Instantiating quantified variables,

Expanding the definition of range,

which is trivially true.

This completes the proof of `dt_pod.2.1.1.2`.

`dt_pod.2.1.2`:

<pre> {-1} (∀ (l: list[Byte], a1, a2: Address): valid?(uidt(dt_const(dt(typ(typ'))))(l, a1) ≡ valid?(uidt(dt_const(dt(typ(typ'))))(l, a2)) ∧ size(uidt(dt_const(dt(typ(typ'))))) > 0) {-2} Cpp_Type?(typ') {-3} const?(typ') {-4} non_bool_integral_enum?(typ(typ')) </pre>	<hr style="border: 0.5px solid black;"/> <pre> {1} ∃ (x1: (interpreted_data_type?[(range(typ(typ')))])): TRUE {2} interpreted_data_type?(dt_const(dt(typ(typ')))) ∧ (∀ (l: list[Byte], a1, a2: Address): valid?(uidt(dt_const(dt(typ(typ'))))(l, a1) ≡ valid?(uidt(dt_const(dt(typ(typ'))))(l, a2)) ∧ size(uidt(dt_const(dt(typ(typ'))))) > 0) </pre>
--	--

Instantiating the top quantifier in 1 with the terms: `(dt(typ(typ!1)))`,

which is trivially true.

This completes the proof of `dt_pod.2.1.2`.

C Proof scripts

dt_pod.2.2:

{-1}	Cpp_Type?(typ')
{-2}	const?(typ')
{-3}	non_bool_integral_enum?(typ(typ'))
{1}	pod_data_type?[(range(typ(typ')))](dt(typ(typ')))
{2}	pod_data_type?[(range(typ'))](dt_const(dt(typ(typ'))))

Hiding formulas: (-2 2),

Expanding the definition of dt,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 2 subgoals:

dt_pod.2.2.1:

{-1}	const?(typ(typ'))
{-2}	Cpp_Type?(typ')
{-3}	non_bool_integral_enum?(typ(typ'))
{1}	pod_data_type?[(range(typ(typ')))](dt_const(dt(typ(typ(typ')))))

Keeping (-1 -3) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of dt_pod.2.2.1.

dt_pod.2.2.2:

{-1}	volatile?(typ(typ'))
{-2}	Cpp_Type?(typ')
{-3}	non_bool_integral_enum?(typ(typ'))
{1}	pod_data_type?[(range(typ(typ')))](dt_volatile(dt(typ(typ(typ')))))

Keeping (-1 -3) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of dt_pod.2.2.2.

dt_pod.2.3:

{-1}	Cpp_Type?(typ')
{-2}	const?(typ')
{-3}	non_bool_integral_enum?(typ(typ'))
{1}	$\exists (x_1 : (\text{interpreted_data_type?}[(\text{range}(\text{typ}(\text{typ}')))])) : \text{TRUE}$
{2}	pod_data_type?[(range(typ'))](dt_const(dt(typ(typ'))))

Instantiating the top quantifier in 1 with the terms: (dt(typ(typ!1))),

which is trivially true.

This completes the proof of dt_pod.2.3.

dt_pod.3:

{-1}	Cpp_Type?(typ')
{-2}	const?(typ')
{-3}	volatile?(typ(typ'))
{-4}	non_bool_integral_enum?(typ(typ(typ')))
{1}	pod_data_type?[(range(typ'))](dt(typ'))

Expanding the definition of dt,

Using lemma dt_const_pod[(range(typ(typ')))],

we get 3 subgoals:

dt_pod.3.1:

{-1}	pod_data_type?[(range(typ(typ')))](dt_const(dt(typ(typ'))))
{-2}	Cpp_Type?(typ')
{-3}	const?(typ')
{-4}	volatile?(typ(typ'))
{-5}	non_bool_integral_enum?(typ(typ(typ')))
{1}	pod_data_type?[((range(typ')))](dt_const(dt(typ(typ'))))

Expanding the definition of pod_data_type?,

Adding type constraints for dt_const(dt(typ(typ!1))),

we get 2 subgoals:

dt_pod.3.1.1:

{-1}	interpreted_data_type?[(range(typ(typ')))](dt_const(dt(typ(typ'))))
{-2}	($\forall (l: \text{list}[\text{Byte}], a_1, a_2: \text{Address}):$ $\text{valid?}(\text{uidt}(\text{dt_const}(\text{dt}(\text{typ}(\text{typ}')))))(l, a_1) \equiv$ $\text{valid?}(\text{uidt}(\text{dt_const}(\text{dt}(\text{typ}(\text{typ}')))))(l, a_2)$ $\wedge \text{size}(\text{uidt}(\text{dt_const}(\text{dt}(\text{typ}(\text{typ}')))) > 0$
{-3}	Cpp_Type?(typ')
{-4}	const?(typ')
{-5}	volatile?(typ(typ'))
{-6}	non_bool_integral_enum?(typ(typ(typ')))
{1}	interpreted_data_type?(dt_const(dt(typ(typ')))) \wedge $(\forall (l: \text{list}[\text{Byte}], a_1, a_2: \text{Address}):$ $\text{valid?}(\text{uidt}(\text{dt_const}(\text{dt}(\text{typ}(\text{typ}')))))(l, a_1) \equiv$ $\text{valid?}(\text{uidt}(\text{dt_const}(\text{dt}(\text{typ}(\text{typ}')))))(l, a_2)$ $\wedge \text{size}(\text{uidt}(\text{dt_const}(\text{dt}(\text{typ}(\text{typ}')))) > 0$

Expanding the definition of interpreted_data_type?,

Applying propositional simplification,

we get 2 subgoals:

dt_pod.3.1.1.1:

{-1}	uninterpreted_data_type?(dt_const(dt(typ(typ'))))'uidt)
{-2}	$\forall (d: ((\text{range}(\text{typ}(\text{typ}')))), a: \text{Address}):$ $\text{valid?}(\text{uidt}(\text{dt_const}(\text{dt}(\text{typ}(\text{typ}')))))(d, a)$ $(\text{to_byte}(\text{dt_const}(\text{dt}(\text{typ}(\text{typ}')))))(d, a), a)$
{-3}	$\forall (l: \text{list}[\text{Byte}], a: \text{Address}):$ $\text{valid?}(\text{uidt}(\text{dt_const}(\text{dt}(\text{typ}(\text{typ}')))))(l, a) \equiv$ $\text{up?}(\text{from_byte}(\text{dt_const}(\text{dt}(\text{typ}(\text{typ}')))))(l, a)$
{-4}	$\forall (d: ((\text{range}(\text{typ}(\text{typ}')))), a: \text{Address}):$ $\text{down}(\text{from_byte}(\text{dt_const}(\text{dt}(\text{typ}(\text{typ}')))))(d, a), a)$ $\equiv d$
{-5}	$\forall (l: \text{list}[\text{Byte}], a_1, a_2: \text{Address}):$ $\text{valid?}(\text{uidt}(\text{dt_const}(\text{dt}(\text{typ}(\text{typ}')))))(l, a_1) \equiv$ $\text{valid?}(\text{uidt}(\text{dt_const}(\text{dt}(\text{typ}(\text{typ}')))))(l, a_2)$
{-6}	$\text{size}(\text{uidt}(\text{dt_const}(\text{dt}(\text{typ}(\text{typ}')))) > 0$
{-7}	Cpp_Type?(typ')
{-8}	const?(typ')
{-9}	volatile?(typ(typ'))
{-10}	non_bool_integral_enum?(typ(typ(typ')))
{1}	$\forall (d: ((\text{range}(\text{typ}(\text{typ}')))), a: \text{Address}):$ $\text{valid?}(\text{uidt}(\text{dt_const}(\text{dt}(\text{typ}(\text{typ}')))))(d, a), a)$

C Proof scripts

Repeatedly Skolemizing and flattening,
 Instantiating quantified variables,
 Expanding the definition of range,
 which is trivially true.

This completes the proof of `dt_pod.3.1.1.1`.

`dt_pod.3.1.1.2`:

{-1}	<code>uninterpreted_data_type?(dt_const(dt(typ(typ')))'uidt)</code>
{-2}	$\forall (d: ((\text{range}(\text{typ}(\text{typ}')))), a: \text{Address}):$ $\text{valid?}(\text{uidt}(\text{dt_const}(\text{dt}(\text{typ}(\text{typ}')))))$ $(\text{to_byte}(\text{dt_const}(\text{dt}(\text{typ}(\text{typ}'))))(d, a), a)$
{-3}	$\forall (l: \text{list}[\text{Byte}], a: \text{Address}):$ $\text{valid?}(\text{uidt}(\text{dt_const}(\text{dt}(\text{typ}(\text{typ}'))))(l, a) \equiv$ $\text{up?}(\text{from_byte}(\text{dt_const}(\text{dt}(\text{typ}(\text{typ}'))))(l, a))$
{-4}	$\forall (d: ((\text{range}(\text{typ}(\text{typ}')))), a: \text{Address}):$ $\text{down}(\text{from_byte}(\text{dt_const}(\text{dt}(\text{typ}(\text{typ}')))))$ $(\text{to_byte}(\text{dt_const}(\text{dt}(\text{typ}(\text{typ}'))))(d, a), a))$ $= d$
{-5}	$\forall (l: \text{list}[\text{Byte}], a_1, a_2: \text{Address}):$ $\text{valid?}(\text{uidt}(\text{dt_const}(\text{dt}(\text{typ}(\text{typ}'))))(l, a_1) \equiv$ $\text{valid?}(\text{uidt}(\text{dt_const}(\text{dt}(\text{typ}(\text{typ}'))))(l, a_2))$
{-6}	<code>size(uidt(dt_const(dt(typ(typ')))) > 0</code>
{-7}	<code>Cpp_Type?(typ')</code>
{-8}	<code>const?(typ')</code>
{-9}	<code>volatile?(typ(typ'))</code>
{-10}	<code>non_bool_integral_enum?(typ(typ(typ')))</code>
{1}	$\forall (d: ((\text{range}(\text{typ}(\text{typ}')))), a: \text{Address}):$ $\text{down}(\text{from_byte}(\text{dt_const}(\text{dt}(\text{typ}(\text{typ}')))))$ $(\text{to_byte}(\text{dt_const}(\text{dt}(\text{typ}(\text{typ}'))))(d, a), a))$ $= d$

Repeatedly Skolemizing and flattening,
 Instantiating quantified variables,
 Expanding the definition of range,
 which is trivially true.

This completes the proof of `dt_pod.3.1.1.2`.

`dt_pod.3.1.2`:

{-1}	$(\forall (l: \text{list}[\text{Byte}], a_1, a_2: \text{Address}):$ $\text{valid?}(\text{uidt}(\text{dt_const}(\text{dt}(\text{typ}(\text{typ}'))))(l, a_1) \equiv$ $\text{valid?}(\text{uidt}(\text{dt_const}(\text{dt}(\text{typ}(\text{typ}'))))(l, a_2))$ $\wedge \text{size}(\text{uidt}(\text{dt_const}(\text{dt}(\text{typ}(\text{typ}'))))) > 0$
{-2}	<code>Cpp_Type?(typ')</code>
{-3}	<code>const?(typ')</code>
{-4}	<code>volatile?(typ(typ'))</code>
{-5}	<code>non_bool_integral_enum?(typ(typ(typ')))</code>
{1}	$\exists (x_1: (\text{interpreted_data_type?}[(\text{range}(\text{typ}(\text{typ}')))])): \text{TRUE}$
{2}	<code>interpreted_data_type?(dt_const(dt(typ(typ'))))</code> \wedge $(\forall (l: \text{list}[\text{Byte}], a_1, a_2: \text{Address}):$ $\text{valid?}(\text{uidt}(\text{dt_const}(\text{dt}(\text{typ}(\text{typ}'))))(l, a_1) \equiv$ $\text{valid?}(\text{uidt}(\text{dt_const}(\text{dt}(\text{typ}(\text{typ}'))))(l, a_2))$ $\wedge \text{size}(\text{uidt}(\text{dt_const}(\text{dt}(\text{typ}(\text{typ}'))))) > 0$

Instantiating the top quantifier in 1 with the terms: `(dt(typ(typ!1)))`,
 which is trivially true.

This completes the proof of `dt_pod.3.1.2`.

`dt_pod.3.2`:

{-1}	<code>Cpp_Type?(typ')</code>
{-2}	<code>const?(typ')</code>
{-3}	<code>volatile?(typ(typ'))</code>
{-4}	<code>non_bool_integral_enum?(typ(typ(typ')))</code>
{1}	<code>pod_data_type?[(range(typ(typ')))](dt(typ(typ')))</code>
{2}	<code>pod_data_type?[(range(typ'))](dt_const(dt(typ(typ'))))</code>

Hiding formulas: (-2 2),

Expanding the definition of `dt`,

Using lemma `dt_volatile_pod[(range(typ(typ(typ'))))]`,

we get 3 subgoals:

`dt_pod.3.2.1`:

{-1}	<code>pod_data_type?[(range(typ(typ(typ')))](dt_volatile(dt(typ(typ(typ'))))</code>
{-2}	<code>Cpp_Type?(typ')</code>
{-3}	<code>volatile?(typ(typ'))</code>
{-4}	<code>non_bool_integral_enum?(typ(typ(typ')))</code>
{1}	<code>pod_data_type?[(range(typ(typ')))](dt_volatile(dt(typ(typ(typ'))))</code>

Expanding the definition of `pod_data_type?`,

Adding type constraints for `dt_volatile(dt(typ(typ(typ!1))))`,

we get 2 subgoals:

`dt_pod.3.2.1.1`:

{-1}	<code>interpreted_data_type?[(range(typ(typ(typ')))]</code> <code>(dt_volatile(dt(typ(typ(typ'))))</code>
{-2}	$(\forall (l: \text{list}[\text{Byte}], a_1, a_2: \text{Address}):$ <code>valid?(uidt(dt_volatile(dt(typ(typ(typ')))))(l, a_1) ≡</code> <code>valid?(uidt(dt_volatile(dt(typ(typ(typ')))))(l, a_2))</code> <code>∧ size(uidt(dt_volatile(dt(typ(typ(typ')))) > 0</code>
{-3}	<code>Cpp_Type?(typ')</code>
{-4}	<code>volatile?(typ(typ'))</code>
{-5}	<code>non_bool_integral_enum?(typ(typ(typ')))</code>
{1}	<code>interpreted_data_type?(dt_volatile(dt(typ(typ(typ')))) ∧</code> $(\forall (l: \text{list}[\text{Byte}], a_1, a_2: \text{Address}):$ <code>valid?(uidt(dt_volatile(dt(typ(typ(typ')))))(l, a_1) ≡</code> <code>valid?(uidt(dt_volatile(dt(typ(typ(typ')))))(l, a_2))</code> <code>∧ size(uidt(dt_volatile(dt(typ(typ(typ')))) > 0</code>

Expanding the definition of `interpreted_data_type?`,

Applying propositional simplification,

we get 2 subgoals:

dt_pod.3.2.1.1.1:

{-1}	uninterpreted_data_type?(dt_volatile(dt(typ(typ(typ'))))'uidt)
{-2}	$\forall (d: ((\text{range}(\text{typ}(\text{typ}(\text{typ}'))))), a: \text{Address}):$ $\text{valid?}(\text{uidt}(\text{dt_volatile}(\text{dt}(\text{typ}(\text{typ}(\text{typ}'))))))$ $(\text{to_byte}(\text{dt_volatile}(\text{dt}(\text{typ}(\text{typ}(\text{typ}'))))))(d, a), a)$
{-3}	$\forall (l: \text{list}[\text{Byte}], a: \text{Address}):$ $\text{valid?}(\text{uidt}(\text{dt_volatile}(\text{dt}(\text{typ}(\text{typ}(\text{typ}'))))))(l, a) \equiv$ $\text{up?}(\text{from_byte}(\text{dt_volatile}(\text{dt}(\text{typ}(\text{typ}(\text{typ}'))))))(l, a))$
{-4}	$\forall (d: ((\text{range}(\text{typ}(\text{typ}(\text{typ}'))))), a: \text{Address}):$ $\text{down}(\text{from_byte}(\text{dt_volatile}(\text{dt}(\text{typ}(\text{typ}(\text{typ}'))))))$ $(\text{to_byte}(\text{dt_volatile}(\text{dt}(\text{typ}(\text{typ}(\text{typ}'))))))(d, a), a))$
{-5}	$\stackrel{= d}{\forall} (l: \text{list}[\text{Byte}], a_1, a_2: \text{Address}):$ $\text{valid?}(\text{uidt}(\text{dt_volatile}(\text{dt}(\text{typ}(\text{typ}(\text{typ}'))))))(l, a_1) \equiv$ $\text{valid?}(\text{uidt}(\text{dt_volatile}(\text{dt}(\text{typ}(\text{typ}(\text{typ}'))))))(l, a_2)$
{-6}	$\text{size}(\text{uidt}(\text{dt_volatile}(\text{dt}(\text{typ}(\text{typ}(\text{typ}')))))) > 0$
{-7}	Cpp_Type?(typ')
{-8}	volatile?(typ(typ'))
{-9}	non_bool_integral_enum?(typ(typ(typ')))
{1}	$\forall (d: ((\text{range}(\text{typ}(\text{typ}(\text{typ}'))))), a: \text{Address}):$ $\text{valid?}(\text{uidt}(\text{dt_volatile}(\text{dt}(\text{typ}(\text{typ}(\text{typ}'))))))$ $(\text{to_byte}(\text{dt_volatile}(\text{dt}(\text{typ}(\text{typ}(\text{typ}'))))))(d, a), a)$

Repeatedly Skolemizing and flattening,
Instantiating quantified variables,
Expanding the definition of range,
which is trivially true.

This completes the proof of dt_pod.3.2.1.1.1.

dt_pod.3.2.1.1.2:

{-1}	uninterpreted_data_type?(dt_volatile(dt(typ(typ(typ'))))'uidt)
{-2}	$\forall (d: ((\text{range}(\text{typ}(\text{typ}(\text{typ}'))))), a: \text{Address}):$ $\text{valid?}(\text{uidt}(\text{dt_volatile}(\text{dt}(\text{typ}(\text{typ}(\text{typ}'))))))$ $(\text{to_byte}(\text{dt_volatile}(\text{dt}(\text{typ}(\text{typ}(\text{typ}'))))))(d, a), a)$
{-3}	$\forall (l: \text{list}[\text{Byte}], a: \text{Address}):$ $\text{valid?}(\text{uidt}(\text{dt_volatile}(\text{dt}(\text{typ}(\text{typ}(\text{typ}'))))))(l, a) \equiv$ $\text{up?}(\text{from_byte}(\text{dt_volatile}(\text{dt}(\text{typ}(\text{typ}(\text{typ}'))))))(l, a))$
{-4}	$\forall (d: ((\text{range}(\text{typ}(\text{typ}(\text{typ}'))))), a: \text{Address}):$ $\text{down}(\text{from_byte}(\text{dt_volatile}(\text{dt}(\text{typ}(\text{typ}(\text{typ}'))))))$ $(\text{to_byte}(\text{dt_volatile}(\text{dt}(\text{typ}(\text{typ}(\text{typ}'))))))(d, a), a))$
{-5}	$\stackrel{= d}{\forall} (l: \text{list}[\text{Byte}], a_1, a_2: \text{Address}):$ $\text{valid?}(\text{uidt}(\text{dt_volatile}(\text{dt}(\text{typ}(\text{typ}(\text{typ}'))))))(l, a_1) \equiv$ $\text{valid?}(\text{uidt}(\text{dt_volatile}(\text{dt}(\text{typ}(\text{typ}(\text{typ}'))))))(l, a_2)$
{-6}	$\text{size}(\text{uidt}(\text{dt_volatile}(\text{dt}(\text{typ}(\text{typ}(\text{typ}')))))) > 0$
{-7}	Cpp_Type?(typ')
{-8}	volatile?(typ(typ'))
{-9}	non_bool_integral_enum?(typ(typ(typ')))
{1}	$\forall (d: ((\text{range}(\text{typ}(\text{typ}(\text{typ}'))))), a: \text{Address}):$ $\text{down}(\text{from_byte}(\text{dt_volatile}(\text{dt}(\text{typ}(\text{typ}(\text{typ}'))))))$ $(\text{to_byte}(\text{dt_volatile}(\text{dt}(\text{typ}(\text{typ}(\text{typ}'))))))(d, a), a)$ $= d$

Repeatedly Skolemizing and flattening,
Instantiating quantified variables,

Expanding the definition of range,
which is trivially true.

This completes the proof of `dt_pod.3.2.1.1.2`.

`dt_pod.3.2.1.2`:

<pre>{-1} (∀ (l: list[Byte], a1, a2: Address): valid?(uidt(dt_volatile(dt(typ(typ(typ'))))))(l, a1) ≡ valid?(uidt(dt_volatile(dt(typ(typ(typ'))))))(l, a2)) ∧ size(uidt(dt_volatile(dt(typ(typ(typ')))))) > 0 {-2} Cpp_Type?(typ') {-3} volatile?(typ(typ')) {-4} non_bool_integral_enum?(typ(typ(typ')))</pre>	<pre>{1} ∃ (x1: (interpreted_data_type?[(range(typ(typ(typ')))])): TRUE {2} interpreted_data_type?(dt_volatile(dt(typ(typ(typ'))))) ∧ (∀ (l: list[Byte], a1, a2: Address): valid?(uidt(dt_volatile(dt(typ(typ(typ'))))))(l, a1) ≡ valid?(uidt(dt_volatile(dt(typ(typ(typ'))))))(l, a2)) ∧ size(uidt(dt_volatile(dt(typ(typ(typ')))))) > 0</pre>
--	--

Instantiating the top quantifier in 1 with the terms: `(dt(typ(typ(typ!1))))`,
which is trivially true.

This completes the proof of `dt_pod.3.2.1.2`.

`dt_pod.3.2.2`:

<pre>{-1} Cpp_Type?(typ') {-2} volatile?(typ(typ')) {-3} non_bool_integral_enum?(typ(typ(typ')))</pre>	<pre>{1} pod_data_type?[(range(typ(typ(typ')))](dt(typ(typ(typ')))) {2} pod_data_type?[(range(typ(typ(typ')))](dt_volatile(dt(typ(typ(typ')))))</pre>
---	---

Hiding formulas: `(-2 2)`,

Expanding the definition of `dt`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
we get 2 subgoals:

`dt_pod.3.2.2.1`:

<pre>{-1} const?(typ(typ(typ')) {-2} Cpp_Type?(typ') {-3} non_bool_integral_enum?(typ(typ(typ')))</pre>	<pre>{1} pod_data_type?[(range(typ(typ(typ')))](dt_const(dt(typ(typ(typ')))))</pre>
--	--

Keeping `(-1 -3)` and hiding `*`,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `dt_pod.3.2.2.1`.

`dt_pod.3.2.2.2`:

<pre>{-1} volatile?(typ(typ(typ')) {-2} Cpp_Type?(typ') {-3} non_bool_integral_enum?(typ(typ(typ')))</pre>	<pre>{1} pod_data_type?[(range(typ(typ(typ')))] (dt_volatile(dt(typ(typ(typ')))))</pre>
---	--

Keeping `(-1 -3)` and hiding `*`,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `dt_pod.3.2.2.2`.

C Proof scripts

dt_pod.3.2.3:

{-1}	Cpp_Type?(typ')
{-2}	volatile?(typ(typ'))
{-3}	non_bool_integral_enum?(typ(typ(typ')))
{1}	$\exists (x_1 : (\text{interpreted_data_type?}[(\text{range}(\text{typ}(\text{typ}(\text{typ}')))]))): \text{TRUE}$
{2}	pod_data_type?[(range(typ(typ')))](dt_volatile(dt(typ(typ(typ')))))

Instantiating the top quantifier in 1 with the terms: (dt(typ(typ(typ!1)))),

which is trivially true.

This completes the proof of dt_pod.3.2.3.

dt_pod.3.3:

{-1}	Cpp_Type?(typ')
{-2}	const?(typ')
{-3}	volatile?(typ(typ'))
{-4}	non_bool_integral_enum?(typ(typ(typ')))
{1}	$\exists (x_1 : (\text{interpreted_data_type?}[(\text{range}(\text{typ}(\text{typ}')))]))): \text{TRUE}$
{2}	pod_data_type?[(range(typ'))](dt_const(dt(typ(typ'))))

Instantiating the top quantifier in 1 with the terms: (dt(typ(typ!1))),

which is trivially true.

This completes the proof of dt_pod.3.3.

dt_pod.4:

{-1}	Cpp_Type?(typ')
{-2}	non_bool_integral_enum?(typ(typ'))
{-3}	volatile?(typ')
{1}	pod_data_type?[(range(typ'))](dt(typ'))

Expanding the definition of dt,

Using lemma dt_volatile_pod[(range(typ(typ')))],

we get 3 subgoals:

dt_pod.4.1:

{-1}	pod_data_type?[(range(typ(typ')))](dt_volatile(dt(typ(typ'))))
{-2}	Cpp_Type?(typ')
{-3}	non_bool_integral_enum?(typ(typ'))
{-4}	volatile?(typ')
{1}	pod_data_type?[(range(typ'))](dt_volatile(dt(typ(typ'))))

Expanding the definition of pod_data_type?,

Adding type constraints for dt_volatile(dt(typ(typ!1))),

we get 2 subgoals:

dt_pod.4.1.1.1:

{-1}	interpreted_data_type?[(range(typ(typ')))](dt_volatile(dt(typ(typ'))))
{-2}	($\forall (l: \text{list}[\text{Byte}], a_1, a_2: \text{Address}):$ $\text{valid?}(\text{uidt}(\text{dt_volatile}(\text{dt}(\text{typ}(\text{typ}')))))(l, a_1) \equiv$ $\text{valid?}(\text{uidt}(\text{dt_volatile}(\text{dt}(\text{typ}(\text{typ}')))))(l, a_2)$ $\wedge \text{size}(\text{uidt}(\text{dt_volatile}(\text{dt}(\text{typ}(\text{typ}'))))) > 0$
{-3}	Cpp_Type?(typ')
{-4}	non_bool_integral_enum?(typ(typ'))
{-5}	volatile?(typ')
{1}	interpreted_data_type?(dt_volatile(dt(typ(typ')))) \wedge $(\forall (l: \text{list}[\text{Byte}], a_1, a_2: \text{Address}):$ $\text{valid?}(\text{uidt}(\text{dt_volatile}(\text{dt}(\text{typ}(\text{typ}')))))(l, a_1) \equiv$ $\text{valid?}(\text{uidt}(\text{dt_volatile}(\text{dt}(\text{typ}(\text{typ}')))))(l, a_2)$ $\wedge \text{size}(\text{uidt}(\text{dt_volatile}(\text{dt}(\text{typ}(\text{typ}'))))) > 0$

Expanding the definition of interpreted_data_type?,

Applying propositional simplification,

we get 2 subgoals:

dt_pod.4.1.1.1.1:

{-1}	uninterpreted_data_type?(dt_volatile(dt(typ(typ')))' uidt)
{-2}	$\forall (d: ((\text{range}(\text{typ}(\text{typ}')))), a: \text{Address}):$ $\text{valid?}(\text{uidt}(\text{dt_volatile}(\text{dt}(\text{typ}(\text{typ}')))))($ $\text{to_byte}(\text{dt_volatile}(\text{dt}(\text{typ}(\text{typ}'))))(d, a), a)$
{-3}	$\forall (l: \text{list}[\text{Byte}], a: \text{Address}):$ $\text{valid?}(\text{uidt}(\text{dt_volatile}(\text{dt}(\text{typ}(\text{typ}')))))(l, a) \equiv$ $\text{up?}(\text{from_byte}(\text{dt_volatile}(\text{dt}(\text{typ}(\text{typ}')))))(l, a)$
{-4}	$\forall (d: ((\text{range}(\text{typ}(\text{typ}')))), a: \text{Address}):$ $\text{down}(\text{from_byte}(\text{dt_volatile}(\text{dt}(\text{typ}(\text{typ}')))))($ $\text{to_byte}(\text{dt_volatile}(\text{dt}(\text{typ}(\text{typ}'))))(d, a), a)$
{-5}	$\forall (l: \text{list}[\text{Byte}], a_1, a_2: \text{Address}):$ $\text{valid?}(\text{uidt}(\text{dt_volatile}(\text{dt}(\text{typ}(\text{typ}')))))(l, a_1) \equiv$ $\text{valid?}(\text{uidt}(\text{dt_volatile}(\text{dt}(\text{typ}(\text{typ}')))))(l, a_2)$
{-6}	$\text{size}(\text{uidt}(\text{dt_volatile}(\text{dt}(\text{typ}(\text{typ}'))))) > 0$
{-7}	Cpp_Type?(typ')
{-8}	non_bool_integral_enum?(typ(typ'))
{-9}	volatile?(typ')
{1}	$\forall (d: ((\text{range}(\text{typ}(\text{typ}')))), a: \text{Address}):$ $\text{valid?}(\text{uidt}(\text{dt_volatile}(\text{dt}(\text{typ}(\text{typ}')))))($ $\text{to_byte}(\text{dt_volatile}(\text{dt}(\text{typ}(\text{typ}'))))(d, a), a)$

Repeatedly Skolemizing and flattening,

Instantiating quantified variables,

Expanding the definition of range,

which is trivially true.

This completes the proof of dt_pod.4.1.1.1.

dt_pod.4.1.1.2:

{-1}	uninterpreted_data_type?(dt_volatile(dt(typ(typ')))'uidt)
{-2}	$\forall (d: ((\text{range}(\text{typ}(\text{typ}')))), a: \text{Address}):$ $\text{valid?}(\text{uidt}(\text{dt_volatile}(\text{dt}(\text{typ}(\text{typ}')))))$ $(\text{to_byte}(\text{dt_volatile}(\text{dt}(\text{typ}(\text{typ}'))))(d, a), a)$
{-3}	$\forall (l: \text{list}[\text{Byte}], a: \text{Address}):$ $\text{valid?}(\text{uidt}(\text{dt_volatile}(\text{dt}(\text{typ}(\text{typ}')))))(l, a) \equiv$ $\text{up?}(\text{from_byte}(\text{dt_volatile}(\text{dt}(\text{typ}(\text{typ}'))))(l, a))$
{-4}	$\forall (d: ((\text{range}(\text{typ}(\text{typ}')))), a: \text{Address}):$ $\text{down}(\text{from_byte}(\text{dt_volatile}(\text{dt}(\text{typ}(\text{typ}')))))$ $(\text{to_byte}(\text{dt_volatile}(\text{dt}(\text{typ}(\text{typ}'))))(d, a), a))$ $= d$
{-5}	$\forall (l: \text{list}[\text{Byte}], a_1, a_2: \text{Address}):$ $\text{valid?}(\text{uidt}(\text{dt_volatile}(\text{dt}(\text{typ}(\text{typ}')))))(l, a_1) \equiv$ $\text{valid?}(\text{uidt}(\text{dt_volatile}(\text{dt}(\text{typ}(\text{typ}')))))(l, a_2)$
{-6}	$\text{size}(\text{uidt}(\text{dt_volatile}(\text{dt}(\text{typ}(\text{typ}'))))) > 0$
{-7}	Cpp_Type?(typ')
{-8}	non_bool_integral_enum?(typ(typ'))
{-9}	volatile?(typ')
{1}	$\forall (d: ((\text{range}(\text{typ}(\text{typ}')))), a: \text{Address}):$ $\text{down}(\text{from_byte}(\text{dt_volatile}(\text{dt}(\text{typ}(\text{typ}')))))$ $(\text{to_byte}(\text{dt_volatile}(\text{dt}(\text{typ}(\text{typ}'))))(d, a), a))$ $= d$

Repeatedly Skolemizing and flattening,
Instantiating quantified variables,
Expanding the definition of range,
which is trivially true.

This completes the proof of dt_pod.4.1.1.2.

dt_pod.4.1.2:

{-1}	$(\forall (l: \text{list}[\text{Byte}], a_1, a_2: \text{Address}):$ $\text{valid?}(\text{uidt}(\text{dt_volatile}(\text{dt}(\text{typ}(\text{typ}')))))(l, a_1) \equiv$ $\text{valid?}(\text{uidt}(\text{dt_volatile}(\text{dt}(\text{typ}(\text{typ}')))))(l, a_2))$ $\wedge \text{size}(\text{uidt}(\text{dt_volatile}(\text{dt}(\text{typ}(\text{typ}'))))) > 0$
{-2}	Cpp_Type?(typ')
{-3}	non_bool_integral_enum?(typ(typ'))
{-4}	volatile?(typ')
{1}	$\exists (x_1: (\text{interpreted_data_type?}[(\text{range}(\text{typ}(\text{typ}')))])): \text{TRUE}$
{2}	$\text{interpreted_data_type?}(\text{dt_volatile}(\text{dt}(\text{typ}(\text{typ}')))) \wedge$ $(\forall (l: \text{list}[\text{Byte}], a_1, a_2: \text{Address}):$ $\text{valid?}(\text{uidt}(\text{dt_volatile}(\text{dt}(\text{typ}(\text{typ}')))))(l, a_1) \equiv$ $\text{valid?}(\text{uidt}(\text{dt_volatile}(\text{dt}(\text{typ}(\text{typ}')))))(l, a_2))$ $\wedge \text{size}(\text{uidt}(\text{dt_volatile}(\text{dt}(\text{typ}(\text{typ}'))))) > 0$

Instantiating the top quantifier in 1 with the terms: (dt(typ(typ!))),
which is trivially true.

This completes the proof of dt_pod.4.1.2.

dt_pod.4.2:

{-1}	Cpp_Type?(typ')
{-2}	non_bool_integral_enum?(typ(typ'))
{-3}	volatile?(typ')
{1}	$\text{pod_data_type?}[(\text{range}(\text{typ}(\text{typ}')))](\text{dt}(\text{typ}(\text{typ}')))$
{2}	$\text{pod_data_type?}[(\text{range}(\text{typ}(\text{typ}')))](\text{dt_volatile}(\text{dt}(\text{typ}(\text{typ}'))))$

Hiding formulas: (-3 2),

Expanding the definition of dt,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 2 subgoals:

dt_pod.4.2.1:

$$\frac{\begin{array}{l} \{-1\} \text{ const?}(\text{typ}(\text{typ}')) \\ \{-2\} \text{ Cpp_Type?}(\text{typ}')) \\ \{-3\} \text{ non_bool_integral_enum?}(\text{typ}(\text{typ}')) \end{array}}{\{1\} \text{ pod_data_type?}[(\text{range}(\text{typ}(\text{typ}')))](\text{dt_const}(\text{dt}(\text{typ}(\text{typ}'))))}$$

Keeping (-1 -3) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of dt_pod.4.2.1.

dt_pod.4.2.2:

$$\frac{\begin{array}{l} \{-1\} \text{ volatile?}(\text{typ}(\text{typ}')) \\ \{-2\} \text{ Cpp_Type?}(\text{typ}')) \\ \{-3\} \text{ non_bool_integral_enum?}(\text{typ}(\text{typ}')) \end{array}}{\{1\} \text{ pod_data_type?}[(\text{range}(\text{typ}(\text{typ}')))](\text{dt_volatile}(\text{dt}(\text{typ}(\text{typ}'))))}$$

Keeping (-1 -3) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of dt_pod.4.2.2.

dt_pod.4.3:

$$\frac{\begin{array}{l} \{-1\} \text{ Cpp_Type?}(\text{typ}')) \\ \{-2\} \text{ non_bool_integral_enum?}(\text{typ}(\text{typ}')) \\ \{-3\} \text{ volatile?}(\text{typ}')) \end{array}}{\begin{array}{l} \{1\} \exists (x_1: (\text{interpreted_data_type?}[(\text{range}(\text{typ}(\text{typ}')))])): \text{TRUE} \\ \{2\} \text{ pod_data_type?}[(\text{range}(\text{typ}(\text{typ}')))](\text{dt_volatile}(\text{dt}(\text{typ}(\text{typ}')))) \end{array}}$$

Instantiating the top quantifier in 1 with the terms: (dt(typ(typ!1))),
which is trivially true.

This completes the proof of dt_pod.4.3.

Q.E.D.

C.44.39 Cpp_Types.dt_cv_pointer_TCC1

Terse proof for dt_cv_pointer_TCC1.

dt_cv_pointer_TCC1:

$$\frac{}{\{1\} \text{ well_founded?}(\lambda (x: \text{Cpp_Subtype}(\text{cv}(\text{pointer?})), y: \text{Cpp_Subtype}(\text{cv}(\text{pointer?}))) : \text{restrict} \quad \begin{array}{l} [[\text{Cpp_Type_}, \text{Cpp_Type_}], \\ [\text{Cpp_Subtype}(\text{cv}(\text{pointer?})), \text{Cpp_Subtype}(\text{cv}(\text{pointer?}))], \text{boolean}] \\ (\ll)(x, y) \end{array})}$$

Using lemma well_founded_restrict[Cpp_Type_, Cpp_Subtype(cv(pointer?))],

Expanding the definition of restrict,

which is trivially true.

This completes the proof of dt_cv_pointer_TCC1.

Q.E.D.

C.44.40 Cpp_Types.dt_cv_pointer_TCC2

Terse proof for dt_cv_pointer_TCC2.

dt_cv_pointer_TCC2:

{1}	\forall (typ: Cpp_Subtype(cv(pointer?)), t: Cpp_Type_): typ = const(t) \supset Cpp_Type?(t) \wedge cv(pointer?)(t)
-----	---

Repeatedly Skolemizing and flattening,

Replacing using formula -3,

Applying propositional simplification,

we get 2 subgoals:

dt_cv_pointer_TCC2.1:

{-1}	Cpp_Type?(const(t'))
{-2}	cv(pointer?)(const(t'))
{-3}	typ' = const(t')
{1}	Cpp_Type?(t')

Expanding the definition of Cpp_Type?,

Repeatedly Skolemizing and flattening,

Instantiating the top quantifier in -2 with the terms: (t!2),

Expanding the definition of subterm,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of dt_cv_pointer_TCC2.1.

dt_cv_pointer_TCC2.2:

{-1}	Cpp_Type?(const(t'))
{-2}	cv(pointer?)(const(t'))
{-3}	typ' = const(t')
{1}	cv(pointer?)(t')

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of dt_cv_pointer_TCC2.2.

Q.E.D.

C.44.41 Cpp_Types.dt_cv_pointer_TCC3

Terse proof for dt_cv_pointer_TCC3.

dt_cv_pointer_TCC3:

{1}	\forall (typ: Cpp_Subtype(cv(pointer?)), t: Cpp_Type_): typ = const(t) \supset restrict [[Cpp_Type_, Cpp_Type_], [Cpp_Subtype(cv(pointer?)), Cpp_Subtype(cv(pointer?))], boolean] (\ll)(t, typ)
-----	--

Repeatedly Skolemizing and flattening,

Expanding the definition of restrict,

Replacing using formula -3,

Expanding the definition of ■,

which is trivially true.

This completes the proof of dt_cv_pointer_TCC3.

Q.E.D.

C.44.42 Cpp_Types.dt_cv_pointer_TCC4

Terse proof for dt_cv_pointer_TCC4.

dt_cv_pointer_TCC4:

$$\frac{}{\{1\} \quad \forall (\text{typ}: \text{Cpp_Subtype}(\text{cv}(\text{pointer?})), t: \text{Cpp_Type_}): \text{typ} = \text{const}(t) \supset \text{Cpp_Type?}(t)}$$

Repeatedly Skolemizing and flattening,

Using lemma dt_cv_pointer_TCC2,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of dt_cv_pointer_TCC4.

Q.E.D.

C.44.43 Cpp_Types.dt_cv_pointer_TCC5

Terse proof for dt_cv_pointer_TCC5.

dt_cv_pointer_TCC5:

$$\frac{}{\{1\} \quad \forall (\text{typ}: \text{Cpp_Subtype}(\text{cv}(\text{pointer?})), t: \text{Cpp_Type_}): \text{typ} = \text{const}(t) \supset \text{pointer?}(\text{cv_base}(t))}$$

Repeatedly Skolemizing and flattening,

Replacing using formula -3,

Using lemma cv_base_result,

we get 2 subgoals:

dt_cv_pointer_TCC5.1:

$$\frac{\begin{array}{l} \{-1\} \quad \text{Cpp_Type?}(\text{const}(t')) \\ \{-2\} \quad \text{cv}(\text{pointer?})(\text{const}(t')) \\ \{-3\} \quad \text{typ}' = \text{const}(t') \end{array}}{\begin{array}{l} \{1\} \quad \text{cv}(\text{pointer?})(t') \\ \{2\} \quad \text{pointer?}(\text{cv_base}(t')) \end{array}}$$

Using lemma dt_cv_pointer_TCC2,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of dt_cv_pointer_TCC5.1.

dt_cv_pointer_TCC5.2:

$$\frac{\begin{array}{l} \{-1\} \quad \text{Cpp_Type?}(\text{const}(t')) \\ \{-2\} \quad \text{cv}(\text{pointer?})(\text{const}(t')) \\ \{-3\} \quad \text{typ}' = \text{const}(t') \end{array}}{\begin{array}{l} \{1\} \quad (\text{pointer?} \subseteq \text{interpreted?}) \\ \{2\} \quad \text{pointer?}(\text{cv_base}(t')) \end{array}}$$

Keeping (1) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of dt_cv_pointer_TCC5.2.

Q.E.D.

C.44.44 Cpp_Types.dt_cv_pointer_TCC6

Terse proof for dt_cv_pointer_TCC6.

dt_cv_pointer_TCC6:

{1}	\forall (typ: Cpp_Subtype(cv(pointer?)), t: Cpp_Type_): typ = const(t) \supset $(\exists (x_1: (\text{interpreted_data_type?}[\text{((range_pointer(cv_base(t)))]))}): \text{TRUE})$
-----	--

Repeatedly Skolemizing and flattening,

Instantiating the top quantifier in 1 with the terms: (dt_pointer(cv_base(t!1))),

we get 3 subgoals:

dt_cv_pointer_TCC6.1:

{-1}	Cpp_Type?(typ')
{-2}	cv(pointer?)(typ')
{-3}	typ' = const(t')
{1}	TRUE

which is trivially true.

This completes the proof of dt_cv_pointer_TCC6.1.

dt_cv_pointer_TCC6.2:

{-1}	Cpp_Type?(typ')
{-2}	cv(pointer?)(typ')
{-3}	typ' = const(t')
{1}	pointer?(cv_base(t'))

Using lemma dt_cv_pointer_TCC5,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of dt_cv_pointer_TCC6.2.

dt_cv_pointer_TCC6.3:

{-1}	Cpp_Type?(typ')
{-2}	cv(pointer?)(typ')
{-3}	typ' = const(t')
{1}	Cpp_Type?(t')

Using lemma dt_cv_pointer_TCC4,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of dt_cv_pointer_TCC6.3.

Q.E.D.

C.44.45 Cpp_Types.dt_cv_pointer_TCC7

Terse proof for dt_cv_pointer_TCC7.

dt_cv_pointer_TCC7:

```

{1}  ∀ (typ: Cpp_Subtype(cv(pointer?)),
      v:
        [typ1:
          {z: Cpp_Subtype(cv(pointer?)) |
            restrict
              [[Cpp_Type_, Cpp_Type_],
               [Cpp_Subtype(cv(pointer?)), Cpp_Subtype(cv(pointer?))],
               boolean]
              (≪)(z, typ)} →
            (interpreted_data_type?[(range_pointer(cv_base(typ1)))]),
          t: Cpp_Type_):
  typ = const(t) ⊃
  (∀ (x: lift[Pointer_Base_Type]):
    range_pointer(cv_base(t))(x) ≡ range_pointer(cv_base(typ))(x))
  ∧
  (∀ (x: lift[Pointer_Base_Type]):
    range_pointer(cv_base(t))(x) ≡ range_pointer(cv_base(typ))(x))
  ∧
  (∀ (x₁: [list[Byte], Address]):
    every[[(range_pointer(cv_base(t)))]
           ((range_pointer(cv_base(typ))))
           (dt_const[[(range_pointer(cv_base(t)))](v(t))'from_byte(x₁))]]
  ∧
  interpreted_data_type?[(range_pointer(cv_base(typ)))]
  (dt_const[[(range_pointer(cv_base(t)))](v(t))])

```

Repeatedly Skolemizing and flattening,

Replacing using formula -3,

Expanding the definition of cv_base,

Applying propositional simplification,

we get 2 subgoals:

dt_cv_pointer_TCC7.1:

```

{-1} Cpp_Type?(const(t'))
{-2} cv(pointer?)(const(t'))
{-3} typ' = const(t')


---


{1}  ∀ (x₁: [list[Byte], Address]):
      every[[(range_pointer(cv_base(t')))]
             ((range_pointer(cv_base(const(t'))))]
             (dt_const[[(range_pointer(cv_base(t')))](v'(t'))'from_byte(x₁))]]

```

Repeatedly Skolemizing and flattening,

Expanding the definition of every,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Expanding the definition of cv_base,

which is trivially true.

This completes the proof of dt_cv_pointer_TCC7.1.

C Proof scripts

dt_cv_pointer_TCC7.2:

{-1}	Cpp_Type?(const(t'))
{-2}	cv(pointer?)(const(t'))
{-3}	typ' = const(t')
{1}	interpreted_data_type?[((range_pointer(cv_base(typ'))))] (dt_const[((range_pointer(cv_base(t')))]($v'(t'))$

Adding type constraints for dt_const($v!1(t!1)$),

we get 3 subgoals:

dt_cv_pointer_TCC7.2.1:

{-1}	interpreted_data_type?[((range_pointer(cv_base(t')))](dt_const($v'(t'))$
{-2}	Cpp_Type?(const(t'))
{-3}	cv(pointer?)(const(t'))
{-4}	typ' = const(t')
{1}	interpreted_data_type?[((range_pointer(cv_base(typ'))))] (dt_const[((range_pointer(cv_base(t')))]($v'(t'))$

Expanding the definition of interpreted_data_type?,

Applying propositional simplification,

we get 2 subgoals:

dt_cv_pointer_TCC7.2.1.1:

{-1}	uninterpreted_data_type?(dt_const($v'(t'))'uidt)$
{-2}	$\forall (d: ((range_pointer(cv_base(t')))) , a: Address):$
	valid?(uidt(dt_const($v'(t'))))(to_byte(dt_const(v'(t')))(d, a, a)$
{-3}	$\forall (l: list[Byte] , a: Address):$
	valid?(uidt(dt_const($v'(t'))))(l, a \equiv$
	up?(from_byte(dt_const($v'(t')))(l, a)$
{-4}	$\forall (d: ((range_pointer(cv_base(t')))) , a: Address):$
	down(from_byte(dt_const($v'(t')))(to_byte(dt_const(v'(t')))(d, a, a)) =$
	d
{-5}	Cpp_Type?(const(t'))
{-6}	cv(pointer?)(const(t'))
{-7}	typ' = const(t')
{1}	$\forall (d: ((range_pointer(cv_base(typ')))) , a: Address):$
	valid?(uidt(dt_const[((range_pointer(cv_base(t')))]($v'(t'))$
	(to_byte(dt_const[((range_pointer(cv_base(t')))]($v'(t')))(d, a, a)$

Repeatedly Skolemizing and flattening,

Instantiating quantified variables,

Replacing using formula -7,

Expanding the definition of cv_base,

which is trivially true.

This completes the proof of dt_cv_pointer_TCC7.2.1.1.

dt_cv_pointer_TCC7.2.1.2:

{-1}	uninterpreted_data_type?(dt_const(v'(t'))'uidt)
{-2}	$\forall (d: ((\text{range_pointer}(\text{cv_base}(t')))), a: \text{Address}):$ valid?(uidt(dt_const(v'(t')))(to_byte(dt_const(v'(t')))(d, a), a)
{-3}	$\forall (l: \text{list}[\text{Byte}], a: \text{Address}):$ valid?(uidt(dt_const(v'(t')))(l, a) \equiv up?(from_byte(dt_const(v'(t')))(l, a)
{-4}	$\forall (d: ((\text{range_pointer}(\text{cv_base}(t')))), a: \text{Address}):$ down(from_byte(dt_const(v'(t')))(to_byte(dt_const(v'(t')))(d, a), a) =
{-5}	$\overset{d}{\text{Cpp_Type?}}(\text{const}(t'))$
{-6}	cv(pointer?)(const(t'))
{-7}	typ' = const(t')
{1} $\forall (d: ((\text{range_pointer}(\text{cv_base}(\text{typ}')))), a: \text{Address}):$ down(from_byte(dt_const[[(range_pointer(cv_base(t')))](v'(t'))] (to_byte(dt_const[[(range_pointer(cv_base(t')))](v'(t'))] (d, a), a)) = d	

Repeatedly Skolemizing and flattening,

Instantiating quantified variables,

Replacing using formula -7,

Expanding the definition of cv_base,

which is trivially true.

This completes the proof of dt_cv_pointer_TCC7.2.1.2.

dt_cv_pointer_TCC7.2.2:

{-1}	Cpp_Type?(const(t'))
{-2}	cv(pointer?)(const(t'))
{-3}	typ' = const(t')
{1} $\exists (x_1: (\text{interpreted_data_type?}[\text{[(range_pointer}(\text{cv_base}(t'))])])): \text{TRUE}$	
{2}	interpreted_data_type?[\text{[(range_pointer}(\text{cv_base}(\text{typ}')))] (dt_const[\text{[(range_pointer}(\text{cv_base}(t')))](v'(t'))]

Using lemma dt_cv_pointer_TCC6,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of dt_cv_pointer_TCC7.2.2.

dt_cv_pointer_TCC7.2.3:

{-1}	Cpp_Type?(const(t'))
{-2}	cv(pointer?)(const(t'))
{-3}	typ' = const(t')
{1} pointer?(cv_base(t'))	
{2}	interpreted_data_type?[\text{[(range_pointer}(\text{cv_base}(\text{typ}')))] (dt_const[\text{[(range_pointer}(\text{cv_base}(t')))](v'(t'))]

Using lemma dt_cv_pointer_TCC5,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of dt_cv_pointer_TCC7.2.3.

Q.E.D.

C.44.46 Cpp_Types.dt_cv_pointer_TCC8

Terse proof for dt_cv_pointer_TCC8.

dt_cv_pointer_TCC8:

{1}	\forall (typ: Cpp_Subtype(cv(pointer?)), t: Cpp_Type_): typ = volatile(t) \supset Cpp_Type?(t) \wedge cv(pointer?)(t)
-----	--

Repeatedly Skolemizing and flattening,
Replacing using formula -3,
Applying propositional simplification,
we get 2 subgoals:

dt_cv_pointer_TCC8.1:

{-1}	Cpp_Type?(volatile(t'))
{-2}	cv(pointer?)(volatile(t'))
{-3}	typ' = volatile(t')
{1}	Cpp_Type?(t')

Expanding the definition of Cpp_Type?,
Repeatedly Skolemizing and flattening,
Instantiating the top quantifier in -2 with the terms: (t!2),
Expanding the definition of subterm,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of dt_cv_pointer_TCC8.1.

dt_cv_pointer_TCC8.2:

{-1}	Cpp_Type?(volatile(t'))
{-2}	cv(pointer?)(volatile(t'))
{-3}	typ' = volatile(t')
{1}	cv(pointer?)(t')

Keeping (-2 1) and hiding *,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of dt_cv_pointer_TCC8.2.
Q.E.D.

C.44.47 Cpp_Types.dt_cv_pointer_TCC9

Terse proof for dt_cv_pointer_TCC9.

dt_cv_pointer_TCC9:

{1}	\forall (typ: Cpp_Subtype(cv(pointer?)), t: Cpp_Type_): typ = volatile(t) \supset restrict [[Cpp_Type_, Cpp_Type_], [Cpp_Subtype(cv(pointer?)), Cpp_Subtype(cv(pointer?))], boolean] (\ll)(t, typ)
-----	---

Expanding the definition of restrict,
Repeatedly Skolemizing and flattening,
Replacing using formula -3,
Expanding the definition of ■,
which is trivially true.

This completes the proof of dt_cv_pointer_TCC9.
Q.E.D.

C.44.48 Cpp_Types.dt_cv_pointer_TCC10

Terse proof for dt_cv_pointer_TCC10.

dt_cv_pointer_TCC10:

{1}	$\forall (\text{typ}: \text{Cpp_Subtype}(\text{cv}(\text{pointer?})), t: \text{Cpp_Type_}): \text{typ} = \text{volatile}(t) \supset \text{Cpp_Type?}(t)$
-----	--

Repeatedly Skolemizing and flattening,
Using lemma dt_cv_pointer_TCC8,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of dt_cv_pointer_TCC10.
Q.E.D.

C.44.49 Cpp_Types.dt_cv_pointer_TCC11

Terse proof for dt_cv_pointer_TCC11.

dt_cv_pointer_TCC11:

{1}	$\forall (\text{typ}: \text{Cpp_Subtype}(\text{cv}(\text{pointer?})), t: \text{Cpp_Type_}):$ $\text{typ} = \text{volatile}(t) \supset \text{pointer?}(\text{cv_base}(t))$
-----	--

Repeatedly Skolemizing and flattening,
Using lemma dt_cv_pointer_TCC8,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Rewriting using cv_base_result, matching in *,
Keeping (1) and hiding *,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of dt_cv_pointer_TCC11.
Q.E.D.

C.44.50 Cpp_Types.dt_cv_pointer_TCC12

Terse proof for dt_cv_pointer_TCC12.

dt_cv_pointer_TCC12:

{1}	$\forall (\text{typ}: \text{Cpp_Subtype}(\text{cv}(\text{pointer?})), t: \text{Cpp_Type_}):$ $\text{typ} = \text{volatile}(t) \supset$ $(\exists (x_1: (\text{interpreted_data_type?}[\text{((range_pointer}(\text{cv_base}(t))))]))): \text{TRUE}$
-----	--

Repeatedly Skolemizing and flattening,
Instantiating the top quantifier in 1 with the terms: (dt_pointer(cv_base(t!1))),
we get 3 subgoals:

dt_cv_pointer_TCC12.1:

{-1}	Cpp_Type?(typ')
{-2}	cv(pointer?)(typ')
{-3}	typ' = volatile(t')
{1}	TRUE

C Proof scripts

which is trivially true.

This completes the proof of `dt_cv_pointer_TCC12.1`.

`dt_cv_pointer_TCC12.2`:

{-1}	Cpp_Type?(typ')
{-2}	cv(pointer?)(typ')
{-3}	typ' = volatile(t')
{1}	pointer?(cv_base(t'))

Using lemma `dt_cv_pointer_TCC11`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `dt_cv_pointer_TCC12.2`.

`dt_cv_pointer_TCC12.3`:

{-1}	Cpp_Type?(typ')
{-2}	cv(pointer?)(typ')
{-3}	typ' = volatile(t')
{1}	Cpp_Type?(t')

Using lemma `dt_cv_pointer_TCC8`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `dt_cv_pointer_TCC12.3`.

Q.E.D.

C.44.51 Cpp_Types.dt_cv_pointer_TCC13

Terse proof for `dt_cv_pointer_TCC13`.

dt_cv_pointer_TCC13:

$ \begin{aligned} &\{1\} \quad \forall (\text{typ}: \text{Cpp_Subtype}(\text{cv}(\text{pointer?})), \\ &\quad v: \\ &\quad \quad [\text{typ1}: \\ &\quad \quad \quad \{z: \text{Cpp_Subtype}(\text{cv}(\text{pointer?})) \mid \\ &\quad \quad \quad \quad \text{restrict} \\ &\quad \quad \quad \quad \quad [[\text{Cpp_Type_}, \text{Cpp_Type_}], \\ &\quad \quad \quad \quad \quad \quad [\text{Cpp_Subtype}(\text{cv}(\text{pointer?})), \text{Cpp_Subtype}(\text{cv}(\text{pointer?}))], \\ &\quad \quad \quad \quad \quad \quad \text{boolean}] \\ &\quad \quad \quad \quad \quad (\ll)(z, \text{typ})\} \rightarrow \\ &\quad \quad \quad \quad \quad (\text{interpreted_data_type?}[(\text{range_pointer}(\text{cv_base}(\text{typ1}))))]), \\ &\quad t: \text{Cpp_Type_}: \\ &\quad \text{typ} = \text{volatile}(t) \supset \\ &\quad (\forall (x: \text{lift}[\text{Pointer_Base_Type}]): \\ &\quad \quad \text{range_pointer}(\text{cv_base}(t))(x) \equiv \text{range_pointer}(\text{cv_base}(\text{typ}))(x)) \\ &\quad \wedge \\ &\quad (\forall (x: \text{lift}[\text{Pointer_Base_Type}]): \\ &\quad \quad \text{range_pointer}(\text{cv_base}(t))(x) \equiv \text{range_pointer}(\text{cv_base}(\text{typ}))(x)) \\ &\quad \wedge \\ &\quad (\forall (x_1: [\text{list}[\text{Byte}], \text{Address}]): \\ &\quad \quad \text{every}[(\text{range_pointer}(\text{cv_base}(t)))] \\ &\quad \quad \quad (\text{range_pointer}(\text{cv_base}(\text{typ}))) \\ &\quad \quad \quad (\text{dt_volatile}[(\text{range_pointer}(\text{cv_base}(t)))](v(t)) \text{ 'from_byte} \\ &\quad \quad \quad \quad (x_1))) \\ &\quad \wedge \\ &\quad \text{interpreted_data_type?}[(\text{range_pointer}(\text{cv_base}(\text{typ})))]) \\ &\quad \quad (\text{dt_volatile}[(\text{range_pointer}(\text{cv_base}(t)))](v(t))) \end{aligned} $

Repeatedly Skolemizing and flattening,

Expanding the definition of cv_base,

Replacing using formula -3,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 2 subgoals:

dt_cv_pointer_TCC13.1:

$ \begin{aligned} &\{-1\} \quad \text{Cpp_Type?}(\text{volatile}(t')) \\ &\{-2\} \quad \text{cv}(\text{pointer?})(\text{volatile}(t')) \\ &\{-3\} \quad \text{typ}' = \text{volatile}(t') \end{aligned} $
$ \begin{aligned} &\{1\} \quad \text{interpreted_data_type?}[(\text{range_pointer}(\text{cv_base}(\text{typ}')))] \\ &\quad \quad (\text{dt_volatile}[(\text{range_pointer}(\text{cv_base}(t')))](v'(t'))) \end{aligned} $

Adding type constraints for dt_volatile(v!1(t!1)),

we get 3 subgoals:

dt_cv_pointer_TCC13.1.1:

$ \begin{aligned} &\{-1\} \quad \text{interpreted_data_type?}[(\text{range_pointer}(\text{cv_base}(t')))](\text{dt_volatile}(v'(t'))) \\ &\{-2\} \quad \text{Cpp_Type?}(\text{volatile}(t')) \\ &\{-3\} \quad \text{cv}(\text{pointer?})(\text{volatile}(t')) \\ &\{-4\} \quad \text{typ}' = \text{volatile}(t') \end{aligned} $
$ \begin{aligned} &\{1\} \quad \text{interpreted_data_type?}[(\text{range_pointer}(\text{cv_base}(\text{typ}')))] \\ &\quad \quad (\text{dt_volatile}[(\text{range_pointer}(\text{cv_base}(t')))](v'(t'))) \end{aligned} $

Expanding the definition of `interpreted_data_type?`,
 Applying propositional simplification,
 we get 2 subgoals:

`dt_cv_pointer_TCC13.1.1.1:`

<div style="display: flex; flex-direction: column; gap: 5px;"> <div style="display: flex; align-items: flex-start;"> {-1} $\text{uninterpreted_data_type?}(\text{dt_volatile}(v'(t')) \text{'uidt})$ </div> <div style="display: flex; align-items: flex-start;"> {-2} $\forall (d: ((\text{range_pointer}(\text{cv_base}(t')))), a: \text{Address}):$ $\text{valid?}(\text{uidt}(\text{dt_volatile}(v'(t'))))(\text{to_byte}(\text{dt_volatile}(v'(t'))))(d, a), a)$ </div> <div style="display: flex; align-items: flex-start;"> {-3} $\forall (l: \text{list}[\text{Byte}], a: \text{Address}):$ $\text{valid?}(\text{uidt}(\text{dt_volatile}(v'(t'))))(l, a) \equiv$ $\text{up?}(\text{from_byte}(\text{dt_volatile}(v'(t'))))(l, a)$ </div> <div style="display: flex; align-items: flex-start;"> {-4} $\forall (d: ((\text{range_pointer}(\text{cv_base}(t')))), a: \text{Address}):$ $\text{down}(\text{from_byte}(\text{dt_volatile}(v'(t'))))$ $(\text{to_byte}(\text{dt_volatile}(v'(t'))))(d, a), a)$ $= d$ </div> <div style="display: flex; align-items: flex-start;"> {-5} $\text{Cpp_Type?}(\text{volatile}(t'))$ </div> <div style="display: flex; align-items: flex-start;"> {-6} $\text{cv}(\text{pointer?})(\text{volatile}(t'))$ </div> <div style="display: flex; align-items: flex-start;"> {-7} $\text{typ}' = \text{volatile}(t')$ </div> </div>	<hr style="border: 0.5px solid black;"/> <div style="display: flex; align-items: flex-start;"> {1} $\forall (d: ((\text{range_pointer}(\text{cv_base}(\text{typ}')))), a: \text{Address}):$ $\text{valid?}(\text{uidt}(\text{dt_volatile}[\text{range_pointer}(\text{cv_base}(t'))](v'(t'))))$ $(\text{to_byte}(\text{dt_volatile}[\text{range_pointer}(\text{cv_base}(t'))](v'(t'))))(d, a),$ $a)$ </div>
--	---

Repeatedly Skolemizing and flattening,
 Instantiating quantified variables,
 Replacing using formula -7,
 Expanding the definition of `cv_base`,
 which is trivially true.

This completes the proof of `dt_cv_pointer_TCC13.1.1.1`.

`dt_cv_pointer_TCC13.1.1.2:`

<div style="display: flex; flex-direction: column; gap: 5px;"> <div style="display: flex; align-items: flex-start;"> {-1} $\text{uninterpreted_data_type?}(\text{dt_volatile}(v'(t')) \text{'uidt})$ </div> <div style="display: flex; align-items: flex-start;"> {-2} $\forall (d: ((\text{range_pointer}(\text{cv_base}(t')))), a: \text{Address}):$ $\text{valid?}(\text{uidt}(\text{dt_volatile}(v'(t'))))(\text{to_byte}(\text{dt_volatile}(v'(t'))))(d, a), a)$ </div> <div style="display: flex; align-items: flex-start;"> {-3} $\forall (l: \text{list}[\text{Byte}], a: \text{Address}):$ $\text{valid?}(\text{uidt}(\text{dt_volatile}(v'(t'))))(l, a) \equiv$ $\text{up?}(\text{from_byte}(\text{dt_volatile}(v'(t'))))(l, a)$ </div> <div style="display: flex; align-items: flex-start;"> {-4} $\forall (d: ((\text{range_pointer}(\text{cv_base}(t')))), a: \text{Address}):$ $\text{down}(\text{from_byte}(\text{dt_volatile}(v'(t'))))$ $(\text{to_byte}(\text{dt_volatile}(v'(t'))))(d, a), a)$ $= d$ </div> <div style="display: flex; align-items: flex-start;"> {-5} $\text{Cpp_Type?}(\text{volatile}(t'))$ </div> <div style="display: flex; align-items: flex-start;"> {-6} $\text{cv}(\text{pointer?})(\text{volatile}(t'))$ </div> <div style="display: flex; align-items: flex-start;"> {-7} $\text{typ}' = \text{volatile}(t')$ </div> </div>	<hr style="border: 0.5px solid black;"/> <div style="display: flex; align-items: flex-start;"> {1} $\forall (d: ((\text{range_pointer}(\text{cv_base}(\text{typ}')))), a: \text{Address}):$ $\text{down}(\text{from_byte}(\text{dt_volatile}[\text{range_pointer}(\text{cv_base}(t'))](v'(t'))))$ $(\text{to_byte}(\text{dt_volatile}[\text{range_pointer}(\text{cv_base}(t'))](v'(t'))))$ $(d, a),$ $a))$ $= d$ </div>
--	--

Repeatedly Skolemizing and flattening,
 Instantiating quantified variables,
 Replacing using formula -7,
 Expanding the definition of `cv_base`,

which is trivially true.

This completes the proof of `dt_cv_pointer_TCC13.1.1.2`.

`dt_cv_pointer_TCC13.1.2`:

{-1}	<code>Cpp_Type?(volatile(t'))</code>	
{-2}	<code>cv(pointer?)(volatile(t'))</code>	
{-3}	<code>typ' = volatile(t')</code>	
{1}		$\exists (x_1: (\text{interpreted_data_type?}[\text{((range_pointer(cv_base(t')))]))): \text{TRUE}$
{2}	$\text{interpreted_data_type?}[\text{((range_pointer(cv_base(typ')))])$ $(\text{dt_volatile}[\text{((range_pointer(cv_base(t')))])(v'(t'))})$	

Using lemma `dt_cv_pointer_TCC12`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `dt_cv_pointer_TCC13.1.2`.

`dt_cv_pointer_TCC13.1.3`:

{-1}	<code>Cpp_Type?(volatile(t'))</code>	
{-2}	<code>cv(pointer?)(volatile(t'))</code>	
{-3}	<code>typ' = volatile(t')</code>	
{1}		<code>pointer?(cv_base(t'))</code>
{2}	$\text{interpreted_data_type?}[\text{((range_pointer(cv_base(typ')))])$ $(\text{dt_volatile}[\text{((range_pointer(cv_base(t')))])(v'(t'))})$	

Using lemma `dt_cv_pointer_TCC11`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `dt_cv_pointer_TCC13.1.3`.

`dt_cv_pointer_TCC13.2`:

{-1}	<code>Cpp_Type?(volatile(t'))</code>	
{-2}	<code>cv(pointer?)(volatile(t'))</code>	
{-3}	<code>typ' = volatile(t')</code>	
{1}		$\forall (x_1: [\text{list}[\text{Byte}], \text{Address}]):$ $\text{every}[\text{((range_pointer(cv_base(t')))])$ $(\text{range_pointer}(cv_base(volatile(t'))))$ $(\text{dt_volatile}[\text{((range_pointer(cv_base(t')))])(v'(t'))} \text{ 'from_byte}(x_1))$

Repeatedly Skolemizing and flattening,

Expanding the definition of `every`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Expanding the definition of `cv_base`,

which is trivially true.

This completes the proof of `dt_cv_pointer_TCC13.2`.

Q.E.D.

C.44.52 Cpp_Types.dt_cv_pointer_TCC14

Terse proof for `dt_cv_pointer_TCC14`.

`dt_cv_pointer_TCC14`:

{1}	$\forall (\text{typ}: \text{Cpp_Subtype}(cv(\text{pointer?})))$ $\neg \text{volatile?}(\text{typ}) \wedge \neg \text{const?}(\text{typ}) \supset \text{pointer?}(\text{typ})$
-----	---

Repeatedly Skolemizing and flattening,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `dt_cv_pointer_TCC14`.
 Q.E.D.

C.44.53 Cpp_Types.dt_cv_pointer_TCC15

Terse proof for `dt_cv_pointer_TCC15`.

`dt_cv_pointer_TCC15`:

<pre> {1} ∀ (typ: Cpp_Subtype(cv(pointer?))): ¬ volatile?(typ) ∧ ¬ const?(typ) ⊃ (∀ (x: lift[Pointer_Base_Type]): range_pointer(typ)(x) ≡ range_pointer(cv_base(typ))(x)) ∧ (∀ (x: lift[Pointer_Base_Type]): range_pointer(typ)(x) ≡ range_pointer(cv_base(typ))(x)) ∧ (∀ (x₁: [list[Byte], Address]): every[((range_pointer(typ)))] ((range_pointer(cv_base(typ)))(dt_pointer(typ) 'from_byte(x₁))) ∧ interpreted_data_type?[(range_pointer(cv_base(typ)))(dt_pointer(typ))]) </pre>
--

Repeatedly Skolemizing and flattening,
 Using lemma `cv_base_result2`,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 we get 3 subgoals:

`dt_cv_pointer_TCC15.1`:

<pre> {-1} cv_base(typ') = typ' {-2} Cpp_Type?(typ') {-3} cv(pointer?)(typ') </pre>
<pre> {1} const?(typ') {2} volatile?(typ') {3} interpreted_data_type?[((range_pointer(cv_base(typ')))](dt_pointer(typ')) </pre>

Replacing using formula -1,
 Adding type constraints for `dt_pointer(cv_base(typ!1))`,
 Rewriting using `pod_is_interpreted_data`, matching in *,
 This completes the proof of `dt_cv_pointer_TCC15.1`.

`dt_cv_pointer_TCC15.2`:

<pre> {-1} cv_base(typ') = typ' {-2} Cpp_Type?(typ') {-3} cv(pointer?)(typ') </pre>
<pre> {1} const?(typ') {2} volatile?(typ') {3} ∀ (x₁: [list[Byte], Address]): every[((range_pointer(typ')))] ((range_pointer(cv_base(typ')))(dt_pointer(typ') 'from_byte(x₁))) </pre>

Expanding the definition of every,
 Repeatedly Skolemizing and flattening,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 This completes the proof of `dt_cv_pointer_TCC15.2`.

dt_cv_pointer_TCC15.3:

{-1}	cv_base(typ') = typ'
{-2}	Cpp_Type?(typ')
{-3}	cv(pointer?)(typ')
{1}	const?(typ')
{2}	volatile?(typ')
{3}	$\forall (x: \text{lift}[\text{Pointer_Base_Type}]):$ $\text{range_pointer}(typ')(x) \equiv \text{range_pointer}(cv_base(typ'))(x)$

Repeatedly Skolemizing and flattening,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 This completes the proof of dt_cv_pointer_TCC15.3.
 Q.E.D.

C.44.54 Cpp_Types.dt_cv_bitfield_TCC1

Terse proof for dt_cv_bitfield_TCC1.

dt_cv_bitfield_TCC1:

{1}	$\text{well_founded?}(\lambda (x: \text{Cpp_Subtype}(cv(\text{bitfield?})), y: \text{Cpp_Subtype}(cv(\text{bitfield?}))):$ restrict $[[\text{Cpp_Type_}, \text{Cpp_Type_}],$ $[\text{Cpp_Subtype}(cv(\text{bitfield?})), \text{Cpp_Subtype}(cv(\text{bitfield?}))], \text{boolean}]$ $(\ll)(x, y))$
-----	---

Using lemma well_founded_restrict[Cpp_Type_, Cpp_Subtype(cv(bitfield?))],
 Expanding the definition of restrict,
 which is trivially true.
 This completes the proof of dt_cv_bitfield_TCC1.
 Q.E.D.

C.44.55 Cpp_Types.dt_cv_bitfield_TCC2

Terse proof for dt_cv_bitfield_TCC2.

dt_cv_bitfield_TCC2:

{1}	$\forall (typ: \text{Cpp_Subtype}(cv(\text{bitfield?})), t: \text{Cpp_Type_}):$ $typ = \text{const}(t) \supset \text{Cpp_Type?}(t) \wedge cv(\text{bitfield?})(t)$
-----	---

Repeatedly Skolemizing and flattening,
 Replacing using formula -3,
 Applying propositional simplification,
 we get 2 subgoals:

dt_cv_bitfield_TCC2.1:

{-1}	Cpp_Type?(const(t'))
{-2}	cv(bitfield?)(const(t'))
{-3}	typ' = const(t')
{1}	Cpp_Type?(t')

Expanding the definition of Cpp_Type?,
 Repeatedly Skolemizing and flattening,
 Instantiating the top quantifier in -2 with the terms: (t!2),

C Proof scripts

Expanding the definition of subterm,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `dt_cv_bitfield_TCC2.1`.
`dt_cv_bitfield_TCC2.2`:

{-1}	Cpp_Type?(const(t'))
{-2}	cv(bitfield?)(const(t'))
{-3}	typ' = const(t')
{1}	cv(bitfield?)(t')

Keeping (-2 1) and hiding *,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `dt_cv_bitfield_TCC2.2`.
Q.E.D.

C.44.56 Cpp_Types.dt_cv_bitfield_TCC3

Terse proof for `dt_cv_bitfield_TCC3`.

`dt_cv_bitfield_TCC3`:

{1}	\forall (typ: Cpp_Subtype(cv(bitfield?)), t : Cpp_Type_-): typ = const(t) \supset restrict [[Cpp_Type_-, Cpp_Type_-], [Cpp_Subtype(cv(bitfield?)), Cpp_Subtype(cv(bitfield?))], boolean] (\ll)(t , typ)
-----	---

Expanding the definition of restrict,
Repeatedly Skolemizing and flattening,
Replacing using formula -3,
Expanding the definition of \blacksquare ,
which is trivially true.
This completes the proof of `dt_cv_bitfield_TCC3`.
Q.E.D.

C.44.57 Cpp_Types.dt_cv_bitfield_TCC4

Terse proof for `dt_cv_bitfield_TCC4`.

`dt_cv_bitfield_TCC4`:

{1}	\forall (typ: Cpp_Subtype(cv(bitfield?)), t : Cpp_Type_-): typ = const(t) \supset Cpp_Type?(t)
-----	--

Repeatedly Skolemizing and flattening,
Using lemma `dt_cv_bitfield_TCC2`,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `dt_cv_bitfield_TCC4`.
Q.E.D.

C.44.58 Cpp_Types.dt_cv_bitfield_TCC5

Terse proof for `dt_cv_bitfield_TCC5`.

dt_cv_bitfield_TCC5:

{1}	\forall (typ: Cpp_Subtype(cv(bitfield?)), t: Cpp_Type_): typ = const(t) \supset bitfield?(cv_base(t))
-----	--

Repeatedly Skolemizing and flattening,
Rewriting using cv_base_result, matching in *,
we get 2 subgoals:

dt_cv_bitfield_TCC5.1:

{-1}	Cpp_Type?(typ')
{-2}	cv(bitfield?)(typ')
{-3}	typ' = const(t')
{1}	cv(bitfield?)(t')
{2}	bitfield?(cv_base(t'))

Using lemma dt_cv_bitfield_TCC2,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of dt_cv_bitfield_TCC5.1.

dt_cv_bitfield_TCC5.2:

{-1}	Cpp_Type?(typ')
{-2}	cv(bitfield?)(typ')
{-3}	typ' = const(t')
{1}	(bitfield? \subseteq interpreted?)
{2}	bitfield?(cv_base(t'))

Keeping (1) and hiding *,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of dt_cv_bitfield_TCC5.2.
Q.E.D.

C.44.59 Cpp_Types.dt_cv_bitfield_TCC6

Terse proof for dt_cv_bitfield_TCC6.

dt_cv_bitfield_TCC6:

{1}	\forall (typ: Cpp_Subtype(cv(bitfield?)), t: Cpp_Type_): typ = const(t) \supset (\exists (x ₁ : (interpreted_data_type?[(range_bitfield(cv_base(t))]))): TRUE)
-----	--

Repeatedly Skolemizing and flattening,
Using lemma dt_cv_bitfield_TCC2,
Using lemma dt_bitfield_exists,
we get 2 subgoals:

dt_cv_bitfield_TCC6.1:

{-1}	typ' = const(t') \supset Cpp_Type?(t') \wedge cv(bitfield?)(t')
{-2}	Cpp_Type?(typ')
{-3}	cv(bitfield?)(typ')
{-4}	typ' = const(t')
{1}	bitfield?(cv_base(t'))
{2}	\exists (x ₁ : (interpreted_data_type?[(range_bitfield(cv_base(t')))])): TRUE

Using lemma dt_cv_bitfield_TCC5,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `dt_cv_bitfield_TCC6.1`.

`dt_cv_bitfield_TCC6.2`:

<pre>{-1} typ' = const(t') ⊃ Cpp_Type?(t') ∧ cv(bitfield?)(t') {-2} Cpp_Type?(typ') {-3} cv(bitfield?)(typ') {-4} typ' = const(t')</pre>	<hr style="border: 0.5px solid black;"/> <pre>{1} Cpp_Type?(t') {2} ∃ (x₁: (interpreted_data_type?[(range_bitfield(cv_base(t')))])): TRUE</pre>
--	--

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `dt_cv_bitfield_TCC6.2`.

Q.E.D.

C.44.60 Cpp_Types.dt_cv_bitfield_TCC7

Terse proof for `dt_cv_bitfield_TCC7`.

`dt_cv_bitfield_TCC7`:

<pre>{1} ∀ (typ: Cpp_Subtype(cv(bitfield?)), v: [typ1: {z: Cpp_Subtype(cv(bitfield?)) restrict [[Cpp_Type_, Cpp_Type_], [Cpp_Subtype(cv(bitfield?)), Cpp_Subtype(cv(bitfield?))], boolean] (≪≪)(z, typ)} → (interpreted_data_type?[(range_bitfield(cv_base(typ1)))]), t: Cpp_Type_): typ = const(t) ⊃ (∀ (x: int): range_bitfield(cv_base(t))(x) ≡ range_bitfield(cv_base(typ))(x)) ∧ (∀ (x: int): range_bitfield(cv_base(t))(x) ≡ range_bitfield(cv_base(typ))(x)) ∧ (∀ (x₁: [list[Byte], Address]): every[(range_bitfield(cv_base(t)))] ((range_bitfield(cv_base(typ)))) (dt_const[(range_bitfield(cv_base(t)))](v(t)) 'from_byte(x₁))) ∧ interpreted_data_type?[(range_bitfield(cv_base(typ)))] (dt_const[(range_bitfield(cv_base(t)))](v(t)))</pre>	<hr style="border: 0.5px solid black;"/>
--	--

Repeatedly Skolemizing and flattening,
Replacing using formula -3,
Expanding the definition of `cv_base`,
Applying propositional simplification,
we get 2 subgoals:

dt_cv_bitfield_TCC7.1:

{-1}	Cpp_Type?(const(t'))
{-2}	cv(bitfield?)(const(t'))
{-3}	typ' = const(t')
{1}	$\forall (x_1: [\text{list}[\text{Byte}], \text{Address}]):$ every[$((\text{range_bitfield}(\text{cv_base}(\mathit{t}'))))$] $((\text{range_bitfield}(\text{cv_base}(\text{const}(\mathit{t}'))))$)] $(\text{dt_const}[\mathit{((\text{range_bitfield}(\text{cv_base}(\mathit{t}'))))}] (\mathit{v}'(\mathit{t}')) \text{'from_byte}(x_1))$

Expanding the definition of every,

Repeatedly Skolemizing and flattening,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Expanding the definition of cv_base,

which is trivially true.

This completes the proof of dt_cv_bitfield_TCC7.1.

dt_cv_bitfield_TCC7.2:

{-1}	Cpp_Type?(const(t'))
{-2}	cv(bitfield?)(const(t'))
{-3}	typ' = const(t')
{1}	interpreted_data_type? $\mathit{((\text{range_bitfield}(\text{cv_base}(\mathit{t}'))))}$] $(\text{dt_const}[\mathit{((\text{range_bitfield}(\text{cv_base}(\mathit{t}'))))}] (\mathit{v}'(\mathit{t}'))$

Adding type constraints for dt_const($v!1(\mathit{t}!1)$),

we get 3 subgoals:

dt_cv_bitfield_TCC7.2.1:

{-1}	interpreted_data_type? $\mathit{((\text{range_bitfield}(\text{cv_base}(\mathit{t}'))))}$](dt_const($v'(\mathit{t}')$))
{-2}	Cpp_Type?(const(t'))
{-3}	cv(bitfield?)(const(t'))
{-4}	typ' = const(t')
{1}	interpreted_data_type? $\mathit{((\text{range_bitfield}(\text{cv_base}(\mathit{t}'))))}$] $(\text{dt_const}[\mathit{((\text{range_bitfield}(\text{cv_base}(\mathit{t}'))))}] (\mathit{v}'(\mathit{t}'))$

Expanding the definition of interpreted_data_type?,

Applying propositional simplification,

we get 2 subgoals:

dt_cv_bitfield_TCC7.2.1.1:

{-1}	uninterpreted_data_type?(dt_const($v'(\mathit{t}')$)'uidt)
{-2}	$\forall (d: ((\text{range_bitfield}(\text{cv_base}(\mathit{t}')))), a: \text{Address}):$ valid?(uidt(dt_const($v'(\mathit{t}')$)))(to_byte(dt_const($v'(\mathit{t}')$)))(d, a, a)
{-3}	$\forall (l: \text{list}[\text{Byte}], a: \text{Address}):$ valid?(uidt(dt_const($v'(\mathit{t}')$)))(l, a) \equiv up?(from_byte(dt_const($v'(\mathit{t}')$)))(l, a)
{-4}	$\forall (d: ((\text{range_bitfield}(\text{cv_base}(\mathit{t}')))), a: \text{Address}):$ down(from_byte(dt_const($v'(\mathit{t}')$)))(to_byte(dt_const($v'(\mathit{t}')$)))(d, a, a) = d
{-5}	Cpp_Type?(const(t'))
{-6}	cv(bitfield?)(const(t'))
{-7}	typ' = const(t')
{1}	$\forall (d: ((\text{range_bitfield}(\text{cv_base}(\mathit{t}')))), a: \text{Address}):$ valid?(uidt(dt_const $\mathit{((\text{range_bitfield}(\text{cv_base}(\mathit{t}')))] (\mathit{v}'(\mathit{t}'))$)] $(\text{to_byte}(\text{dt_const}[\mathit{((\text{range_bitfield}(\text{cv_base}(\mathit{t}')))] (\mathit{v}'(\mathit{t}')))] (\mathit{v}'(\mathit{t}')))(\mathit{d}, \mathit{a}),$ $\mathit{a})$

C Proof scripts

Repeatedly Skolemizing and flattening,
 Instantiating quantified variables,
 Replacing using formula -7,
 Expanding the definition of cv_base,
 which is trivially true.

This completes the proof of dt_cv_bitfield_TCC7.2.1.1.

dt_cv_bitfield_TCC7.2.1.2:

<pre> {-1} uninterpreted_data_type?(dt_const(v'(t'))'uidt) {-2} ∀ (d: ((range_bitfield(cv_base(t')))), a: Address): valid?(uidt(dt_const(v'(t')))(to_byte(dt_const(v'(t')))(d, a), a) {-3} ∀ (l: list[Byte], a: Address): valid?(uidt(dt_const(v'(t')))(l, a) ≡ up?(from_byte(dt_const(v'(t')))(l, a)) {-4} ∀ (d: ((range_bitfield(cv_base(t')))), a: Address): down(from_byte(dt_const(v'(t')))(to_byte(dt_const(v'(t')))(d, a), a) = d {-5} Cpp_Type?(const(t')) {-6} cv(bitfield?)(const(t')) {-7} typ' = const(t') </pre>	<pre> {1} ∀ (d: ((range_bitfield(cv_base(typ')))), a: Address): down(from_byte(dt_const[((range_bitfield(cv_base(t')))](v'(t')) (to_byte(dt_const[((range_bitfield(cv_base(t')))](v'(t')) (d, a), a)) = d </pre>
---	---

Repeatedly Skolemizing and flattening,
 Instantiating quantified variables,
 Replacing using formula -7,
 Expanding the definition of cv_base,
 which is trivially true.

This completes the proof of dt_cv_bitfield_TCC7.2.1.2.

dt_cv_bitfield_TCC7.2.2:

<pre> {-1} Cpp_Type?(const(t')) {-2} cv(bitfield?)(const(t')) {-3} typ' = const(t') </pre>	<pre> {1} ∃ (x1: (interpreted_data_type?[((range_bitfield(cv_base(t')))])): TRUE {2} interpreted_data_type?[((range_bitfield(cv_base(typ')))] (dt_const[((range_bitfield(cv_base(t')))](v'(t')) </pre>
---	--

Using lemma dt_cv_bitfield_TCC6,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of dt_cv_bitfield_TCC7.2.2.

dt_cv_bitfield_TCC7.2.3:

<pre> {-1} Cpp_Type?(const(t')) {-2} cv(bitfield?)(const(t')) {-3} typ' = const(t') </pre>	<pre> {1} bitfield?(cv_base(t')) {2} interpreted_data_type?[((range_bitfield(cv_base(typ')))] (dt_const[((range_bitfield(cv_base(t')))](v'(t')) </pre>
---	--

Using lemma dt_cv_bitfield_TCC5,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of dt_cv_bitfield_TCC7.2.3.
Q.E.D.

C.44.61 Cpp_Types.dt_cv_bitfield_TCC8

Terse proof for dt_cv_bitfield_TCC8.

dt_cv_bitfield_TCC8:

$\{1\} \quad \forall (\text{typ}: \text{Cpp_Subtype}(\text{cv}(\text{bitfield?})), t: \text{Cpp_Type_}):$ $\text{typ} = \text{volatile}(t) \supset \text{Cpp_Type?}(t) \wedge \text{cv}(\text{bitfield?})(t)$

Repeatedly Skolemizing and flattening,
Applying propositional simplification,
we get 2 subgoals:

dt_cv_bitfield_TCC8.1:

$\{-1\} \quad \text{Cpp_Type?}(\text{typ}')$ $\{-2\} \quad \text{cv}(\text{bitfield?})(\text{typ}')$ $\{-3\} \quad \text{typ}' = \text{volatile}(t')$
$\{1\} \quad \text{Cpp_Type?}(t')$

Expanding the definition of Cpp_Type?,
Repeatedly Skolemizing and flattening,
Instantiating the top quantifier in -2 with the terms: (t!2),
Replacing using formula -4,
Expanding the definition of subterm,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of dt_cv_bitfield_TCC8.1.

dt_cv_bitfield_TCC8.2:

$\{-1\} \quad \text{Cpp_Type?}(\text{typ}')$ $\{-2\} \quad \text{cv}(\text{bitfield?})(\text{typ}')$ $\{-3\} \quad \text{typ}' = \text{volatile}(t')$
$\{1\} \quad \text{cv}(\text{bitfield?})(t')$

Hiding formulas: -1,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of dt_cv_bitfield_TCC8.2.
Q.E.D.

C.44.62 Cpp_Types.dt_cv_bitfield_TCC9

Terse proof for dt_cv_bitfield_TCC9.

dt_cv_bitfield_TCC9:

$\{1\} \quad \forall (\text{typ}: \text{Cpp_Subtype}(\text{cv}(\text{bitfield?})), t: \text{Cpp_Type_}):$ $\text{typ} = \text{volatile}(t) \supset$ restrict $\quad [[\text{Cpp_Type_}, \text{Cpp_Type_}],$ $\quad [\text{Cpp_Subtype}(\text{cv}(\text{bitfield?})), \text{Cpp_Subtype}(\text{cv}(\text{bitfield?}))], \text{boolean}]$ $(\ll)(t, \text{typ})$
--

Repeatedly Skolemizing and flattening,
Replacing using formula -3,

Expanding the definition of restrict,
 Expanding the definition of ■,
 which is trivially true.
 This completes the proof of dt_cv_bitfield_TCC9.
 Q.E.D.

C.44.63 Cpp_Types.dt_cv_bitfield_TCC10

Terse proof for dt_cv_bitfield_TCC10.
 dt_cv_bitfield_TCC10:

{1}	$\forall (\text{typ}: \text{Cpp_Subtype}(\text{cv}(\text{bitfield?})), t: \text{Cpp_Type_}): \text{typ} = \text{volatile}(t) \supset \text{Cpp_Type?}(t)$
-----	---

Repeatedly Skolemizing and flattening,
 Using lemma dt_cv_bitfield_TCC8,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 This completes the proof of dt_cv_bitfield_TCC10.
 Q.E.D.

C.44.64 Cpp_Types.dt_cv_bitfield_TCC11

Terse proof for dt_cv_bitfield_TCC11.
 dt_cv_bitfield_TCC11:

{1}	$\forall (\text{typ}: \text{Cpp_Subtype}(\text{cv}(\text{bitfield?})), t: \text{Cpp_Type_}):$ $\text{typ} = \text{volatile}(t) \supset \text{bitfield?}(\text{cv_base}(t))$
-----	--

Repeatedly Skolemizing and flattening,
 Using lemma dt_cv_bitfield_TCC8,
 Using lemma cv_base_result,
 we get 2 subgoals:
 dt_cv_bitfield_TCC11.1:

{-1}	$\text{typ}' = \text{volatile}(t') \supset \text{Cpp_Type?}(t') \wedge \text{cv}(\text{bitfield?})(t')$
{-2}	$\text{Cpp_Type?}(\text{typ}')$
{-3}	$\text{cv}(\text{bitfield?})(\text{typ}')$
{-4}	$\text{typ}' = \text{volatile}(t')$
{1}	$\text{cv}(\text{bitfield?})(t')$
{2}	$\text{bitfield?}(\text{cv_base}(t'))$

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 This completes the proof of dt_cv_bitfield_TCC11.1.
 dt_cv_bitfield_TCC11.2:

{-1}	$\text{typ}' = \text{volatile}(t') \supset \text{Cpp_Type?}(t') \wedge \text{cv}(\text{bitfield?})(t')$
{-2}	$\text{Cpp_Type?}(\text{typ}')$
{-3}	$\text{cv}(\text{bitfield?})(\text{typ}')$
{-4}	$\text{typ}' = \text{volatile}(t')$
{1}	$(\text{bitfield?} \subseteq \text{interpreted?})$
{2}	$\text{bitfield?}(\text{cv_base}(t'))$

Keeping (1) and hiding *,
 Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of dt_cv_bitfield_TCC11.2.

Q.E.D.

C.44.65 Cpp_Types.dt_cv_bitfield_TCC12

Terse proof for dt_cv_bitfield_TCC12.

dt_cv_bitfield_TCC12:

{1}	$\forall (\text{typ}: \text{Cpp_Subtype}(\text{cv}(\text{bitfield?})), t: \text{Cpp_Type_}):$ $\text{typ} = \text{volatile}(t) \supset$ $(\exists (x_1: (\text{interpreted_data_type?}[\text{((range_bitfield}(\text{cv_base}(t))))]))): \text{TRUE}$
-----	--

Repeatedly Skolemizing and flattening,

Using lemma dt_bitfield_exists,

we get 2 subgoals:

dt_cv_bitfield_TCC12.1:

{-1}	Cpp_Type?(typ')
{-2}	cv(bitfield?)(typ')
{-3}	typ' = volatile(t')
{1}	bitfield?(cv_base(t'))
{2}	$\exists (x_1: (\text{interpreted_data_type?}[\text{((range_bitfield}(\text{cv_base}(t')))]))): \text{TRUE}$

Using lemma dt_cv_bitfield_TCC11,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of dt_cv_bitfield_TCC12.1.

dt_cv_bitfield_TCC12.2:

{-1}	Cpp_Type?(typ')
{-2}	cv(bitfield?)(typ')
{-3}	typ' = volatile(t')
{1}	Cpp_Type?(t')
{2}	$\exists (x_1: (\text{interpreted_data_type?}[\text{((range_bitfield}(\text{cv_base}(t')))]))): \text{TRUE}$

Using lemma dt_cv_bitfield_TCC8,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of dt_cv_bitfield_TCC12.2.

Q.E.D.

C.44.66 Cpp_Types.dt_cv_bitfield_TCC13

Terse proof for dt_cv_bitfield_TCC13.

dt_cv_bitfield_TCC13:

```

{1}  ∀ (typ: Cpp_Subtype(cv(bitfield?)),
      v:
        [typ1:
          {z: Cpp_Subtype(cv(bitfield?)) |
            restrict
              [[Cpp_Type_, Cpp_Type_],
               [Cpp_Subtype(cv(bitfield?)), Cpp_Subtype(cv(bitfield?))],
               boolean]
              (≪≪)(z, typ)} →
            (interpreted_data_type?[(range_bitfield(cv_base(typ1)))]),
          t: Cpp_Type_):
  typ = volatile(t) ⊃
  (∀ (x: int):
    range_bitfield(cv_base(t))(x) ≡ range_bitfield(cv_base(typ))(x))
  ∧
  (∀ (x: int):
    range_bitfield(cv_base(t))(x) ≡ range_bitfield(cv_base(typ))(x))
  ∧
  (∀ (x1: [list[Byte], Address]):
    every[(range_bitfield(cv_base(t)))]
      ((range_bitfield(cv_base(typ))))
      (dt_volatile[(range_bitfield(cv_base(t)))](v(t))'from_byte
        (x1)))
  ∧
  interpreted_data_type?[(range_bitfield(cv_base(typ)))]
    (dt_volatile[(range_bitfield(cv_base(t)))](v(t)))

```

Repeatedly Skolemizing and flattening,
 Replacing using formula -3,
 Expanding the definition of cv_base,
 Applying propositional simplification,
 we get 2 subgoals:

dt_cv_bitfield_TCC13.1:

```

{-1} Cpp_Type?(volatile(t'))
{-2} cv(bitfield?)(volatile(t'))
{-3} typ' = volatile(t')
{1}  ∀ (x1: [list[Byte], Address]):
  every[(range_bitfield(cv_base(t')))]
    ((range_bitfield(cv_base(volatile(t')))))
    (dt_volatile[(range_bitfield(cv_base(t')))](v'(t'))'from_byte(x1))

```

Expanding the definition of every,
 Repeatedly Skolemizing and flattening,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 Expanding the definition of cv_base,
 which is trivially true.

This completes the proof of dt_cv_bitfield_TCC13.1.

dt_cv_bitfield_TCC13.2:

{-1}	Cpp_Type?(volatile(t'))
{-2}	cv(bitfield?)(volatile(t'))
{-3}	typ' = volatile(t')
{1}	interpreted_data_type?[((range_bitfield(cv_base(typ'))))] (dt_volatile[((range_bitfield(cv_base(t')))](v' (t')))

Adding type constraints for dt_volatile($v'!$ ($t!$)),

we get 3 subgoals:

dt_cv_bitfield_TCC13.2.1:

{-1}	interpreted_data_type?[((range_bitfield(cv_base(t')))](dt_volatile(v' (t')))
{-2}	Cpp_Type?(volatile(t'))
{-3}	cv(bitfield?)(volatile(t'))
{-4}	typ' = volatile(t')
{1}	interpreted_data_type?[((range_bitfield(cv_base(typ'))))] (dt_volatile[((range_bitfield(cv_base(t')))](v' (t')))

Expanding the definition of interpreted_data_type?,

Applying propositional simplification,

we get 2 subgoals:

dt_cv_bitfield_TCC13.2.1.1:

{-1}	uninterpreted_data_type?(dt_volatile(v' (t'))'uidt)
{-2}	$\forall (d: ((range_bitfield(cv_base(t')))), a: Address):$ valid?(uidt(dt_volatile(v' (t')))(to_byte(dt_volatile(v' (t')))(d, a), a)
{-3}	$\forall (l: list[Byte], a: Address):$ valid?(uidt(dt_volatile(v' (t')))(l, a) \equiv up?(from_byte(dt_volatile(v' (t')))(l, a))
{-4}	$\forall (d: ((range_bitfield(cv_base(t')))), a: Address):$ down(from_byte(dt_volatile(v' (t')))((to_byte(dt_volatile(v' (t')))(d, a), a)) = d
{-5}	Cpp_Type?(volatile(t'))
{-6}	cv(bitfield?)(volatile(t'))
{-7}	typ' = volatile(t')
{1}	$\forall (d: ((range_bitfield(cv_base(typ')))), a: Address):$ valid?(uidt(dt_volatile[((range_bitfield(cv_base(t')))](v' (t'))) (to_byte(dt_volatile[((range_bitfield(cv_base(t')))](v' (t')))(d, a), a))

Repeatedly Skolemizing and flattening,

Instantiating quantified variables,

Replacing using formula -7,

Expanding the definition of cv_base,

which is trivially true.

This completes the proof of dt_cv_bitfield_TCC13.2.1.1.

dt_cv_bitfield_TCC13.2.1.2:

<div style="display: flex; flex-direction: column; gap: 5px;"> <div style="display: flex; align-items: flex-start;"> {-1} uninterpreted_data_type?(dt_volatile(v'(t'))'uidt) </div> <div style="display: flex; align-items: flex-start;"> {-2} $\forall (d: ((\text{range_bitfield}(\text{cv_base}(t')))), a: \text{Address}):$ $\text{valid?}(\text{uidt}(\text{dt_volatile}(v'(t'))))(\text{to_byte}(\text{dt_volatile}(v'(t'))))(d, a), a)$ </div> <div style="display: flex; align-items: flex-start;"> {-3} $\forall (l: \text{list}[\text{Byte}], a: \text{Address}):$ $\text{valid?}(\text{uidt}(\text{dt_volatile}(v'(t'))))(l, a) \equiv$ $\text{up?}(\text{from_byte}(\text{dt_volatile}(v'(t'))))(l, a)$ </div> <div style="display: flex; align-items: flex-start;"> {-4} $\forall (d: ((\text{range_bitfield}(\text{cv_base}(t')))), a: \text{Address}):$ $\text{down}(\text{from_byte}(\text{dt_volatile}(v'(t'))))$ $(\text{to_byte}(\text{dt_volatile}(v'(t'))))(d, a), a)$ $= d$ </div> <div style="display: flex; align-items: flex-start;"> {-5} Cpp_Type?(volatile(t')) </div> <div style="display: flex; align-items: flex-start;"> {-6} cv(bitfield?)(volatile(t')) </div> <div style="display: flex; align-items: flex-start;"> {-7} typ' = volatile(t') </div> </div>	<hr style="border: 0.5px solid black;"/> <div style="display: flex; align-items: flex-start;"> {1} $\forall (d: ((\text{range_bitfield}(\text{cv_base}(\text{typ}')))), a: \text{Address}):$ $\text{down}(\text{from_byte}(\text{dt_volatile}[\text{range_bitfield}(\text{cv_base}(t'))](v'(t'))))$ $(\text{to_byte}(\text{dt_volatile}[\text{range_bitfield}(\text{cv_base}(t'))](v'(t'))))$ $(d, a),$ $a)$ $= d$ </div>
---	---

Repeatedly Skolemizing and flattening,
Instantiating quantified variables,
Replacing using formula -7,
Expanding the definition of cv_base,
which is trivially true.

This completes the proof of dt_cv_bitfield_TCC13.2.1.2.

dt_cv_bitfield_TCC13.2.2:

<div style="display: flex; flex-direction: column; gap: 5px;"> <div style="display: flex; align-items: flex-start;"> {-1} Cpp_Type?(volatile(t')) </div> <div style="display: flex; align-items: flex-start;"> {-2} cv(bitfield?)(volatile(t')) </div> <div style="display: flex; align-items: flex-start;"> {-3} typ' = volatile(t') </div> </div>	<hr style="border: 0.5px solid black;"/> <div style="display: flex; align-items: flex-start;"> {1} $\exists (x_1: (\text{interpreted_data_type?}[\text{range_bitfield}(\text{cv_base}(t'))])): \text{TRUE}$ </div> <div style="display: flex; align-items: flex-start;"> {2} $\text{interpreted_data_type?}[\text{range_bitfield}(\text{cv_base}(\text{typ}'))]$ $(\text{dt_volatile}[\text{range_bitfield}(\text{cv_base}(t'))])(v'(t'))$ </div>
---	--

Using lemma dt_cv_bitfield_TCC12,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of dt_cv_bitfield_TCC13.2.2.

dt_cv_bitfield_TCC13.2.3:

<div style="display: flex; flex-direction: column; gap: 5px;"> <div style="display: flex; align-items: flex-start;"> {-1} Cpp_Type?(volatile(t')) </div> <div style="display: flex; align-items: flex-start;"> {-2} cv(bitfield?)(volatile(t')) </div> <div style="display: flex; align-items: flex-start;"> {-3} typ' = volatile(t') </div> </div>	<hr style="border: 0.5px solid black;"/> <div style="display: flex; align-items: flex-start;"> {1} bitfield?(cv_base(t')) </div> <div style="display: flex; align-items: flex-start;"> {2} $\text{interpreted_data_type?}[\text{range_bitfield}(\text{cv_base}(\text{typ}'))]$ $(\text{dt_volatile}[\text{range_bitfield}(\text{cv_base}(t'))])(v'(t'))$ </div>
---	---

Using lemma dt_cv_bitfield_TCC11,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of dt_cv_bitfield_TCC13.2.3.

Q.E.D.

C.44.67 Cpp_Types.dt_cv_bitfield_TCC14

Terse proof for dt_cv_bitfield_TCC14.

dt_cv_bitfield_TCC14:

$\{1\} \quad \forall (\text{typ}: \text{Cpp_Subtype}(\text{cv}(\text{bitfield?}))) : \\ \neg \text{volatile?}(\text{typ}) \wedge \neg \text{const?}(\text{typ}) \supset \text{bitfield?}(\text{typ})$
--

Repeatedly Skolemizing and flattening,

Hiding formulas: -1,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of dt_cv_bitfield_TCC14.

Q.E.D.

C.44.68 Cpp_Types.dt_cv_bitfield_TCC15

Terse proof for dt_cv_bitfield_TCC15.

dt_cv_bitfield_TCC15:

$\{1\} \quad \forall (\text{typ}: \text{Cpp_Subtype}(\text{cv}(\text{bitfield?}))) : \\ \neg \text{volatile?}(\text{typ}) \wedge \neg \text{const?}(\text{typ}) \supset \\ (\forall (x: \text{int}): \text{range_bitfield}(\text{typ})(x) \equiv \text{range_bitfield}(\text{cv_base}(\text{typ}))(x)) \wedge \\ (\forall (x: \text{int}): \text{range_bitfield}(\text{typ})(x) \equiv \text{range_bitfield}(\text{cv_base}(\text{typ}))(x)) \wedge \\ (\forall (x_1: [\text{list}[\text{Byte}], \text{Address}]): \\ \text{every}[(\text{range_bitfield}(\text{typ}))] \\ ((\text{range_bitfield}(\text{cv_base}(\text{typ}))) (\text{dt_bitfield}(\text{typ}) \text{from_byte}(x_1))) \\ \wedge \text{interpreted_data_type?}[(\text{range_bitfield}(\text{cv_base}(\text{typ}))) (\text{dt_bitfield}(\text{typ}))])$
--

Repeatedly Skolemizing and flattening,

Using lemma cv_base_result2,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 3 subgoals:

dt_cv_bitfield_TCC15.1:

$\{-1\} \quad \text{cv_base}(\text{typ}') = \text{typ}' \\ \{-2\} \quad \text{Cpp_Type?}(\text{typ}') \\ \{-3\} \quad \text{cv}(\text{bitfield?})(\text{typ}')$
$\{1\} \quad \text{const?}(\text{typ}') \\ \{2\} \quad \text{volatile?}(\text{typ}') \\ \{3\} \quad \text{interpreted_data_type?}[(\text{range_bitfield}(\text{cv_base}(\text{typ}')))](\text{dt_bitfield}(\text{typ}'))$

Replacing using formula -1,

Adding type constraints for dt_bitfield(cv_base(typ'1)),

which is trivially true.

This completes the proof of dt_cv_bitfield_TCC15.1.

dt_cv_bitfield_TCC15.2:

{-1}	cv_base(typ') = typ'
{-2}	Cpp_Type?(typ')
{-3}	cv(bitfield?)(typ')
{1}	const?(typ')
{2}	volatile?(typ')
{3}	$\forall (x_1: [\text{list}[\text{Byte}], \text{Address}]):$ every[$((\text{range_bitfield}(\text{typ}')))$] $((\text{range_bitfield}(\text{cv_base}(\text{typ}'))))(\text{dt_bitfield}(\text{typ}') \text{ 'from_byte}(x_1))$

Expanding the definition of every,
 Repeatedly Skolemizing and flattening,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 This completes the proof of dt_cv_bitfield_TCC15.2.

dt_cv_bitfield_TCC15.3:

{-1}	cv_base(typ') = typ'
{-2}	Cpp_Type?(typ')
{-3}	cv(bitfield?)(typ')
{1}	const?(typ')
{2}	volatile?(typ')
{3}	$\forall (x: \text{int}): \text{range_bitfield}(\text{typ}')(x) \equiv \text{range_bitfield}(\text{cv_base}(\text{typ}'))(x)$

Repeatedly Skolemizing and flattening,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 This completes the proof of dt_cv_bitfield_TCC15.3.
 Q.E.D.

C.44.69 Cpp_Types.dt_cv_ptm_TCC1

Terse proof for dt_cv_ptm_TCC1.

dt_cv_ptm_TCC1:

{1}	well_founded? $(\lambda (x: \text{Cpp_Subtype}(\text{cv}(\text{pointer_to_member?})),$ $y: \text{Cpp_Subtype}(\text{cv}(\text{pointer_to_member?})))$): restrict $[[\text{Cpp_Type_}, \text{Cpp_Type_}],$ $[\text{Cpp_Subtype}(\text{cv}(\text{pointer_to_member?})),$ $\text{Cpp_Subtype}(\text{cv}(\text{pointer_to_member?}))]$], boolean] $(\ll)(x, y)$
-----	--

Using lemma well_founded_restrict[Cpp_Type_, Cpp_Subtype(cv(pointer_to_member?))],
 Expanding the definition of restrict,
 which is trivially true.
 This completes the proof of dt_cv_ptm_TCC1.
 Q.E.D.

C.44.70 Cpp_Types.dt_cv_ptm_TCC2

Terse proof for dt_cv_ptm_TCC2.

dt_cv_ptm_TCC2:

$\{1\} \quad \forall (\text{typ}: \text{Cpp_Subtype}(\text{cv}(\text{pointer_to_member?})), t: \text{Cpp_Type_}):$ $\text{typ} = \text{const}(t) \supset \text{Cpp_Type?}(t) \wedge \text{cv}(\text{pointer_to_member?})(t)$
--

Repeatedly Skolemizing and flattening,

Applying propositional simplification,

we get 2 subgoals:

dt_cv_ptm_TCC2.1:

$\{-1\} \quad \text{Cpp_Type?}(\text{typ}')$ $\{-2\} \quad \text{cv}(\text{pointer_to_member?})(\text{typ}')$ $\{-3\} \quad \text{typ}' = \text{const}(t')$
$\{1\} \quad \text{Cpp_Type?}(t')$

Expanding the definition of Cpp_Type?,

Repeatedly Skolemizing and flattening,

Instantiating the top quantifier in -2 with the terms: (t!2),

Expanding the definition of subterm,

Replacing using formula -4,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of dt_cv_ptm_TCC2.1.

dt_cv_ptm_TCC2.2:

$\{-1\} \quad \text{Cpp_Type?}(\text{typ}')$ $\{-2\} \quad \text{cv}(\text{pointer_to_member?})(\text{typ}')$ $\{-3\} \quad \text{typ}' = \text{const}(t')$
$\{1\} \quad \text{cv}(\text{pointer_to_member?})(t')$

Hiding formulas: -1,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of dt_cv_ptm_TCC2.2.

Q.E.D.

C.44.71 Cpp_Types.dt_cv_ptm_TCC3

Terse proof for dt_cv_ptm_TCC3.

dt_cv_ptm_TCC3:

$\{1\} \quad \forall (\text{typ}: \text{Cpp_Subtype}(\text{cv}(\text{pointer_to_member?})), t: \text{Cpp_Type_}):$ $\text{typ} = \text{const}(t) \supset$ restrict $\quad [[\text{Cpp_Type_}, \text{Cpp_Type_}],$ $\quad [\text{Cpp_Subtype}(\text{cv}(\text{pointer_to_member?})), \text{Cpp_Subtype}(\text{cv}(\text{pointer_to_member?}))],$ $\quad \text{boolean}]$ $(\ll)(t, \text{typ})$
--

Expanding the definition of restrict,

Repeatedly Skolemizing and flattening,

Replacing using formula -3,

Expanding the definition of ■,

which is trivially true.

This completes the proof of dt_cv_ptm_TCC3.

Q.E.D.

C.44.72 Cpp_Types.dt_cv_ptm_TCC4

Terse proof for dt_cv_ptm_TCC4.

dt_cv_ptm_TCC4:

{1}	$\forall (\text{typ}: \text{Cpp_Subtype}(\text{cv}(\text{pointer_to_member?})), t: \text{Cpp_Type_}):$ $\text{typ} = \text{const}(t) \supset \text{Cpp_Type?}(t)$
-----	--

Repeatedly Skolemizing and flattening,

Using lemma dt_cv_ptm_TCC2,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of dt_cv_ptm_TCC4.

Q.E.D.

C.44.73 Cpp_Types.dt_cv_ptm_TCC5

Terse proof for dt_cv_ptm_TCC5.

dt_cv_ptm_TCC5:

{1}	$\forall (\text{typ}: \text{Cpp_Subtype}(\text{cv}(\text{pointer_to_member?})), t: \text{Cpp_Type_}):$ $\text{typ} = \text{const}(t) \supset \text{pointer_to_member?}(\text{cv_base}(t))$
-----	---

Repeatedly Skolemizing and flattening,

Using lemma cv_base_result,

we get 2 subgoals:

dt_cv_ptm_TCC5.1:

{-1}	Cpp_Type?(typ')
{-2}	cv(pointer_to_member?)(typ')
{-3}	typ' = const(t')
{1}	cv(pointer_to_member?)(t')
{2}	pointer_to_member?(cv_base(t'))

Using lemma dt_cv_ptm_TCC2,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of dt_cv_ptm_TCC5.1.

dt_cv_ptm_TCC5.2:

{-1}	Cpp_Type?(typ')
{-2}	cv(pointer_to_member?)(typ')
{-3}	typ' = const(t')
{1}	(pointer_to_member? \subseteq interpreted?)
{2}	pointer_to_member?(cv_base(t'))

Keeping (1) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of dt_cv_ptm_TCC5.2.

Q.E.D.

C.44.74 Cpp_Types.dt_cv_ptm_TCC6

Terse proof for dt_cv_ptm_TCC6.

dt_cv_ptm_TCC6:

<div style="border-bottom: 1px solid black; padding-bottom: 5px;"> <p>{1} \forall (typ: Cpp_Subtype(cv(pointer_to_member?)), t: Cpp_Type_): typ = const(t) \supset (\exists (x₁: (interpreted_data_type?[(range_ptm(cv_base(t))]))): TRUE)</p> </div>	
---	--

Repeatedly Skolemizing and flattening,

Using lemma dt_pointer_to_member_exists,

we get 3 subgoals:

dt_cv_ptm_TCC6.1:

<div style="border-bottom: 1px solid black; padding-bottom: 5px;"> <p>{-1} \exists (dt: (pod_data_type?[(range_ptm(cv_base(t')))])): TRUE {-2} Cpp_Type?(typ') {-3} cv(pointer_to_member?)(typ') {-4} typ' = const(t')</p> </div>	<div style="border-bottom: 1px solid black; padding-bottom: 5px;"> <p>{1} \exists (x₁: (interpreted_data_type?[(range_ptm(cv_base(t')))])): TRUE</p> </div>
---	---

Repeatedly Skolemizing and flattening,

Instantiating quantified variables,

This completes the proof of dt_cv_ptm_TCC6.1.

dt_cv_ptm_TCC6.2:

<div style="border-bottom: 1px solid black; padding-bottom: 5px;"> <p>{-1} Cpp_Type?(typ') {-2} cv(pointer_to_member?)(typ') {-3} typ' = const(t')</p> </div>	<div style="border-bottom: 1px solid black; padding-bottom: 5px;"> <p>{1} pointer_to_member?(cv_base(t')) {2} \exists (x₁: (interpreted_data_type?[(range_ptm(cv_base(t')))])): TRUE</p> </div>
---	--

Using lemma dt_cv_ptm_TCC5,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of dt_cv_ptm_TCC6.2.

dt_cv_ptm_TCC6.3:

<div style="border-bottom: 1px solid black; padding-bottom: 5px;"> <p>{-1} Cpp_Type?(typ') {-2} cv(pointer_to_member?)(typ') {-3} typ' = const(t')</p> </div>	<div style="border-bottom: 1px solid black; padding-bottom: 5px;"> <p>{1} Cpp_Type?(t') {2} \exists (x₁: (interpreted_data_type?[(range_ptm(cv_base(t')))])): TRUE</p> </div>
---	--

Using lemma dt_cv_ptm_TCC2,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of dt_cv_ptm_TCC6.3.

Q.E.D.

C.44.75 Cpp_Types.dt_cv_ptm_TCC7

Terse proof for dt_cv_ptm_TCC7.

dt_cv_ptm_TCC7:

$\{1\} \quad \forall (\text{typ}: \text{Cpp_Subtype}(\text{cv}(\text{pointer_to_member?})),$ $v:$ $\quad [\text{typ1}:$ $\quad \quad \{z: \text{Cpp_Subtype}(\text{cv}(\text{pointer_to_member?})) \mid$ $\quad \quad \quad \text{restrict}$ $\quad \quad \quad \quad [[\text{Cpp_Type_}, \text{Cpp_Type_}],$ $\quad \quad \quad \quad [\text{Cpp_Subtype}(\text{cv}(\text{pointer_to_member?})),$ $\quad \quad \quad \quad \text{Cpp_Subtype}(\text{cv}(\text{pointer_to_member?}))],$ $\quad \quad \quad \text{boolean}]$ $\quad \quad \quad (\ll)(z, \text{typ})\} \rightarrow$ $\quad \quad \quad (\text{interpreted_data_type?}[[(\text{range_ptm}(\text{cv_base}(\text{typ1})))]]),$ $t: \text{Cpp_Type_}):$ $\text{typ} = \text{const}(t) \supset$ $(\forall (x: \text{nat}): \text{range_ptm}(\text{cv_base}(t))(x) \equiv \text{range_ptm}(\text{cv_base}(\text{typ}))(x)) \wedge$ $(\forall (x: \text{nat}): \text{range_ptm}(\text{cv_base}(t))(x) \equiv \text{range_ptm}(\text{cv_base}(\text{typ}))(x)) \wedge$ $(\forall (x_1: [\text{list}[\text{Byte}], \text{Address}]):$ $\quad \text{every}[[(\text{range_ptm}(\text{cv_base}(t)))]$ $\quad \quad ((\text{range_ptm}(\text{cv_base}(\text{typ})))$ $\quad \quad (\text{dt_const}[[(\text{range_ptm}(\text{cv_base}(t)))](v(t)) \text{'from_byte}(x_1)])$ $\quad \wedge$ $\quad \text{interpreted_data_type?}[[\text{range_ptm}(\text{cv_base}(\text{typ}))]]$ $\quad \quad (\text{dt_const}[[(\text{range_ptm}(\text{cv_base}(t)))](v(t))])$

Repeatedly Skolemizing and flattening,
 Replacing using formula -3,
 Expanding the definition of cv_base,
 Applying propositional simplification,
 we get 2 subgoals:

dt_cv_ptm_TCC7.1:

$\{-1\} \quad \text{Cpp_Type?}(\text{const}(t'))$ $\{-2\} \quad \text{cv}(\text{pointer_to_member?})(\text{const}(t'))$ $\{-3\} \quad \text{typ}' = \text{const}(t')$
$\{1\} \quad \forall (x_1: [\text{list}[\text{Byte}], \text{Address}]):$ $\quad \text{every}[[(\text{range_ptm}(\text{cv_base}(t')))]$ $\quad \quad ((\text{range_ptm}(\text{cv_base}(\text{const}(t')))))$ $\quad \quad (\text{dt_const}[[(\text{range_ptm}(\text{cv_base}(t')))](v'(t')) \text{'from_byte}(x_1)])$

Expanding the definition of every,
 Repeatedly Skolemizing and flattening,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 Expanding the definition of cv_base,
 which is trivially true.
 This completes the proof of dt_cv_ptm_TCC7.1.

dt_cv_ptm_TCC7.2:

$\{-1\} \quad \text{Cpp_Type?}(\text{const}(t'))$ $\{-2\} \quad \text{cv}(\text{pointer_to_member?})(\text{const}(t'))$ $\{-3\} \quad \text{typ}' = \text{const}(t')$
$\{1\} \quad \text{interpreted_data_type?}[[(\text{range_ptm}(\text{cv_base}(\text{typ}')))]$ $\quad \quad (\text{dt_const}[[(\text{range_ptm}(\text{cv_base}(t')))](v'(t'))])$

Adding type constraints for $\text{dt_const}[\text{((range_ptm(cv_base(t!1))))}](v!1(t!1))$,

we get 3 subgoals:

$\text{dt_cv_ptm_TCC7.2.1.1}$:

{-1}	interpreted_data_type? [((range_ptm(cv_base(t'))))] (dt_const [((range_ptm(cv_base(t')))] (v'(t'))))
{-2}	Cpp_Type?(const(t'))
{-3}	cv(pointer_to_member?)(const(t'))
{-4}	typ' = const(t')
{1}	interpreted_data_type? [((range_ptm(cv_base(typ'))))] (dt_const [((range_ptm(cv_base(t')))] (v'(t'))))

Expanding the definition of `interpreted_data_type?`,

Applying propositional simplification,

we get 2 subgoals:

$\text{dt_cv_ptm_TCC7.2.1.1.1}$:

{-1}	uninterpreted_data_type?(dt_const [((range_ptm(cv_base(t')))] (v'(t')) 'uidt)
{-2}	$\forall (d: ((\text{range_ptm}(\text{cv_base}(t')))), a: \text{Address}):$ valid?(uidt(dt_const [((range_ptm(cv_base(t')))] (v'(t')))) (to_byte(dt_const [((range_ptm(cv_base(t')))] (v'(t')))(d, a), a)
{-3}	$\forall (l: \text{list}[\text{Byte}], a: \text{Address}):$ valid?(uidt(dt_const [((range_ptm(cv_base(t')))] (v'(t')))(l, a) \equiv up?(from_byte(dt_const [((range_ptm(cv_base(t')))] (v'(t')))(l, a))
{-4}	$\forall (d: ((\text{range_ptm}(\text{cv_base}(t')))), a: \text{Address}):$ down(from_byte(dt_const [((range_ptm(cv_base(t')))] (v'(t'))) (to_byte(dt_const [((range_ptm(cv_base(t')))] (v'(t')))(d, a), a)) = d
{-5}	Cpp_Type?(const(t'))
{-6}	cv(pointer_to_member?)(const(t'))
{-7}	typ' = const(t')
{1}	$\forall (d: ((\text{range_ptm}(\text{cv_base}(\text{typ'}))), a: \text{Address}):$ valid?(uidt(dt_const [((range_ptm(cv_base(t')))] (v'(t')))) (to_byte(dt_const [((range_ptm(cv_base(t')))] (v'(t')))(d, a), a)

Repeatedly Skolemizing and flattening,

Instantiating quantified variables,

Replacing using formula -7,

Expanding the definition of `cv_base`,

which is trivially true.

This completes the proof of $\text{dt_cv_ptm_TCC7.2.1.1}$.

dt_cv_ptm_TCC7.2.1.2:

<pre> {-1} uninterpreted_data_type?(dt_const[((range_ptm(cv_base(t')))](v'(t')))'uidt) {-2} ∀ (d: (((range_ptm(cv_base(t'))))), a: Address): valid?(uidt(dt_const[((range_ptm(cv_base(t')))](v'(t')))) (to_byte(dt_const[((range_ptm(cv_base(t')))](v'(t')))(d, a), a) {-3} ∀ (l: list[Byte], a: Address): valid?(uidt(dt_const[((range_ptm(cv_base(t')))](v'(t')))(l, a) ≡ up?(from_byte(dt_const[((range_ptm(cv_base(t')))](v'(t')))(l, a) {-4} ∀ (d: (((range_ptm(cv_base(t'))))), a: Address): down(from_byte(dt_const[((range_ptm(cv_base(t')))](v'(t')))) (to_byte(dt_const[((range_ptm(cv_base(t')))](v'(t')))(d, a), a)) = d {-5} Cpp_Type?(const(t')) {-6} cv(pointer_to_member?)(const(t')) {-7} typ' = const(t') </pre>	<pre> {1} ∀ (d: (((range_ptm(cv_base(typ'))))), a: Address): down(from_byte(dt_const[((range_ptm(cv_base(t')))](v'(t')))) (to_byte(dt_const[((range_ptm(cv_base(t')))](v'(t')))(d, a), a)) = d </pre>
---	--

Repeatedly Skolemizing and flattening,
Instantiating quantified variables,
Replacing using formula -7,
Expanding the definition of cv_base,
which is trivially true.

This completes the proof of dt_cv_ptm_TCC7.2.1.2.

dt_cv_ptm_TCC7.2.2:

<pre> {-1} Cpp_Type?(const(t')) {-2} cv(pointer_to_member?)(const(t')) {-3} typ' = const(t') </pre>	<pre> {1} ∃ (x1: (interpreted_data_type?[((range_ptm(cv_base(t')))])): TRUE {2} interpreted_data_type?[((range_ptm(cv_base(typ')))] (dt_const[((range_ptm(cv_base(t')))](v'(t')))) </pre>
--	---

Using lemma dt_cv_ptm_TCC6,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of dt_cv_ptm_TCC7.2.2.

dt_cv_ptm_TCC7.2.3:

<pre> {-1} Cpp_Type?(const(t')) {-2} cv(pointer_to_member?)(const(t')) {-3} typ' = const(t') </pre>	<pre> {1} pointer_to_member?(cv_base(t')) {2} interpreted_data_type?[((range_ptm(cv_base(typ')))] (dt_const[((range_ptm(cv_base(t')))](v'(t')))) </pre>
--	---

Using lemma dt_cv_ptm_TCC5,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of dt_cv_ptm_TCC7.2.3.

Q.E.D.

C.44.76 Cpp_Types.dt_cv_ptm_TCC8

Terse proof for dt_cv_ptm_TCC8.

dt_cv_ptm_TCC8:

{1}	$\forall (\text{typ}: \text{Cpp_Subtype}(\text{cv}(\text{pointer_to_member?})), t: \text{Cpp_Type_}):$ $\text{typ} = \text{volatile}(t) \supset \text{Cpp_Type?}(t) \wedge \text{cv}(\text{pointer_to_member?})(t)$
-----	--

Repeatedly Skolemizing and flattening,

Applying propositional simplification,

we get 2 subgoals:

dt_cv_ptm_TCC8.1:

{-1}	Cpp_Type?(typ')
{-2}	cv(pointer_to_member?)(typ')
{-3}	typ' = volatile(t')
{1}	Cpp_Type?(t')

Expanding the definition of Cpp_Type?,

Repeatedly Skolemizing and flattening,

Instantiating the top quantifier in -2 with the terms: (t!2),

Expanding the definition of subterm,

Replacing using formula -4,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of dt_cv_ptm_TCC8.1.

dt_cv_ptm_TCC8.2:

{-1}	Cpp_Type?(typ')
{-2}	cv(pointer_to_member?)(typ')
{-3}	typ' = volatile(t')
{1}	cv(pointer_to_member?)(t')

Hiding formulas: -1,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of dt_cv_ptm_TCC8.2.

Q.E.D.

C.44.77 Cpp_Types.dt_cv_ptm_TCC9

Terse proof for dt_cv_ptm_TCC9.

dt_cv_ptm_TCC9:

{1}	$\forall (\text{typ}: \text{Cpp_Subtype}(\text{cv}(\text{pointer_to_member?})), t: \text{Cpp_Type_}):$ $\text{typ} = \text{volatile}(t) \supset$ restrict $\quad [[\text{Cpp_Type_}, \text{Cpp_Type_}],$ $\quad \quad [\text{Cpp_Subtype}(\text{cv}(\text{pointer_to_member?})), \text{Cpp_Subtype}(\text{cv}(\text{pointer_to_member?}))],$ $\quad \quad \text{boolean}]$ $\quad (\ll)(t, \text{typ})$
-----	---

Expanding the definition of restrict,

Repeatedly Skolemizing and flattening,

Replacing using formula -3,

Expanding the definition of ■,

which is trivially true.

This completes the proof of `dt_cv_ptm_TCC9`.

Q.E.D.

C.44.78 Cpp_Types.dt_cv_ptm_TCC10

Terse proof for `dt_cv_ptm_TCC10`.

`dt_cv_ptm_TCC10`:

{1}	$\forall (\text{typ}: \text{Cpp_Subtype}(\text{cv}(\text{pointer_to_member?})), t: \text{Cpp_Type_}):$ $\text{typ} = \text{volatile}(t) \supset \text{Cpp_Type?}(t)$
-----	---

Repeatedly Skolemizing and flattening,

Using lemma `dt_cv_ptm_TCC8`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `dt_cv_ptm_TCC10`.

Q.E.D.

C.44.79 Cpp_Types.dt_cv_ptm_TCC11

Terse proof for `dt_cv_ptm_TCC11`.

`dt_cv_ptm_TCC11`:

{1}	$\forall (\text{typ}: \text{Cpp_Subtype}(\text{cv}(\text{pointer_to_member?})), t: \text{Cpp_Type_}):$ $\text{typ} = \text{volatile}(t) \supset \text{pointer_to_member?}(\text{cv_base}(t))$
-----	--

Repeatedly Skolemizing and flattening,

Rewriting using `cv_base_result`, matching in `*`,

we get 2 subgoals:

`dt_cv_ptm_TCC11.1`:

{-1}	<code>Cpp_Type?(typ')</code>
{-2}	<code>cv(pointer_to_member?)(typ')</code>
{-3}	<code>typ' = volatile(t')</code>
{1}	<code>cv(pointer_to_member?)(t')</code>
{2}	<code>pointer_to_member?(cv_base(t'))</code>

Using lemma `dt_cv_ptm_TCC8`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `dt_cv_ptm_TCC11.1`.

`dt_cv_ptm_TCC11.2`:

{-1}	<code>Cpp_Type?(typ')</code>
{-2}	<code>cv(pointer_to_member?)(typ')</code>
{-3}	<code>typ' = volatile(t')</code>
{1}	<code>(pointer_to_member? \subseteq interpreted?)</code>
{2}	<code>pointer_to_member?(cv_base(t'))</code>

Keeping (1) and hiding `*`,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `dt_cv_ptm_TCC11.2`.

Q.E.D.

C.44.80 Cpp_Types.dt_cv_ptm_TCC12

Terse proof for dt_cv_ptm_TCC12.

dt_cv_ptm_TCC12:

$\{1\} \quad \forall (\text{typ}: \text{Cpp_Subtype}(\text{cv}(\text{pointer_to_member?})), t: \text{Cpp_Type_}):$ $\text{typ} = \text{volatile}(t) \supset$ $(\exists (x_1: (\text{interpreted_data_type?}[\text{((range_ptm}(\text{cv_base}(t))))]))): \text{TRUE}$	
--	--

Repeatedly Skolemizing and flattening,

Using lemma dt_pointer_to_member_exists,

we get 3 subgoals:

dt_cv_ptm_TCC12.1:

$\{-1\} \quad \exists (\text{dt}: (\text{pod_data_type?}[\text{((range_ptm}(\text{cv_base}(t')))]))): \text{TRUE}$ $\{-2\} \quad \text{Cpp_Type?}(\text{typ}')$ $\{-3\} \quad \text{cv}(\text{pointer_to_member?})(\text{typ}')$ $\{-4\} \quad \text{typ}' = \text{volatile}(t')$	
$\{1\} \quad \exists (x_1: (\text{interpreted_data_type?}[\text{((range_ptm}(\text{cv_base}(t')))]))): \text{TRUE}$	

Repeatedly Skolemizing and flattening,

Instantiating quantified variables,

This completes the proof of dt_cv_ptm_TCC12.1.

dt_cv_ptm_TCC12.2:

$\{-1\} \quad \text{Cpp_Type?}(\text{typ}')$ $\{-2\} \quad \text{cv}(\text{pointer_to_member?})(\text{typ}')$ $\{-3\} \quad \text{typ}' = \text{volatile}(t')$	
$\{1\} \quad \text{pointer_to_member?}(\text{cv_base}(t'))$ $\{2\} \quad \exists (x_1: (\text{interpreted_data_type?}[\text{((range_ptm}(\text{cv_base}(t')))]))): \text{TRUE}$	

Using lemma dt_cv_ptm_TCC11,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of dt_cv_ptm_TCC12.2.

dt_cv_ptm_TCC12.3:

$\{-1\} \quad \text{Cpp_Type?}(\text{typ}')$ $\{-2\} \quad \text{cv}(\text{pointer_to_member?})(\text{typ}')$ $\{-3\} \quad \text{typ}' = \text{volatile}(t')$	
$\{1\} \quad \text{Cpp_Type?}(t')$ $\{2\} \quad \exists (x_1: (\text{interpreted_data_type?}[\text{((range_ptm}(\text{cv_base}(t')))]))): \text{TRUE}$	

Using lemma dt_cv_ptm_TCC8,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of dt_cv_ptm_TCC12.3.

Q.E.D.

C.44.81 Cpp_Types.dt_cv_ptm_TCC13

Terse proof for dt_cv_ptm_TCC13.

dt_cv_ptm_TCC13:

<pre> {1} ∃ (typ: Cpp_Subtype(cv(pointer_to_member?)), v: [typ1: {z: Cpp_Subtype(cv(pointer_to_member?)) restrict [[Cpp_Type_, Cpp_Type_], [Cpp_Subtype(cv(pointer_to_member?)), Cpp_Subtype(cv(pointer_to_member?))], boolean] (≪≪)(z, typ)} → (interpreted_data_type?[(range_ptm(cv_base(typ1)))]), t: Cpp_Type_): typ = volatile(t) ⊃ (∀ (x: nat): range_ptm(cv_base(t))(x) ≡ range_ptm(cv_base(typ))(x)) ∧ (∀ (x: nat): range_ptm(cv_base(t))(x) ≡ range_ptm(cv_base(typ))(x)) ∧ (∀ (x₁: [list[Byte], Address]): every[(range_ptm(cv_base(t)))] ((range_ptm(cv_base(typ)))) (dt_volatile[(range_ptm(cv_base(t)))](v(t)) 'from_byte(x₁))) ∧ interpreted_data_type?[(range_ptm(cv_base(typ)))] (dt_volatile[(range_ptm(cv_base(t)))](v(t))) </pre>

Repeatedly Skolemizing and flattening,

Expanding the definition of cv_base,

Replacing using formula -3,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 2 subgoals:

dt_cv_ptm_TCC13.1:

<pre> {-1} Cpp_Type?(volatile(t')) {-2} cv(pointer_to_member?)(volatile(t')) {-3} typ' = volatile(t') </pre> <hr/> <pre> {1} interpreted_data_type?[(range_ptm(cv_base(typ')))] (dt_volatile[(range_ptm(cv_base(t')))](v'(t'))) </pre>

Adding type constraints for dt_volatile[(range_ptm(cv_base(t!)))](v!1(t!1)),

we get 3 subgoals:

dt_cv_ptm_TCC13.1.1:

<pre> {-1} interpreted_data_type?[(range_ptm(cv_base(t')))] (dt_volatile[(range_ptm(cv_base(t')))](v'(t'))) {-2} Cpp_Type?(volatile(t')) {-3} cv(pointer_to_member?)(volatile(t')) {-4} typ' = volatile(t') </pre> <hr/> <pre> {1} interpreted_data_type?[(range_ptm(cv_base(typ')))] (dt_volatile[(range_ptm(cv_base(t')))](v'(t'))) </pre>

Expanding the definition of interpreted_data_type?,

Applying propositional simplification,

we get 2 subgoals:

dt_cv_ptm_TCC13.1.1.1:

<pre> {-1} uninterpreted_data_type?(dt_volatile[((range_ptm(cv_base(t')))](v'(t))'uidt) {-2} ∇ (d: (((range_ptm(cv_base(t')))), a: Address): valid?(uidt(dt_volatile[((range_ptm(cv_base(t')))](v'(t)))) (to_byte(dt_volatile[((range_ptm(cv_base(t')))](v'(t)))(d, a), a) {-3} ∇ (l: list[Byte], a: Address): valid?(uidt(dt_volatile[((range_ptm(cv_base(t')))](v'(t)))(l, a) ≡ up?(from_byte(dt_volatile[((range_ptm(cv_base(t')))](v'(t)))(l, a) {-4} ∇ (d: (((range_ptm(cv_base(t')))), a: Address): down(from_byte(dt_volatile[((range_ptm(cv_base(t')))](v'(t)) (to_byte(dt_volatile[((range_ptm(cv_base(t')))](v'(t)) (d, a), a)) = d {-5} Cpp_Type?(volatile(t')) {-6} cv(pointer_to_member?)(volatile(t')) {-7} typ' = volatile(t') </pre>	<pre> {1} ∇ (d: ((range_ptm(cv_base(typ')))), a: Address): valid?(uidt(dt_volatile[((range_ptm(cv_base(t')))](v'(t)))) (to_byte(dt_volatile[((range_ptm(cv_base(t')))](v'(t)))(d, a), a) </pre>
---	--

Repeatedly Skolemizing and flattening,

Instantiating quantified variables,

Replacing using formula -7,

Expanding the definition of cv_base,

which is trivially true.

This completes the proof of dt_cv_ptm_TCC13.1.1.1.

dt_cv_ptm_TCC13.1.1.2:

<pre> {-1} uninterpreted_data_type?(dt_volatile[((range_ptm(cv_base(t')))](v'(t))'uidt) {-2} ∇ (d: (((range_ptm(cv_base(t')))), a: Address): valid?(uidt(dt_volatile[((range_ptm(cv_base(t')))](v'(t)))) (to_byte(dt_volatile[((range_ptm(cv_base(t')))](v'(t)))(d, a), a) {-3} ∇ (l: list[Byte], a: Address): valid?(uidt(dt_volatile[((range_ptm(cv_base(t')))](v'(t)))(l, a) ≡ up?(from_byte(dt_volatile[((range_ptm(cv_base(t')))](v'(t)))(l, a) {-4} ∇ (d: (((range_ptm(cv_base(t')))), a: Address): down(from_byte(dt_volatile[((range_ptm(cv_base(t')))](v'(t)) (to_byte(dt_volatile[((range_ptm(cv_base(t')))](v'(t)) (d, a), a)) = d {-5} Cpp_Type?(volatile(t')) {-6} cv(pointer_to_member?)(volatile(t')) {-7} typ' = volatile(t') </pre>	<pre> {1} ∇ (d: ((range_ptm(cv_base(typ')))), a: Address): down(from_byte(dt_volatile[((range_ptm(cv_base(t')))](v'(t)) (to_byte(dt_volatile[((range_ptm(cv_base(t')))](v'(t)) (d, a), a)) = d </pre>
---	--

Repeatedly Skolemizing and flattening,

Instantiating quantified variables,
 Replacing using formula -7,
 Expanding the definition of cv_base,
 which is trivially true.
 This completes the proof of dt_cv_ptm_TCC13.1.1.2.
 dt_cv_ptm_TCC13.1.2:

<pre>{-1} Cpp_Type?(volatile(t')) {-2} cv(pointer_to_member?)(volatile(t')) {-3} typ' = volatile(t')</pre>	<pre>{1} $\exists (x_1 : (\text{interpreted_data_type?}[\text{((range_ptm(cv_base(t')))]))}) : \text{TRUE}$ {2} $\text{interpreted_data_type?}[\text{((range_ptm(cv_base(typ')))])]$ $(\text{dt_volatile}[\text{((range_ptm(cv_base(t')))])(v'(t'))})$</pre>
--	---

Using lemma dt_cv_ptm_TCC12,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 This completes the proof of dt_cv_ptm_TCC13.1.2.
 dt_cv_ptm_TCC13.1.3:

<pre>{-1} Cpp_Type?(volatile(t')) {-2} cv(pointer_to_member?)(volatile(t')) {-3} typ' = volatile(t')</pre>	<pre>{1} pointer_to_member?(cv_base(t')) {2} $\text{interpreted_data_type?}[\text{((range_ptm(cv_base(typ')))])]$ $(\text{dt_volatile}[\text{((range_ptm(cv_base(t')))])(v'(t'))})$</pre>
--	---

Using lemma dt_cv_ptm_TCC11,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 This completes the proof of dt_cv_ptm_TCC13.1.3.
 dt_cv_ptm_TCC13.2:

<pre>{-1} Cpp_Type?(volatile(t')) {-2} cv(pointer_to_member?)(volatile(t')) {-3} typ' = volatile(t')</pre>	<pre>{1} $\forall (x_1 : [\text{list}[\text{Byte}], \text{Address}]) :$ $\text{every}[\text{((range_ptm(cv_base(t')))]}$ $(\text{range_ptm(cv_base(volatile(t'))))}$ $(\text{dt_volatile}[\text{((range_ptm(cv_base(t')))])(v'(t'))} \text{'from_byte}(x_1))$</pre>
--	--

Repeatedly Skolemizing and flattening,
 Expanding the definition of every,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 Expanding the definition of cv_base,
 which is trivially true.
 This completes the proof of dt_cv_ptm_TCC13.2.
 Q.E.D.

C.44.82 Cpp_Types.dt_cv_ptm_TCC14

Terse proof for dt_cv_ptm_TCC14.
 dt_cv_ptm_TCC14:

<pre>{1}</pre>	<pre>$\forall (\text{typ} : \text{Cpp_Subtype}(\text{cv}(\text{pointer_to_member?}))) :$ $\neg \text{volatile?}(\text{typ}) \wedge \neg \text{const?}(\text{typ}) \supset \text{pointer_to_member?}(\text{typ})$</pre>
----------------	--

Repeatedly Skolemizing and flattening,
Hiding formulas: -1,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of dt_cv_ptm_TCC14.
Q.E.D.

C.44.83 Cpp_Types.dt_cv_ptm_TCC15

Terse proof for dt_cv_ptm_TCC15.

dt_cv_ptm_TCC15:

<pre> {1} \forall (typ: Cpp_Subtype(cv(pointer_to_member?))): \neg volatile?(typ) \wedge \neg const?(typ) \supset (\forall (x: nat): range_ptm(typ)(x) \equiv range_ptm(cv_base(typ))(x)) \wedge (\forall (x: nat): range_ptm(typ)(x) \equiv range_ptm(cv_base(typ))(x)) \wedge (\forall (x₁: [list[Byte], Address]): every[[(range_ptm(typ))]] ((range_ptm(cv_base(typ)))(dt_ptm(typ) 'from_byte(x₁))) \wedge interpreted_data_type?[(range_ptm(cv_base(typ)))](dt_ptm(typ)) </pre>
--

Repeatedly Skolemizing and flattening,
Using lemma cv_base_result2,
Simplifying, rewriting, and recording with decision procedures,
Replacing using formula -1,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
we get 3 subgoals:

dt_cv_ptm_TCC15.1:

<pre> {-1} cv_base(typ') = typ' {-2} Cpp_Type?(cv_base(typ')) {-3} cv(pointer_to_member?)(cv_base(typ')) </pre>
<pre> {1} volatile?(cv_base(typ')) {2} const?(cv_base(typ')) {3} interpreted_data_type?[(range_ptm(cv_base(typ')))](dt_ptm(cv_base(typ'))) </pre>

Adding type constraints for dt_ptm(cv_base(typ!1)),
Rewriting using pod_is_interpreted_data, matching in *,
This completes the proof of dt_cv_ptm_TCC15.1.

dt_cv_ptm_TCC15.2:

<pre> {-1} cv_base(typ') = typ' {-2} Cpp_Type?(cv_base(typ')) {-3} cv(pointer_to_member?)(cv_base(typ')) </pre>
<pre> {1} volatile?(cv_base(typ')) {2} const?(cv_base(typ')) {3} \forall (x₁: [list[Byte], Address]): every[[(range_ptm(typ'))]] ((range_ptm(cv_base(cv_base(typ')))))(dt_ptm(cv_base(typ')) 'from_byte(x₁)) </pre>

Expanding the definition of every,
Repeatedly Skolemizing and flattening,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of dt_cv_ptm_TCC15.2.

dt_cv_ptm_TCC15.3:

{-1}	cv_base(typ') = typ'
{-2}	Cpp_Type?(cv_base(typ'))
{-3}	cv(pointer_to_member?)(cv_base(typ'))
{1}	volatile?(cv_base(typ'))
{2}	const?(cv_base(typ'))
{3}	$\forall (x: \text{nat}):$ range_ptm(cv_base(typ'))(x) \equiv range_ptm(cv_base(cv_base(typ')))(x)

Repeatedly Skolemizing and flattening,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 This completes the proof of dt_cv_ptm_TCC15.3.
 Q.E.D.

C.44.84 Cpp_Types.dt_cv_float_TCC1

Terse proof for dt_cv_float_TCC1.

dt_cv_float_TCC1:

{1}	well_founded?($\lambda (x: \text{Cpp_Subtype}(cv(\text{floating_point?})),$ $y: \text{Cpp_Subtype}(cv(\text{floating_point?})))$): restrict $[[\text{Cpp_Type_}, \text{Cpp_Type_}],$ $[\text{Cpp_Subtype}(cv(\text{floating_point?})),$ $\text{Cpp_Subtype}(cv(\text{floating_point?}))]$], boolean] $(\ll)(x, y)$
-----	--

Using lemma well_founded_restrict[Cpp_Type_, Cpp_Subtype(cv(floating_point?))],
 Expanding the definition of restrict,
 which is trivially true.
 This completes the proof of dt_cv_float_TCC1.
 Q.E.D.

C.44.85 Cpp_Types.dt_cv_float_TCC2

Terse proof for dt_cv_float_TCC2.

dt_cv_float_TCC2:

{1}	$\forall (typ: \text{Cpp_Subtype}(cv(\text{floating_point?})), t: \text{Cpp_Type_}):$ $typ = \text{const}(t) \supset \text{Cpp_Type?}(t) \wedge cv(\text{floating_point?})(t)$
-----	---

Repeatedly Skolemizing and flattening,
 Applying propositional simplification,
 we get 2 subgoals:
 dt_cv_float_TCC2.1:

{-1}	Cpp_Type?(typ')
{-2}	cv(floating_point?)(typ')
{-3}	typ' = const(t')
{1}	Cpp_Type?(t')

Expanding the definition of Cpp_Type?,

Repeatedly Skolemizing and flattening,
 Instantiating the top quantifier in -2 with the terms: (t!2),
 Expanding the definition of subterm,
 Replacing using formula -4,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 This completes the proof of dt_cv_float_TCC2.1.
 dt_cv_float_TCC2.2:

{-1}	Cpp_Type?(typ')
{-2}	cv(floating_point?)(typ')
{-3}	typ' = const(t')
{1}	cv(floating_point?)(t')

Hiding formulas: -1,
 Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of dt_cv_float_TCC2.2.
 Q.E.D.

C.44.86 Cpp_Types.dt_cv_float_TCC3

Terse proof for dt_cv_float_TCC3.

dt_cv_float_TCC3:

{1}	\forall (typ: Cpp_Subtype(cv(floating_point?)), t: Cpp_Type_): typ = const(t) \supset restrict [[Cpp_Type_, Cpp_Type_], [Cpp_Subtype(cv(floating_point?)), Cpp_Subtype(cv(floating_point?))], boolean] (\ll)(t, typ)
-----	---

Expanding the definition of \ll ,
 Expanding the definition of restrict,
 Repeatedly Skolemizing and flattening,
 Replacing using formula -3,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 This completes the proof of dt_cv_float_TCC3.
 Q.E.D.

C.44.87 Cpp_Types.dt_cv_float_TCC4

Terse proof for dt_cv_float_TCC4.

dt_cv_float_TCC4:

{1}	\forall (typ: Cpp_Subtype(cv(floating_point?)), t: Cpp_Type_): typ = const(t) \supset Cpp_Type?(t)
-----	---

Repeatedly Skolemizing and flattening,
 Using lemma dt_cv_float_TCC2,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 This completes the proof of dt_cv_float_TCC4.
 Q.E.D.

C.44.88 Cpp_Types.dt_cv_float_TCC5

Terse proof for dt_cv_float_TCC5.

dt_cv_float_TCC5:

{1}	$\forall (typ: \text{Cpp_Subtype}(cv(\text{floating_point?})), t: \text{Cpp_Type_}):$ $typ = \text{const}(t) \supset \text{floating_point?}(cv_base(t))$
-----	---

Repeatedly Skolemizing and flattening,
Rewriting using cv_base_result, matching in *,
we get 2 subgoals:

dt_cv_float_TCC5.1:

{-1}	$\text{Cpp_Type?}(typ')$
{-2}	$cv(\text{floating_point?})(typ')$
{-3}	$typ' = \text{const}(t')$
{1}	$cv(\text{floating_point?})(t')$
{2}	$\text{floating_point?}(cv_base(t'))$

Using lemma dt_cv_float_TCC2,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of dt_cv_float_TCC5.1.

dt_cv_float_TCC5.2:

{-1}	$\text{Cpp_Type?}(typ')$
{-2}	$cv(\text{floating_point?})(typ')$
{-3}	$typ' = \text{const}(t')$
{1}	$(\text{floating_point?} \subseteq \text{interpreted?})$
{2}	$\text{floating_point?}(cv_base(t'))$

Keeping (1) and hiding *,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of dt_cv_float_TCC5.2.
Q.E.D.

C.44.89 Cpp_Types.dt_cv_float_TCC6

Terse proof for dt_cv_float_TCC6.

dt_cv_float_TCC6:

{1}	$\forall (typ: \text{Cpp_Subtype}(cv(\text{floating_point?})), t: \text{Cpp_Type_}):$ $typ = \text{const}(t) \supset$ $(\exists (x_1: (\text{interpreted_data_type?}[\text{((range_floating_point}(cv_base(t))))]))): \text{TRUE})$
-----	--

Repeatedly Skolemizing and flattening,
Instantiating the top quantifier in 1 with the terms: (dt_floating_point(cv_base(t!1))),
we get 3 subgoals:

dt_cv_float_TCC6.1:

{-1}	$\text{Cpp_Type?}(typ')$
{-2}	$cv(\text{floating_point?})(typ')$
{-3}	$typ' = \text{const}(t')$
{1}	TRUE

which is trivially true.

This completes the proof of `dt_cv_float_TCC6.1`.

`dt_cv_float_TCC6.2`:

{-1}	Cpp_Type?(typ')
{-2}	cv(floating_point?)(typ')
{-3}	typ' = const(t')
{1}	floating_point?(cv_base(t'))

Using lemma `dt_cv_float_TCC5`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `dt_cv_float_TCC6.2`.

`dt_cv_float_TCC6.3`:

{-1}	Cpp_Type?(typ')
{-2}	cv(floating_point?)(typ')
{-3}	typ' = const(t')
{1}	Cpp_Type?(t')

Using lemma `dt_cv_float_TCC2`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `dt_cv_float_TCC6.3`.

Q.E.D.

C.44.90 Cpp_Types.dt_cv_float_TCC7

Terse proof for `dt_cv_float_TCC7`.

dt_cv_float_TCC7:

```

{1}  ∀ (typ: Cpp_Subtype(cv(floating_point?)),
      v:
      [typ1:
        {z: Cpp_Subtype(cv(floating_point?)) |
          restrict
            [[CppType_, Cpp_Type_],
             [Cpp_Subtype(cv(floating_point?)),
              Cpp_Subtype(cv(floating_point?))],
             boolean]
            (≪)(z, typ)} →
          (interpreted_data_type?[(range_floating_point(cv_base(typ1))))]),
      t: Cpp_Type_):
typ = const(t) ⊃
(∀ (x: extended_real):
  range_floating_point(cv_base(t))(x) ≡
  range_floating_point(cv_base(typ))(x))
^
(∀ (x: extended_real):
  range_floating_point(cv_base(t))(x) ≡
  range_floating_point(cv_base(typ))(x))
^
(∀ (x1: [list[Byte], Address]):
  every [(range_floating_point(cv_base(t)))]
         ((range_floating_point(cv_base(typ))))
         (dt_const [(range_floating_point(cv_base(t)))](v(t)) 'from_byte
                    (x1)))
^
  interpreted_data_type?[(range_floating_point(cv_base(typ)))]
  (dt_const [(range_floating_point(cv_base(t)))](v(t)))

```

Repeatedly Skolemizing and flattening,
 Replacing using formula -3,
 Expanding the definition of cv_base,
 Applying propositional simplification,
 we get 2 subgoals:

dt_cv_float_TCC7.1:

```

{-1} Cpp_Type?(const(t'))
{-2} cv(floating_point?)(const(t'))
{-3} typ' = const(t')
{1}  ∀ (x1: [list[Byte], Address]):
      every [(range_floating_point(cv_base(t')))]
            ((range_floating_point(cv_base(const(t'))))
             (dt_const [(range_floating_point(cv_base(t')))](v'(t')) 'from_byte(x1))

```

Repeatedly Skolemizing and flattening,
 Expanding the definition of every,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 Expanding the definition of cv_base,
 which is trivially true.

This completes the proof of dt_cv_float_TCC7.1.

dt_cv_float_TCC7.2:

{-1}	Cpp_Type?(const(t'))
{-2}	cv(floating_point?)(const(t'))
{-3}	$\text{typ}' = \text{const}(t')$
{1}	$\text{interpreted_data_type?}[\text{((range_floating_point(cv_base(\text{typ}'))))}]$ $(\text{dt_const}[\text{((range_floating_point(cv_base}(t')))](v'(t')))$

Adding type constraints for $\text{dt_const}[\text{((range_floating_point(cv_base}(t!1))))](v!1(t!1))$,
we get 3 subgoals:

dt_cv_float_TCC7.2.1:

{-1}	$\text{interpreted_data_type?}[\text{((range_floating_point(cv_base}(t')))]$ $(\text{dt_const}[\text{((range_floating_point(cv_base}(t')))](v'(t')))$
{-2}	Cpp_Type?(const(t'))
{-3}	cv(floating_point?)(const(t'))
{-4}	$\text{typ}' = \text{const}(t')$
{1}	$\text{interpreted_data_type?}[\text{((range_floating_point(cv_base(\text{typ}'))))}]$ $(\text{dt_const}[\text{((range_floating_point(cv_base}(t')))](v'(t')))$

Expanding the definition of $\text{interpreted_data_type?}$,

Applying propositional simplification,

we get 2 subgoals:

dt_cv_float_TCC7.2.1.1:

{-1}	$\text{uninterpreted_data_type?}(\text{dt_const}[\text{((range_floating_point(cv_base}(t')))]$ $(v'(t'))' \text{uidt})$
{-2}	$\forall (d: (\text{((range_floating_point(cv_base}(t')))), a: \text{Address}):$ $\text{valid?}(\text{uidt}(\text{dt_const}[\text{((range_floating_point(cv_base}(t')))](v'(t'))))$ $(\text{to_byte}(\text{dt_const}[\text{((range_floating_point(cv_base}(t')))](v'(t'))))$ $(d, a),$
{-3}	$\forall (l: \text{list}[\text{Byte}], a: \text{Address}):$ $\text{valid?}(\text{uidt}(\text{dt_const}[\text{((range_floating_point(cv_base}(t')))](v'(t'))))$ (l, a) \equiv $\text{up?}(\text{from_byte}(\text{dt_const}[\text{((range_floating_point(cv_base}(t')))](v'(t'))))$ (l, a)
{-4}	$\forall (d: (\text{((range_floating_point(cv_base}(t')))), a: \text{Address}):$ $\text{down}(\text{from_byte}(\text{dt_const}[\text{((range_floating_point(cv_base}(t')))](v'(t'))))$ $(\text{to_byte}(\text{dt_const}[\text{((range_floating_point(cv_base}(t')))](v'(t'))))$ $(d, a),$ $a))$ $= d$
{-5}	Cpp_Type?(const(t'))
{-6}	cv(floating_point?)(const(t'))
{-7}	$\text{typ}' = \text{const}(t')$
{1}	$\forall (d: (\text{((range_floating_point(cv_base(\text{typ}'))))}, a: \text{Address}):$ $\text{valid?}(\text{uidt}(\text{dt_const}[\text{((range_floating_point(cv_base}(t')))](v'(t'))))$ $(\text{to_byte}(\text{dt_const}[\text{((range_floating_point(cv_base}(t')))](v'(t'))))$ $(d, a),$ $a)$

Repeatedly Skolemizing and flattening,

Instantiating quantified variables,

C Proof scripts

Replacing using formula -8,

Expanding the definition of `cv_base`,

which is trivially true.

This completes the proof of `dt_cv_float_TCC7.2.1.1`.

`dt_cv_float_TCC7.2.1.2`:

{-1}	<code>uninterpreted_data_type?(dt_const[[(range_floating_point(cv_base(t')))]</code> <code>(v'(t'))' uidt</code>)
{-2}	$\forall (d: (((range_floating_point(cv_base(t'))))), a: Address):$ <code>valid?(uidt(dt_const[[(range_floating_point(cv_base(t')))](v'(t'))</code> <code>(to_byte(dt_const[[(range_floating_point(cv_base(t')))](v'(t'))</code> <code>(d, a),</code> <code>a)</code>)
{-3}	$\forall (l: list[Byte], a: Address):$ <code>valid?(uidt(dt_const[[(range_floating_point(cv_base(t')))](v'(t'))</code> <code>(l, a)</code>)
	\equiv <code>up?(from_byte(dt_const[[(range_floating_point(cv_base(t')))](v'(t'))</code> <code>(l, a)</code>)
{-4}	$\forall (d: (((range_floating_point(cv_base(t'))))), a: Address):$ <code>down(from_byte(dt_const[[(range_floating_point(cv_base(t')))](v'(t'))</code> <code>(to_byte(dt_const[[(range_floating_point(cv_base(t')))](v'(t'))</code> <code>(d, a),</code> <code>a))</code>
{-5}	<code>= d</code> <code>Cpp_Type?(const(t'))</code>
{-6}	<code>cv(floating_point?(const(t'))</code>
{-7}	<code>typ' = const(t')</code>
{1}	$\forall (d: ((range_floating_point(cv_base(typ')))), a: Address):$ <code>down(from_byte(dt_const[[(range_floating_point(cv_base(t')))](v'(t'))</code> <code>(to_byte(dt_const[[(range_floating_point(cv_base(t')))](v'(t'))</code> <code>(d, a),</code> <code>a))</code>
	<code>= d</code>

Repeatedly Skolemizing and flattening,

Instantiating quantified variables,

Expanding the definition of `cv_base`,

Replacing using formula -8,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `dt_cv_float_TCC7.2.1.2`.

`dt_cv_float_TCC7.2.2`:

{-1}	<code>Cpp_Type?(const(t'))</code>
{-2}	<code>cv(floating_point?(const(t'))</code>
{-3}	<code>typ' = const(t')</code>
{1}	$\exists (x_1: (interpreted_data_type?([(range_floating_point(cv_base(t')))]))): TRUE$
{2}	<code>interpreted_data_type?([(range_floating_point(cv_base(typ')))]</code> <code>(dt_const[[(range_floating_point(cv_base(t')))](v'(t'))</code>)

Using lemma `dt_cv_float_TCC6`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `dt_cv_float_TCC7.2.2`.

dt_cv_float_TCC7.2.3:

{-1}	Cpp_Type?(const(t'))
{-2}	cv(floating_point?)(const(t'))
{-3}	typ' = const(t')
{1}	floating_point?(cv_base(t'))
{2}	interpreted_data_type?[((range_floating_point(cv_base(typ')))] (dt_const[((range_floating_point(cv_base(t')))](v' (t')))

Using lemma dt_cv_float_TCC5,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of dt_cv_float_TCC7.2.3.

Q.E.D.

C.44.91 Cpp_Types.dt_cv_float_TCC8

Terse proof for dt_cv_float_TCC8.

dt_cv_float_TCC8:

{1}	\forall (typ: Cpp_Subtype(cv(floating_point?)), t : Cpp_Type_): typ = volatile(t) \supset Cpp_Type?(t) \wedge cv(floating_point?)(t)
-----	---

Repeatedly Skolemizing and flattening,

Applying propositional simplification,

we get 2 subgoals:

dt_cv_float_TCC8.1:

{-1}	Cpp_Type?(typ')
{-2}	cv(floating_point?)(typ')
{-3}	typ' = volatile(t')
{1}	Cpp_Type?(t')

Expanding the definition of Cpp_Type?,

Repeatedly Skolemizing and flattening,

Instantiating the top quantifier in -2 with the terms: ($t!2$),

Expanding the definition of subterm,

Replacing using formula -4,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of dt_cv_float_TCC8.1.

dt_cv_float_TCC8.2:

{-1}	Cpp_Type?(typ')
{-2}	cv(floating_point?)(typ')
{-3}	typ' = volatile(t')
{1}	cv(floating_point?)(t')

Hiding formulas: -1,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of dt_cv_float_TCC8.2.

Q.E.D.

C.44.92 Cpp_Types.dt_cv_float_TCC9

Terse proof for dt_cv_float_TCC9.

dt_cv_float_TCC9:

$\{1\} \quad \forall (\text{typ}: \text{Cpp_Subtype}(\text{cv}(\text{floating_point?})), t: \text{Cpp_Type_}):$ $\text{typ} = \text{volatile}(t) \supset$ restrict $\quad [[\text{Cpp_Type_}, \text{Cpp_Type_}],$ $\quad [\text{Cpp_Subtype}(\text{cv}(\text{floating_point?})), \text{Cpp_Subtype}(\text{cv}(\text{floating_point?}))], \text{boolean}]$ $(\ll)(t, \text{typ})$

Repeatedly Skolemizing and flattening,
 Expanding the definition of restrict,
 Expanding the definition of ■,
 Replacing using formula -3,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 This completes the proof of dt_cv_float_TCC9.
 Q.E.D.

C.44.93 Cpp_Types.dt_cv_float_TCC10

Terse proof for dt_cv_float_TCC10.

dt_cv_float_TCC10:

$\{1\} \quad \forall (\text{typ}: \text{Cpp_Subtype}(\text{cv}(\text{floating_point?})), t: \text{Cpp_Type_}):$ $\text{typ} = \text{volatile}(t) \supset \text{Cpp_Type?}(t)$
--

Repeatedly Skolemizing and flattening,
 Using lemma dt_cv_float_TCC8,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 This completes the proof of dt_cv_float_TCC10.
 Q.E.D.

C.44.94 Cpp_Types.dt_cv_float_TCC11

Terse proof for dt_cv_float_TCC11.

dt_cv_float_TCC11:

$\{1\} \quad \forall (\text{typ}: \text{Cpp_Subtype}(\text{cv}(\text{floating_point?})), t: \text{Cpp_Type_}):$ $\text{typ} = \text{volatile}(t) \supset \text{floating_point?}(\text{cv_base}(t))$

Repeatedly Skolemizing and flattening,
 Rewriting using cv_base_result, matching in *,
 we get 2 subgoals:

dt_cv_float_TCC11.1:

$\{-1\} \quad \text{Cpp_Type?}(\text{typ}')$ $\{-2\} \quad \text{cv}(\text{floating_point?})(\text{typ}')$ $\{-3\} \quad \text{typ}' = \text{volatile}(t')$
$\{1\} \quad \text{cv}(\text{floating_point?})(t')$ $\{2\} \quad \text{floating_point?}(\text{cv_base}(t'))$

Using lemma dt_cv_float_TCC8,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 This completes the proof of dt_cv_float_TCC11.1.

dt_cv_float_TCC11.2:

{-1}	Cpp_Type?(typ')
{-2}	cv(floating_point?)(typ')
{-3}	typ' = volatile(t')
{1}	(floating_point? \subseteq interpreted?)
{2}	floating_point?(cv_base(t'))

Keeping (1) and hiding *,
 Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of dt_cv_float_TCC11.2.
 Q.E.D.

C.44.95 Cpp_Types.dt_cv_float_TCC12

Terse proof for dt_cv_float_TCC12.

dt_cv_float_TCC12:

{1}	\forall (typ: Cpp_Subtype(cv(floating_point?)), t: Cpp_Type_): typ = volatile(t) \supset (\exists (x ₁ : (interpreted_data_type?[(range_floating_point(cv_base(t)))])): TRUE)
-----	---

Repeatedly Skolemizing and flattening,
 Instantiating the top quantifier in 1 with the terms: (dt_floating_point(cv_base(t!1))),
 we get 3 subgoals:

dt_cv_float_TCC12.1:

{-1}	Cpp_Type?(typ')
{-2}	cv(floating_point?)(typ')
{-3}	typ' = volatile(t')
{1}	TRUE

which is trivially true.

This completes the proof of dt_cv_float_TCC12.1.

dt_cv_float_TCC12.2:

{-1}	Cpp_Type?(typ')
{-2}	cv(floating_point?)(typ')
{-3}	typ' = volatile(t')
{1}	floating_point?(cv_base(t'))

Using lemma dt_cv_float_TCC11,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 This completes the proof of dt_cv_float_TCC12.2.

dt_cv_float_TCC12.3:

{-1}	Cpp_Type?(typ')
{-2}	cv(floating_point?)(typ')
{-3}	typ' = volatile(t')
{1}	Cpp_Type?(t')

Using lemma dt_cv_float_TCC8,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 This completes the proof of dt_cv_float_TCC12.3.
 Q.E.D.

C.44.96 Cpp_Types.dt_cv_float_TCC13

Terse proof for dt_cv_float_TCC13.

dt_cv_float_TCC13:

{1}	$\begin{aligned} & \forall (\text{typ}: \text{Cpp_Subtype}(\text{cv}(\text{floating_point?})), \\ & \quad v: \\ & \quad [\text{typ1}: \\ & \quad \quad \{z: \text{Cpp_Subtype}(\text{cv}(\text{floating_point?})) \mid \\ & \quad \quad \quad \text{restrict} \\ & \quad \quad \quad [[\text{Cpp_Type_}, \text{Cpp_Type_}], \\ & \quad \quad \quad [\text{Cpp_Subtype}(\text{cv}(\text{floating_point?})), \\ & \quad \quad \quad \text{Cpp_Subtype}(\text{cv}(\text{floating_point?}))], \\ & \quad \quad \quad \text{boolean}] \\ & \quad \quad \quad (\ll)(z, \text{typ})\} \rightarrow \\ & \quad \quad \quad (\text{interpreted_data_type?}[[(\text{range_floating_point}(\text{cv_base}(\text{typ1})))]]), \\ & \quad t: \text{Cpp_Type_}: \\ & \quad \text{typ} = \text{volatile}(t) \supset \\ & \quad (\forall (x: \text{extended_real}): \\ & \quad \quad \text{range_floating_point}(\text{cv_base}(t))(x) \equiv \\ & \quad \quad \text{range_floating_point}(\text{cv_base}(\text{typ}))(x)) \\ & \quad \wedge \\ & \quad (\forall (x: \text{extended_real}): \\ & \quad \quad \text{range_floating_point}(\text{cv_base}(t))(x) \equiv \\ & \quad \quad \text{range_floating_point}(\text{cv_base}(\text{typ}))(x)) \\ & \quad \wedge \\ & \quad (\forall (x_1: [\text{list}[\text{Byte}], \text{Address}]): \\ & \quad \quad \text{every}[[\text{range_floating_point}(\text{cv_base}(t))]] \\ & \quad \quad \quad (\text{range_floating_point}(\text{cv_base}(\text{typ}))) \\ & \quad \quad \quad (\text{dt_volatile}[[\text{range_floating_point}(\text{cv_base}(t))]](v(t)) \text{ 'from_byte} \\ & \quad \quad \quad (x_1))) \\ & \quad \wedge \\ & \quad \text{interpreted_data_type?}[[\text{range_floating_point}(\text{cv_base}(\text{typ}))]] \\ & \quad \quad (\text{dt_volatile}[[\text{range_floating_point}(\text{cv_base}(t))]](v(t))) \end{aligned}$
-----	--

Repeatedly Skolemizing and flattening,

Expanding the definition of cv_base,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 3 subgoals:

dt_cv_float_TCC13.1:

{-1}	Cpp_Type?(typ')
{-2}	cv(floating_point?)(typ')
{-3}	typ' = volatile(t')
{1}	$\text{interpreted_data_type?}[[\text{range_floating_point}(\text{cv_base}(\text{typ}'))]] \\ (\text{dt_volatile}[[\text{range_floating_point}(\text{cv_base}(t'))]](v'(t')))$

Adding type constraints for dt_volatile[[range_floating_point(cv_base(t!1))]](v!1(t!1)),

we get 3 subgoals:

dt_cv_float_TCC13.1.1.1:

{-1} interpreted_data_type?[((range_floating_point(cv_base(t')))] (dt_volatile[((range_floating_point(cv_base(t')))](v'(t)))	
{-2} Cpp_Type?(typ')	
{-3} cv(floating_point?)(typ')	
{-4} typ' = volatile(t')	
{1} interpreted_data_type?[((range_floating_point(cv_base(typ')))] (dt_volatile[((range_floating_point(cv_base(t')))](v'(t)))	

Expanding the definition of interpreted_data_type?,

Repeatedly Skolemizing and flattening,

Applying propositional simplification,

we get 2 subgoals:

dt_cv_float_TCC13.1.1.1.1:

{-1} uninterpreted_data_type?(dt_volatile[((range_floating_point(cv_base(t')))] (v'(t))' uidt)	
{-2} $\forall (d: (((range_floating_point(cv_base(t')))), a: Address):$ valid?(uidt(dt_volatile[((range_floating_point(cv_base(t')))](v'(t))) (to_byte(dt_volatile[((range_floating_point(cv_base(t')))](v'(t))) (d, a), a)	
{-3} $\forall (l: list[Byte], a: Address):$ valid?(uidt(dt_volatile[((range_floating_point(cv_base(t')))](v'(t))) (l, a) \equiv up?(from_byte(dt_volatile[((range_floating_point(cv_base(t')))](v'(t))) (l, a))	
{-4} $\forall (d: (((range_floating_point(cv_base(t')))), a: Address):$ down(from_byte(dt_volatile[((range_floating_point(cv_base(t')))](v'(t))) (to_byte(dt_volatile[((range_floating_point(cv_base(t')))](v'(t))) (d, a), a)) $= d$	
{-5} Cpp_Type?(typ')	
{-6} cv(floating_point?)(typ')	
{-7} typ' = volatile(t')	
{1} $\forall (d: ((range_floating_point(cv_base(typ')))), a: Address):$ valid?(uidt(dt_volatile[((range_floating_point(cv_base(t')))](v'(t))) (to_byte(dt_volatile[((range_floating_point(cv_base(t')))](v'(t))) (d, a), a)	

Repeatedly Skolemizing and flattening,

Instantiating quantified variables,

Replacing using formula -8,

Expanding the definition of cv_base,

which is trivially true.

This completes the proof of dt_cv_float_TCC13.1.1.1.

dt_cv_float_TCC13.1.1.2:

<p>{-1} uninterpreted_data_type?(dt_volatile[(((range_floating_point(cv_base(t')))) (v'(t'))'uidt))</p> <p>{-2} $\forall (d: (((range_floating_point(cv_base(t'))))), a: Address):$ $valid?(uidt(dt_volatile[(((range_floating_point(cv_base(t'))))](v'(t'))))$ $(to_byte(dt_volatile[(((range_floating_point(cv_base(t'))))](v'(t'))))$ $(d, a),$ $a)$</p> <p>{-3} $\forall (l: list[Byte], a: Address):$ $valid?(uidt(dt_volatile[(((range_floating_point(cv_base(t'))))](v'(t'))))$ (l, a) \equiv $up?(from_byte(dt_volatile[(((range_floating_point(cv_base(t'))))](v'(t'))))$ (l, a)</p> <p>{-4} $\forall (d: (((range_floating_point(cv_base(t'))))), a: Address):$ $down(from_byte(dt_volatile[(((range_floating_point(cv_base(t'))))](v'(t'))))$ $(to_byte(dt_volatile[(((range_floating_point(cv_base(t'))))](v'(t'))))$ $(d, a),$ $a))$ $= d$</p> <p>{-5} Cpp_Type?(typ')</p> <p>{-6} cv(floating_point?)(typ')</p> <p>{-7} typ' = volatile(t')</p>	<hr style="border: 0.5px solid black;"/> <p>{1} $\forall (d: (((range_floating_point(cv_base(typ'))))), a: Address):$ $down(from_byte(dt_volatile[(((range_floating_point(cv_base(t'))))](v'(t'))))$ $(to_byte(dt_volatile[(((range_floating_point(cv_base(t'))))](v'(t'))))$ $(d, a),$ $a))$ $= d$</p>
--	---

Repeatedly Skolemizing and flattening,

Instantiating quantified variables,

Replacing using formula -8,

Expanding the definition of cv_base,

which is trivially true.

This completes the proof of dt_cv_float_TCC13.1.1.2.

dt_cv_float_TCC13.1.2:

<p>{-1} Cpp_Type?(typ')</p> <p>{-2} cv(floating_point?)(typ')</p> <p>{-3} typ' = volatile(t')</p>	<hr style="border: 0.5px solid black;"/> <p>{1} $\exists (x_1: (interpreted_data_type?(((range_floating_point(cv_base(t'))))))): TRUE$</p> <p>{2} $interpreted_data_type?(((range_floating_point(cv_base(typ')))))$ $(dt_volatile[(((range_floating_point(cv_base(t'))))](v'(t'))))$</p>
---	--

Using lemma dt_cv_float_TCC12,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of dt_cv_float_TCC13.1.2.

dt_cv_float_TCC13.1.3:

{-1}	Cpp_Type?(typ')
{-2}	cv(floating_point?)(typ')
{-3}	typ' = volatile(t')
{1} floating_point?(cv_base(t'))	
{2}	interpreted_data_type?[((range_floating_point(cv_base(typ'))))] (dt_volatile[((range_floating_point(cv_base(t')))](v'(t'))))

Using lemma dt_cv_float_TCC11,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of dt_cv_float_TCC13.1.3.

dt_cv_float_TCC13.2:

{-1}	Cpp_Type?(typ')
{-2}	cv(floating_point?)(typ')
{-3}	typ' = volatile(t')
{1} $\forall (x_1: \text{list}[\text{Byte}, \text{Address}]):$ every[((range_floating_point(cv_base(t')))] (range_floating_point(cv_base(typ')))] (dt_volatile[((range_floating_point(cv_base(t')))](v'(t')))' from_byte (x ₁))	

Repeatedly Skolemizing and flattening,

Expanding the definition of every,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Expanding the definition of cv_base,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Replacing using formula -3,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of dt_cv_float_TCC13.2.

dt_cv_float_TCC13.3:

{-1}	Cpp_Type?(typ')
{-2}	cv(floating_point?)(typ')
{-3}	typ' = volatile(t')
{1} $\forall (x: \text{extended_real}):$ range_floating_point(cv_base(t'))(x) \equiv range_floating_point(cv_base(typ(typ')))(x)	

Replacing using formula -3,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of dt_cv_float_TCC13.3.

Q.E.D.

C.44.97 Cpp_Types.dt_cv_float_TCC14

Terse proof for dt_cv_float_TCC14.

dt_cv_float_TCC14:

{1}	$\forall (\text{typ}: \text{Cpp_Subtype}(\text{cv}(\text{floating_point?}))):$ $\neg \text{volatile?}(\text{typ}) \wedge \neg \text{const?}(\text{typ}) \supset \text{floating_point?}(\text{typ})$
-----	---

Repeatedly Skolemizing and flattening,

Hiding formulas: -1,

Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of `dt_cv_float_TCC14`.
 Q.E.D.

C.44.98 Cpp_Types.dt_cv_float_TCC15

Terse proof for `dt_cv_float_TCC15`.

`dt_cv_float_TCC15`:

<pre> {1} ∀ (typ: Cpp_Subtype(cv(floating_point?))): ¬ volatile?(typ) ∧ ¬ const?(typ) ⊃ (∀ (x: extended_real): range_floating_point(typ)(x) ≡ range_floating_point(cv_base(typ))(x)) ∧ (∀ (x: extended_real): range_floating_point(typ)(x) ≡ range_floating_point(cv_base(typ))(x)) ∧ (∀ (x₁: [list[Byte], Address]): every[[(range_floating_point(typ))] ((range_floating_point(cv_base(typ)))) (dt_floating_point(typ) 'from_byte(x₁))] ∧ interpreted_data_type?[(range_floating_point(cv_base(typ)))] (dt_floating_point(typ)) </pre>

Repeatedly Skolemizing and flattening,
 Using lemma `cv_base_result2`,
 Simplifying, rewriting, and recording with decision procedures,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 we get 3 subgoals:

`dt_cv_float_TCC15.1`:

<pre> {-1} cv_base(typ') = typ' {-2} Cpp_Type?(typ') {-3} cv(floating_point?)(typ') </pre> <hr/> <pre> {1} volatile?(typ') {2} const?(typ') {3} interpreted_data_type?[(range_floating_point(cv_base(typ')))] (dt_floating_point(typ')) </pre>

Replacing using formula -1,
 Adding type constraints for `dt_floating_point(cv_base(typ!1))`,
 Rewriting using `pod_is_interpreted_data`, matching in *,
 This completes the proof of `dt_cv_float_TCC15.1`.

`dt_cv_float_TCC15.2`:

<pre> {-1} cv_base(typ') = typ' {-2} Cpp_Type?(typ') {-3} cv(floating_point?)(typ') </pre> <hr/> <pre> {1} volatile?(typ') {2} const?(typ') {3} ∀ (x₁: [list[Byte], Address]): every[[(range_floating_point(typ'))] ((range_floating_point(cv_base(typ')))] (dt_floating_point(typ') 'from_byte(x₁))] </pre>

Repeatedly Skolemizing and flattening,
 Expanding the definition of every,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 This completes the proof of dt_cv_float_TCC15.2.

dt_cv_float_TCC15.3:

{-1}	cv_base(typ') = typ'
{-2}	Cpp_Type?(typ')
{-3}	cv(floating_point?)(typ')
{1}	volatile?(typ')
{2}	const?(typ')
{3}	$\forall (x: \text{extended_real}):$ range_floating_point(typ')(x) \equiv range_floating_point(cv_base(typ'))(x)

Repeatedly Skolemizing and flattening,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 This completes the proof of dt_cv_float_TCC15.3.
 Q.E.D.

C.44.99 Cpp_Types.uidt_TCC1

Terse proof for uidt_TCC1.

uidt_TCC1:

{1}	$\forall (\text{typ}: \text{Cpp_Type}, r: \text{Cpp_Type_}, a: \text{list}[\text{Cpp_Type_}], v: \text{bool}):$ typ = function_type(r, a, v) \supset function?(typ)
-----	---

Repeatedly Skolemizing and flattening,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 This completes the proof of uidt_TCC1.
 Q.E.D.

C.44.100 Cpp_Types.uidt_TCC2

Terse proof for uidt_TCC2.

uidt_TCC2:

{1}	$\forall (\text{typ}: \text{Cpp_Type}, t: \text{Cpp_Type_}):$ typ = pointer(t) \supset cv(pointer?)(typ)
-----	---

Repeatedly Skolemizing and flattening,
 Replacing using formula -2,
 Hiding formulas: -1,
 Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of uidt_TCC2.
 Q.E.D.

C.44.101 Cpp_Types.uidt_TCC3

Terse proof for uidt_TCC3.

uidt_TCC3:

{1}	$\forall (\text{typ}: \text{Cpp_Type}, t: \text{Cpp_Type_}):$ typ = pointer(t) \supset uninterpreted_data_type?(uidt(dt_cv_pointer(typ)))
-----	---

Repeatedly Skolemizing and flattening,
 Adding type constraints for `dt_cv_pointer(ty!1)`,
 Expanding the definition of `interpreted_data_type?`,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 This completes the proof of `uidt_TCC3`.
 Q.E.D.

C.44.102 Cpp_Types.uidt_TCC4

Terse proof for `uidt_TCC4`.

`uidt_TCC4`:

{1}	$\forall (\text{typ}: \text{Cpp_Type}, t: \text{Cpp_Type_}):$ $\text{typ} = \text{reference}(t) \supset \text{uninterpreted_data_type?}(\text{uidt}(\text{dt_reference}(\text{typ})))$
-----	---

Repeatedly Skolemizing and flattening,
 Adding type constraints for `dt_reference(ty!1)`,
 Expanding the definition of `pod_data_type?`,
 Expanding the definition of `interpreted_data_type?`,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 This completes the proof of `uidt_TCC4`.
 Q.E.D.

C.44.103 Cpp_Types.uidt_TCC5

Terse proof for `uidt_TCC5`.

`uidt_TCC5`:

{1}	$\forall (\text{typ}: \text{Cpp_Type}, c: (\text{class?}), t: \text{Cpp_Type_}):$ $\text{typ} = \text{pointer_to_member}(c, t) \supset \text{cv}(\text{pointer_to_member?})(\text{typ})$
-----	--

Repeatedly Skolemizing and flattening,
 Replacing using formula -3,
 Hiding formulas: -1,
 Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of `uidt_TCC5`.
 Q.E.D.

C.44.104 Cpp_Types.uidt_TCC6

Terse proof for `uidt_TCC6`.

`uidt_TCC6`:

{1}	$\forall (\text{typ}: \text{Cpp_Type}, c: (\text{class?}), t: \text{Cpp_Type_}):$ $\text{typ} = \text{pointer_to_member}(c, t) \supset \text{uninterpreted_data_type?}(\text{uidt}(\text{dt_cv_ptm}(\text{typ})))$
-----	--

Repeatedly Skolemizing and flattening,
 Replacing using formula -3,
 Adding type constraints for `dt_cv_ptm(pointer_to_member(c!1, t!1))`,
 Expanding the definition of `interpreted_data_type?`,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 This completes the proof of `uidt_TCC6`.

Q.E.D.

C.44.105 Cpp_Types.uidt_TCC7

Terse proof for uidt_TCC7.

uidt_TCC7:

$$\{1\} \quad \forall (\text{typ: Cpp_Type}, t: \text{Cpp_Type_}, \text{bo: nat}, s: \text{posnat}):$$

$$\text{typ} = \text{bitfield}(t, \text{bo}, s) \supset \text{cv}(\text{bitfield?})(\text{typ})$$

Repeatedly Skolemizing and flattening,
 Replacing using formula -4,
 Hiding formulas: -1,
 Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of uidt_TCC7.
 Q.E.D.

C.44.106 Cpp_Types.uidt_TCC8

Terse proof for uidt_TCC8.

uidt_TCC8:

$$\{1\} \quad \forall (\text{typ: Cpp_Type}, t: \text{Cpp_Type_}, \text{bo: nat}, s: \text{posnat}):$$

$$\text{typ} = \text{bitfield}(t, \text{bo}, s) \supset \text{uninterpreted_data_type?}(\text{uidt}(\text{dt_cv_bitfield}(\text{typ})))$$

Repeatedly Skolemizing and flattening,
 Adding type constraints for dt_cv_bitfield(typ!1),
 Expanding the definition of interpreted_data_type?,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 This completes the proof of uidt_TCC8.
 Q.E.D.

C.44.107 Cpp_Types.uidt_TCC9

Terse proof for uidt_TCC9.

uidt_TCC9:

$$\{1\} \quad \forall (\text{typ: Cpp_Type}): \text{typ} = \text{bool} \supset \text{uninterpreted_data_type?}(\text{uidt}(\text{dt_bool}))$$

Repeatedly Skolemizing and flattening,
 Adding type constraints for dt_bool,
 Expanding the definition of pod_data_type?,
 Expanding the definition of interpreted_data_type?,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 This completes the proof of uidt_TCC9.
 Q.E.D.

C.44.108 Cpp_Types.uidt_TCC10

Terse proof for uidt_TCC10.

uidt_TCC10:

$$\{1\} \quad \forall (\text{typ}: \text{Cpp_Type}, t: \text{Cpp_Type_}):$$

$$t = \text{bool} \wedge \text{typ} = \text{const}(t) \supset \text{uninterpreted_data_type?}(\text{uidt}(\text{dt_bool}))$$

Repeatedly Skolemizing and flattening,
 Adding type constraints for dt_bool,
 Expanding the definition of pod_data_type?,
 Expanding the definition of interpreted_data_type?,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 This completes the proof of uidt_TCC10.
 Q.E.D.

C.44.109 Cpp_Types.uidt_TCC11

Terse proof for uidt_TCC11.

uidt_TCC11:

$$\{1\} \quad \forall (\text{typ}: \text{Cpp_Type}, t: \text{Cpp_Type_}):$$

$$\text{typ} = \text{const}(t) \supset (\forall (t_1: \text{Cpp_Type_}): t = \text{pointer}(t_1) \supset \text{cv}(\text{pointer?})(\text{typ}))$$

Repeatedly Skolemizing and flattening,
 Hiding formulas: -1,
 Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of uidt_TCC11.
 Q.E.D.

C.44.110 Cpp_Types.uidt_TCC12

Terse proof for uidt_TCC12.

uidt_TCC12:

$$\{1\} \quad \forall (\text{typ}: \text{Cpp_Type}, t: \text{Cpp_Type_}):$$

$$\text{typ} = \text{const}(t) \supset$$

$$(\forall (t_1: \text{Cpp_Type_}):$$

$$t = \text{pointer}(t_1) \supset \text{uninterpreted_data_type?}(\text{uidt}(\text{dt_cv_pointer}(\text{typ}))))$$

Repeatedly Skolemizing and flattening,
 Adding type constraints for dt_cv_pointer(typ!1),
 Expanding the definition of interpreted_data_type?,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 This completes the proof of uidt_TCC12.
 Q.E.D.

C.44.111 Cpp_Types.uidt_TCC13

Terse proof for uidt_TCC13.

uidt_TCC13:

$$\{1\} \quad \forall (\text{typ}: \text{Cpp_Type}, t: \text{Cpp_Type_}):$$

$$\text{typ} = \text{const}(t) \supset$$

$$(\forall (c: (\text{class?}), t_1: \text{Cpp_Type_}):$$

$$t = \text{pointer_to_member}(c, t_1) \supset \text{cv}(\text{pointer_to_member?})(\text{typ}))$$

Repeatedly Skolemizing and flattening,
Hiding formulas: -2,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of uidt_TCC13.
Q.E.D.

C.44.112 Cpp_Types.uidt_TCC14

Terse proof for uidt_TCC14.

uidt_TCC14:

$$\{1\} \quad \forall (\text{typ}: \text{Cpp_Type}, t: \text{Cpp_Type_}):$$

$$\text{typ} = \text{const}(t) \supset$$

$$(\forall (c: (\text{class?}), t_1: \text{Cpp_Type_}):$$

$$t = \text{pointer_to_member}(c, t_1) \supset \text{uninterpreted_data_type?}(\text{uidt}(\text{dt_cv_ptm}(\text{typ}))))$$

Repeatedly Skolemizing and flattening,
Adding type constraints for dt_cv_ptm(typ!1),
Expanding the definition of interpreted_data_type?,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of uidt_TCC14.
Q.E.D.

C.44.113 Cpp_Types.uidt_TCC15

Terse proof for uidt_TCC15.

uidt_TCC15:

$$\{1\} \quad \forall (\text{typ}: \text{Cpp_Type}, t: \text{Cpp_Type_}):$$

$$\text{typ} = \text{const}(t) \supset$$

$$(\forall (t_1: \text{Cpp_Type_}, \text{bo}: \text{nat}, s: \text{posnat}):$$

$$t = \text{bitfield}(t_1, \text{bo}, s) \supset \text{cv}(\text{bitfield?})(\text{typ}))$$

Repeatedly Skolemizing and flattening,
Hiding formulas: -3,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of uidt_TCC15.
Q.E.D.

C.44.114 Cpp_Types.uidt_TCC16

Terse proof for uidt_TCC16.

uidt_TCC16:

```
{1}  ∀ (typ: Cpp_Type, t: Cpp_Type_):
      typ = const(t) ⊃
      (∀ (t1: Cpp_Type_, bo: nat, s: posnat):
        t = bitfield(t1, bo, s) ⊃ uninterpreted_data_type?(uidt(dt_cv_bitfield(typ))))
```

Repeatedly Skolemizing and flattening,
 Adding type constraints for dt_cv_bitfield(typ!1),
 Expanding the definition of interpreted_data_type?,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 This completes the proof of uidt_TCC16.
 Q.E.D.

C.44.115 Cpp_Types.uidt_TCC17

Terse proof for uidt_TCC17.

uidt_TCC17:

```
{1}  ∀ (typ: Cpp_Type, t: Cpp_Type_):
      floating_point?(t) ∧
      ¬ volatile?(t) ∧
      ¬ bitfield?(t) ∧
      ¬ pointer_to_member?(t) ∧ ¬ pointer?(t) ∧ ¬ bool?(t) ∧ typ = const(t)
      ⊃ cv(floating_point?)(typ)
```

Repeatedly Skolemizing and flattening,
 Hiding formulas: -1,
 Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of uidt_TCC17.
 Q.E.D.

C.44.116 Cpp_Types.uidt_TCC18

Terse proof for uidt_TCC18.

uidt_TCC18:

```
{1}  ∀ (typ: Cpp_Type, t: Cpp_Type_):
      floating_point?(t) ∧
      ¬ volatile?(t) ∧
      ¬ bitfield?(t) ∧
      ¬ pointer_to_member?(t) ∧ ¬ pointer?(t) ∧ ¬ bool?(t) ∧ typ = const(t)
      ⊃ uninterpreted_data_type?(uidt(dt_cv_float(typ)))
```

Repeatedly Skolemizing and flattening,
 Adding type constraints for dt_cv_float(typ!1),
 Expanding the definition of interpreted_data_type?,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 This completes the proof of uidt_TCC18.
 Q.E.D.

C.44.117 Cpp_Types.uidt_TCC19

Terse proof for uidt_TCC19.

uidt_TCC19:

```
{1}  ∀ (typ: Cpp_Type, t: Cpp_Type-):
      non_bool_integral_enum?(t) ∧
      ¬ volatile?(t) ∧
      ¬ bitfield?(t) ∧
      ¬ pointer_to_member?(t) ∧ ¬ pointer?(t) ∧ ¬ bool?(t) ∧ typ = const(t)
      ⊃ cv(non_bool_integral_enum?)(typ)
```

Repeatedly Skolemizing and flattening,

Hiding formulas: -1,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of uidt_TCC19.

Q.E.D.

C.44.118 Cpp_Types.uidt_TCC20

Terse proof for uidt_TCC20.

uidt_TCC20:

```
{1}  ∀ (typ: Cpp_Type, t: Cpp_Type-):
      non_bool_integral_enum?(t) ∧
      ¬ volatile?(t) ∧
      ¬ bitfield?(t) ∧
      ¬ pointer_to_member?(t) ∧ ¬ pointer?(t) ∧ ¬ bool?(t) ∧ typ = const(t)
      ⊃ uninterpreted_data_type?(uidt(dt(typ)))
```

Repeatedly Skolemizing and flattening,

Adding type constraints for dt(typ!1),

Expanding the definition of pod_data_type?,

Expanding the definition of interpreted_data_type?,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of uidt_TCC20.

Q.E.D.

C.44.119 Cpp_Types.uidt_TCC21

Terse proof for uidt_TCC21.

uidt_TCC21:

```
{1}  ∀ (typ: Cpp_Type, t: Cpp_Type-):
      ¬ volatile?(t) ∧
      ¬ bitfield?(t) ∧
      ¬ pointer_to_member?(t) ∧ ¬ pointer?(t) ∧ ¬ bool?(t) ∧ typ = const(t)
      ⊃ ¬ (floating_point?(t) ∧ non_bool_integral_enum?(t))
```

Repeatedly Skolemizing and flattening,

Hiding formulas: -1,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of uidt_TCC21.

Q.E.D.

C.44.120 Cpp_Types.uidt_TCC22

Terse proof for uidt_TCC22.

uidt_TCC22:

$$\{1\} \quad \forall (\text{typ}: \text{Cpp_Type}, t: \text{Cpp_Type_}):$$

$$\quad \neg \text{volatile?}(t) \wedge$$

$$\quad \neg \text{bitfield?}(t) \wedge$$

$$\quad \neg \text{pointer_to_member?}(t) \wedge \neg \text{pointer?}(t) \wedge \neg \text{bool?}(t) \wedge \text{typ} = \text{const}(t)$$

$$\quad \supset \text{floating_point?}(t) \vee \text{non_bool_Integral_enum?}(t)$$

Repeatedly Skolemizing and flattening,
 Expanding the definition of Cpp_Type?,
 Instantiating the top quantifier in -1 with the terms: (typ!1),
 Expanding the definition of subterm,
 Applying disjunctive simplification to flatten sequent,
 Keeping (-3 -17 -21 -23 -24 -25 -27 -29 1 2 3 4 5 6 7) and hiding *,
 Replacing using formula -8,
 Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of uidt_TCC22.

Q.E.D.

C.44.121 Cpp_Types.uidt_TCC23

Terse proof for uidt_TCC23.

uidt_TCC23:

$$\{1\} \quad \forall (\text{typ}: \text{Cpp_Type}, t: \text{Cpp_Type_}):$$

$$\quad t = \text{bool} \wedge \text{typ} = \text{volatile}(t) \supset \text{uninterpreted_data_type?}(\text{uidt}(\text{dt_bool}))$$

Repeatedly Skolemizing and flattening,
 Adding type constraints for dt_bool,
 Expanding the definition of pod_data_type?,
 Expanding the definition of interpreted_data_type?,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 This completes the proof of uidt_TCC23.

Q.E.D.

C.44.122 Cpp_Types.uidt_TCC24

Terse proof for uidt_TCC24.

uidt_TCC24:

$$\{1\} \quad \forall (\text{typ}: \text{Cpp_Type}, t: \text{Cpp_Type_}):$$

$$\quad \text{typ} = \text{volatile}(t) \supset (\forall (t_1: \text{Cpp_Type_}): t = \text{pointer}(t_1) \supset \text{cv}(\text{pointer?})(\text{typ}))$$

Repeatedly Skolemizing and flattening,
 Hiding formulas: -1,
 Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of uidt_TCC24.

Q.E.D.

C.44.123 Cpp_Types.uidt_TCC25

Terse proof for uidt_TCC25.

uidt_TCC25:

{1}	$\forall (\text{typ}: \text{Cpp_Type}, t: \text{Cpp_Type_}):$ $\text{typ} = \text{volatile}(t) \supset$ $(\forall (t_1: \text{Cpp_Type_}):$ $t = \text{pointer}(t_1) \supset \text{uninterpreted_data_type?}(\text{uidt}(\text{dt_cv_pointer}(\text{typ}))))$
-----	--

Repeatedly Skolemizing and flattening,
 Adding type constraints for dt_cv_pointer(typ!1),
 Expanding the definition of interpreted_data_type?,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 This completes the proof of uidt_TCC25.

Q.E.D.

C.44.124 Cpp_Types.uidt_TCC26

Terse proof for uidt_TCC26.

uidt_TCC26:

{1}	$\forall (\text{typ}: \text{Cpp_Type}, t: \text{Cpp_Type_}):$ $\text{typ} = \text{volatile}(t) \supset$ $(\forall (c: (\text{class?}), t_1: \text{Cpp_Type_}):$ $t = \text{pointer_to_member}(c, t_1) \supset \text{cv}(\text{pointer_to_member?})(\text{typ}))$
-----	---

Repeatedly Skolemizing and flattening,
 Hiding formulas: -2,
 Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of uidt_TCC26.

Q.E.D.

C.44.125 Cpp_Types.uidt_TCC27

Terse proof for uidt_TCC27.

uidt_TCC27:

{1}	$\forall (\text{typ}: \text{Cpp_Type}, t: \text{Cpp_Type_}):$ $\text{typ} = \text{volatile}(t) \supset$ $(\forall (c: (\text{class?}), t_1: \text{Cpp_Type_}):$ $t = \text{pointer_to_member}(c, t_1) \supset \text{uninterpreted_data_type?}(\text{uidt}(\text{dt_cv_ptm}(\text{typ}))))$
-----	---

Repeatedly Skolemizing and flattening,
 Adding type constraints for dt_cv_ptm(typ!1),
 Expanding the definition of interpreted_data_type?,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 This completes the proof of uidt_TCC27.

Q.E.D.

C.44.126 Cpp_Types.uidt_TCC28

Terse proof for uidt_TCC28.

uidt_TCC28:

$$\{1\} \quad \forall (\text{typ}: \text{Cpp_Type}, t: \text{Cpp_Type_}):$$

$$\text{typ} = \text{volatile}(t) \supset$$

$$(\forall (t_1: \text{Cpp_Type_}, \text{bo}: \text{nat}, s: \text{posnat}):$$

$$t = \text{bitfield}(t_1, \text{bo}, s) \supset \text{cv}(\text{bitfield?})(\text{typ}))$$

Repeatedly Skolemizing and flattening,

Hiding formulas: -3,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of uidt_TCC28.

Q.E.D.

C.44.127 Cpp_Types.uidt_TCC29

Terse proof for uidt_TCC29.

uidt_TCC29:

$$\{1\} \quad \forall (\text{typ}: \text{Cpp_Type}, t: \text{Cpp_Type_}):$$

$$\text{typ} = \text{volatile}(t) \supset$$

$$(\forall (t_1: \text{Cpp_Type_}, \text{bo}: \text{nat}, s: \text{posnat}):$$

$$t = \text{bitfield}(t_1, \text{bo}, s) \supset \text{uninterpreted_data_type?}(\text{uidt}(\text{dt_cv_bitfield}(\text{typ}))))$$

Repeatedly Skolemizing and flattening,

Adding type constraints for dt_cv_bitfield(typ!1),

Expanding the definition of interpreted_data_type?,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of uidt_TCC29.

Q.E.D.

C.44.128 Cpp_Types.uidt_TCC30

Terse proof for uidt_TCC30.

uidt_TCC30:

$$\{1\} \quad \forall (\text{typ}: \text{Cpp_Type}, t: \text{Cpp_Type_}):$$

$$\text{floating_point?}(t) \wedge$$

$$\neg \text{bitfield?}(t) \wedge$$

$$\neg \text{pointer_to_member?}(t) \wedge \neg \text{pointer?}(t) \wedge \neg \text{bool?}(t) \wedge \text{typ} = \text{volatile}(t)$$

$$\supset \text{cv}(\text{floating_point?})(\text{typ})$$

Repeatedly Skolemizing and flattening,

Hiding formulas: -1,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of uidt_TCC30.

Q.E.D.

C.44.129 Cpp_Types.uidt_TCC31

Terse proof for uidt_TCC31.

uidt_TCC31:

```
{1}  ∀ (typ: Cpp_Type, t: Cpp_Type_):
      floating_point?(t) ∧
      ¬ bitfield?(t) ∧
      ¬ pointer_to_member?(t) ∧ ¬ pointer?(t) ∧ ¬ bool?(t) ∧ typ = volatile(t)
      ⊃ uninterpreted_data_type?(uidt(dt_cv_float(typ)))
```

Repeatedly Skolemizing and flattening,
 Adding type constraints for dt_cv_float(typ!1),
 Expanding the definition of interpreted_data_type?,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 This completes the proof of uidt_TCC31.
 Q.E.D.

C.44.130 Cpp_Types.uidt_TCC32

Terse proof for uidt_TCC32.

uidt_TCC32:

```
{1}  ∀ (typ: Cpp_Type, t: Cpp_Type_):
      non_bool_integral_enum?(t) ∧
      ¬ bitfield?(t) ∧
      ¬ pointer_to_member?(t) ∧ ¬ pointer?(t) ∧ ¬ bool?(t) ∧ typ = volatile(t)
      ⊃ cv(non_bool_integral_enum?)(typ)
```

Repeatedly Skolemizing and flattening,
 Hiding formulas: -1,
 Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of uidt_TCC32.
 Q.E.D.

C.44.131 Cpp_Types.uidt_TCC33

Terse proof for uidt_TCC33.

uidt_TCC33:

```
{1}  ∀ (typ: Cpp_Type, t: Cpp_Type_):
      non_bool_integral_enum?(t) ∧
      ¬ bitfield?(t) ∧
      ¬ pointer_to_member?(t) ∧ ¬ pointer?(t) ∧ ¬ bool?(t) ∧ typ = volatile(t)
      ⊃ uninterpreted_data_type?(uidt(dt(typ)))
```

Repeatedly Skolemizing and flattening,
 Adding type constraints for dt(typ!1),
 Expanding the definition of pod_data_type?,
 Expanding the definition of interpreted_data_type?,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 This completes the proof of uidt_TCC33.
 Q.E.D.

C.44.132 Cpp_Types.uidt_TCC34

Terse proof for uidt_TCC34.

uidt_TCC34:

$\{1\} \quad \forall (\text{typ: Cpp_Type}, t: \text{Cpp_Type_}):$ $\quad \neg \text{bitfield?}(t) \wedge$ $\quad \neg \text{pointer_to_member?}(t) \wedge \neg \text{pointer?}(t) \wedge \neg \text{bool?}(t) \wedge \text{typ} = \text{volatile}(t)$ $\quad \supset \neg (\text{floating_point?}(t) \wedge \text{non_bool_integral_enum?}(t))$

Repeatedly Skolemizing and flattening,

Keeping (-3 -4) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of uidt_TCC34.

Q.E.D.

C.44.133 Cpp_Types.uidt_TCC35

Terse proof for uidt_TCC35.

uidt_TCC35:

$\{1\} \quad \forall (\text{typ: Cpp_Type}, t: \text{Cpp_Type_}):$ $\quad \neg \text{bitfield?}(t) \wedge$ $\quad \neg \text{pointer_to_member?}(t) \wedge \neg \text{pointer?}(t) \wedge \neg \text{bool?}(t) \wedge \text{typ} = \text{volatile}(t)$ $\quad \supset \text{floating_point?}(t) \vee \text{non_bool_integral_enum?}(t)$
--

Repeatedly Skolemizing and flattening,

Expanding the definition of Cpp_Type?,

Instantiating the top quantifier in -1 with the terms: (typ!1),

Expanding the definition of subterm,

Applying disjunctive simplification to flatten sequent,

Replacing using formula -29,

Keeping (-3 -17 -20 -22 -23 -24 -25 -27 1 2 3 4 5 6) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of uidt_TCC35.

Q.E.D.

C.44.134 Cpp_Types.uidt_TCC36

Terse proof for uidt_TCC36.

uidt_TCC36:

```

{1}  ∀ (typ: Cpp_Type):
      floating_point?(typ) ∧
      ¬ volatile?(typ) ∧
      ¬ const?(typ) ∧
      ¬ bool?(typ) ∧
      ¬ bitfield?(typ) ∧
      ¬ pointer_to_member?(typ) ∧
      ¬ reference?(typ) ∧
      ¬ pointer?(typ) ∧
      ¬ union?(typ) ∧
      ¬ class?(typ) ∧ ¬ function?(typ) ∧ ¬ array?(typ) ∧ ¬ void?(typ)
    ⊃ cv(floating_point?)(typ)

```

Repeatedly Skolemizing and flattening,
 Keeping (-2 13) and hiding *,
 Expanding the definition of cv,
 Expanding the definition of c,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 This completes the proof of uidt_TCC36.
 Q.E.D.

C.44.135 Cpp_Types.uidt_TCC37

Terse proof for uidt_TCC37.

uidt_TCC37:

```

{1}  ∀ (typ: Cpp_Type):
      floating_point?(typ) ∧
      ¬ volatile?(typ) ∧
      ¬ const?(typ) ∧
      ¬ bool?(typ) ∧
      ¬ bitfield?(typ) ∧
      ¬ pointer_to_member?(typ) ∧
      ¬ reference?(typ) ∧
      ¬ pointer?(typ) ∧
      ¬ union?(typ) ∧
      ¬ class?(typ) ∧ ¬ function?(typ) ∧ ¬ array?(typ) ∧ ¬ void?(typ)
    ⊃ uninterpreted_data_type?(uidt(dt_cv_float(typ)))

```

Repeatedly Skolemizing and flattening,
 Adding type constraints for dt_cv_float(typ!1),
 Expanding the definition of interpreted_data_type?,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 This completes the proof of uidt_TCC37.
 Q.E.D.

C.44.136 Cpp_Types.uidt_TCC38

Terse proof for uidt_TCC38.

uidt_TCC38:

```
{1}  ∀ (typ: Cpp_Type):
      non_bool_integral_enum?(typ) ∧
      ¬ volatile?(typ) ∧
      ¬ const?(typ) ∧
      ¬ bool?(typ) ∧
      ¬ bitfield?(typ) ∧
      ¬ pointer_to_member?(typ) ∧
      ¬ reference?(typ) ∧
      ¬ pointer?(typ) ∧
      ¬ union?(typ) ∧
      ¬ class?(typ) ∧ ¬ function?(typ) ∧ ¬ array?(typ) ∧ ¬ void?(typ)
      ⊃ cv(non_bool_integral_enum?)(typ)
```

Repeatedly Skolemizing and flattening,
 Expanding the definition of cv,
 Expanding the definition of c,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 This completes the proof of uidt_TCC38.
 Q.E.D.

C.44.137 Cpp_Types.uidt_TCC39

Terse proof for uidt_TCC39.

uidt_TCC39:

```
{1}  ∀ (typ: Cpp_Type):
      non_bool_integral_enum?(typ) ∧
      ¬ volatile?(typ) ∧
      ¬ const?(typ) ∧
      ¬ bool?(typ) ∧
      ¬ bitfield?(typ) ∧
      ¬ pointer_to_member?(typ) ∧
      ¬ reference?(typ) ∧
      ¬ pointer?(typ) ∧
      ¬ union?(typ) ∧
      ¬ class?(typ) ∧ ¬ function?(typ) ∧ ¬ array?(typ) ∧ ¬ void?(typ)
      ⊃ uninterpreted_data_type?(uidt(dt(typ)))
```

Repeatedly Skolemizing and flattening,
 Adding type constraints for dt(typ!1),
 Expanding the definition of pod_data_type?,
 Expanding the definition of interpreted_data_type?,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 This completes the proof of uidt_TCC39.
 Q.E.D.

C.44.138 Cpp_Types.uidt_TCC40

Terse proof for uidt_TCC40.

uidt_TCC40:

```
{1}  ∀ (typ: Cpp_Type):
      ¬ volatile?(typ) ∧
      ¬ const?(typ) ∧
      ¬ bool?(typ) ∧
      ¬ bitfield?(typ) ∧
      ¬ pointer_to_member?(typ) ∧
      ¬ reference?(typ) ∧
      ¬ pointer?(typ) ∧
      ¬ union?(typ) ∧
      ¬ class?(typ) ∧ ¬ function?(typ) ∧ ¬ array?(typ) ∧ ¬ void?(typ)
      ⊃ ¬ (floating_point?(typ) ∧ non_bool_integral_enum?(typ))
```

Repeatedly Skolemizing and flattening,
 Keeping (-2 -3) and hiding *,
 Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of uidt_TCC40.
 Q.E.D.

C.44.139 Cpp_Types.uidt_TCC41

Terse proof for uidt_TCC41.

uidt_TCC41:

```
{1}  ∀ (typ: Cpp_Type):
      ¬ volatile?(typ) ∧
      ¬ const?(typ) ∧
      ¬ bool?(typ) ∧
      ¬ bitfield?(typ) ∧
      ¬ pointer_to_member?(typ) ∧
      ¬ reference?(typ) ∧
      ¬ pointer?(typ) ∧
      ¬ union?(typ) ∧
      ¬ class?(typ) ∧ ¬ function?(typ) ∧ ¬ array?(typ) ∧ ¬ void?(typ)
      ⊃ floating_point?(typ) ∨ non_bool_integral_enum?(typ)
```

Repeatedly Skolemizing and flattening,
 Hiding formulas: -1,
 Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of uidt_TCC41.
 Q.E.D.

C.44.140 Cpp_Types.mod_float_TCC1

Terse proof for mod_float_TCC1.

mod_float_TCC1:

```
{1}  ∃ (x: [Cpp_Subtype(cv(floating_point?)) → [extended_real → extended_real]]): TRUE
```

Instantiating the top quantifier in 1 with the terms: $(\lambda (t: \text{Cpp_Subtype}(cv(\text{floating_point?}))) : \lambda (e: \text{extended_real}): e)$,

which is trivially true.
 This completes the proof of `mod_float_TCC1`.
 Q.E.D.

C.44.141 Cpp_Types.value_bitmask_TCC1

Terse proof for `value_bitmask_TCC1`.

`value_bitmask_TCC1`:

```
{1}  ∃ (typ: Cpp_Subtype(cv(non_bool_integral_enum?))):
      ¬ (bool?(typ) ∨
         float?(typ) ∨
         double?(typ) ∨
         longdouble?(typ) ∨
         void?(typ) ∨
         array?(typ) ∨
         function?(typ) ∨
         pointer?(typ) ∨
         reference?(typ) ∨
         class?(typ) ∨ union?(typ) ∨ bitfield?(typ) ∨ pointer_to_member?(typ))
```

Repeatedly Skolemizing and flattening,
 Hiding formulas: -1,
 Expanding the definition of `cv`,
 Expanding the definition of `c`,
 Expanding the definition of `v`,
 Expanding the definition of `non_bool_integral_enum?`,
 Expanding the definition of `non_bool_integral?`,
 Expanding the definition of `unsigned_integer?`,
 Expanding the definition of `signed_integer?`,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 This completes the proof of `value_bitmask_TCC1`.
 Q.E.D.

C.45 Proofs for Cpp_Verification (cpp-verification.pvs)

This theory contains no provable formal statements.

C.46 Proofs for Cpp_array (types.pvs)

C.46.1 Cpp_array.uidt_array_TCC1

Terse proof for `uidt_array_TCC1`.

`uidt_array_TCC1`:

```
{1}  ∃ (x: [Cpp_Subtype(array?) → (uninterpreted_data_type?)]): TRUE
```

Instantiating the top quantifier in 1 with the terms: $(\lambda (c: \text{Cpp_Subtype(array?)})$:
`choose(uninterpreted_data_type?)`),
 we get 2 subgoals:

uidt_array_TCC1.1:

{1} TRUE

which is trivially true.

This completes the proof of uidt_array_TCC1.1.

uidt_array_TCC1.2:

{1} $\forall (c: \text{Cpp_Subtype}(\text{array?})): \text{nonempty?}[\text{Uninterpreted_data_type}](\text{uninterpreted_data_type?})$

Trying repeated skolemization, instantiation, and if-lifting,

Instantiating the top quantifier in -3 with the terms: ((# size := 0, valid? := LAMBDA (l: list[Byte],

a: Address): FALSE #)),

Repeatedly Skolemizing and flattening,

This completes the proof of uidt_array_TCC1.2.

Q.E.D.

C.47 Proofs for Cpp_bitfield (types.pvs)

C.47.1 Cpp_bitfield.range_bitfield_TCC1

Terse proof for range_bitfield_TCC1.

range_bitfield_TCC1:

{1} $\exists (x: [\text{Cpp_Subtype}(\text{bitfield?}) \rightarrow \text{non_empty_finite_set}[\text{int}]])$: TRUE

Instantiating the top quantifier in 1 with the terms: ($\lambda (b: \text{Cpp_Subtype}(\text{bitfield?})): \lambda (i: \text{int}): i = 0$),

we get 2 subgoals:

range_bitfield_TCC1.1:

{1} TRUE

which is trivially true.

This completes the proof of range_bitfield_TCC1.1.

range_bitfield_TCC1.2:

{1} $\forall (b: \text{Cpp_Subtype}(\text{bitfield?})): \text{is_finite}[\text{int}](\lambda (i: \text{int}): i = 0) \wedge \neg \text{empty?}[\text{int}](\lambda (i: \text{int}): i = 0)$

Repeatedly Skolemizing and flattening,

Hiding formulas: -1,

Trying repeated skolemization, instantiation, and if-lifting,

Instantiating the top quantifier in 1 with the terms: (1 LAMBDA (P: (LAMBDA (i: int): i = 0)): 0),

which is trivially true.

This completes the proof of range_bitfield_TCC1.2.

Q.E.D.

C.47.2 Cpp_bitfield.bitfield_range_underlying_TCC1

Terse proof for `bitfield_range_underlying_TCC1`.

`bitfield_range_underlying_TCC1`:

{1} $\forall (b: \text{Cpp_Subtype}(\text{bitfield?})):$
 $\text{array?}(b) \vee$
 $\text{pointer?}(b) \vee$
 $\text{reference?}(b) \vee$
 $\text{bitfield?}(b) \vee \text{enum?}(b) \vee \text{pointer_to_member?}(b) \vee \text{const?}(b) \vee \text{volatile?}(b)$

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `bitfield_range_underlying_TCC1`.
Q.E.D.

C.47.3 Cpp_bitfield.bitfield_range_underlying_TCC2

Terse proof for `bitfield_range_underlying_TCC2`.

`bitfield_range_underlying_TCC2`:

{1} $\forall (b: \text{Cpp_Subtype}(\text{bitfield?})):$ $\text{enum?}(\text{typ}(b)) \supset \text{Cpp_Type?}(\text{typ}(b))$

Repeatedly Skolemizing and flattening,
Expanding the definition of `Cpp_Type?`,
Repeatedly Skolemizing and flattening,
Instantiating quantified variables,
Expanding the definition of subterm,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `bitfield_range_underlying_TCC2`.
Q.E.D.

C.47.4 Cpp_bitfield.bitfield_range_underlying_TCC3

Terse proof for `bitfield_range_underlying_TCC3`.

`bitfield_range_underlying_TCC3`:

{1} $\forall (b: \text{Cpp_Subtype}(\text{bitfield?})):$ $\neg \text{enum?}(\text{typ}(b)) \supset \text{non_bool_integral?}(\text{typ}(b))$

Repeatedly Skolemizing and flattening,
Expanding the definition of `Cpp_Type?`,
Instantiating quantified variables,
Expanding the definition of subterm,
Applying disjunctive simplification to flatten sequent,
Keeping (-16 -29 1 2) and hiding *,
Expanding the definition of `bitfield_underlying_integral_or_enum_type?`,
Expanding the definition of `non_bool_integral_enum?`,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `bitfield_range_underlying_TCC3`.
Q.E.D.

C.47.5 Cpp_bitfield.dt_bitfield_TCC1

Terse proof for dt_bitfield_TCC1.

dt_bitfield_TCC1:

{1}	$\exists (x_1:$ $\quad [b: \text{Cpp_Subtype}(\text{bitfield?}) \rightarrow$ $\quad \quad (\text{interpreted_data_type?}[\text{((range_bitfield}(b))]))]:$ $\quad \quad \text{TRUE}$
-----	---

Instantiating the top quantifier in 1 with the terms: $(\lambda (b: \text{Cpp_Subtype}(\text{bitfield?})): \text{choose}(\text{interpreted_data_type?}[\text{((range_bitfield}(b)))]))$,

we get 2 subgoals:

dt_bitfield_TCC1.1:

{1}	TRUE
-----	------

which is trivially true.

This completes the proof of dt_bitfield_TCC1.1.

dt_bitfield_TCC1.2:

{1}	$\forall (b: \text{Cpp_Subtype}(\text{bitfield?})):$ $\quad \text{nonempty?}[\text{Interpreted_data_type}[\text{((range_bitfield}(b)))]]$ $\quad \quad (\text{interpreted_data_type?}[\text{((range_bitfield}(b)))])$
-----	--

Repeatedly Skolemizing and flattening,
 Expanding the definition of nonempty?,
 Expanding the definition of empty?,
 Expanding the definition of member,
 Using lemma dt_bitfield_exists,
 Repeatedly Skolemizing and flattening,
 Instantiating quantified variables,
 This completes the proof of dt_bitfield_TCC1.2.
 Q.E.D.

C.48 Proofs for Cpp_bool (types.pvs)

C.48.1 Cpp_bool.range_bool_TCC1

Terse proof for range_bool_TCC1.

range_bool_TCC1:

{1}	$\text{is_finite}[\text{bool}](\text{fullset}[\text{bool}]) \wedge \neg \text{empty?}[\text{bool}](\text{fullset}[\text{bool}])$
-----	---

Trying repeated skolemization, instantiation, and if-lifting,

Instantiating the top quantifier in 1 with the terms: (2 LAMBDA (b: (fullset[bool])): IF b THEN 0 ELSE 1 ENDIF),

Repeatedly Skolemizing and flattening,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of range_bool_TCC1.

Q.E.D.

C.48.2 Cpp_bool.dt_bool_TCC1

Terse proof for dt_bool_TCC1.

dt_bool_TCC1:

$$\{1\} \quad \exists (x: (\text{pod_data_type?}[\text{bool}])): \text{TRUE}$$

Rewriting using dt_bool_exists, matching in *,
This completes the proof of dt_bool_TCC1.
Q.E.D.

C.49 Proofs for Cpp_char (types.pvs)

C.49.1 Cpp_char.dt_char_TCC1

Terse proof for dt_char_TCC1.

dt_char_TCC1:

$$\{1\} \quad \exists (x_1: (\text{pod_data_type?}[(\text{range_char})])): \text{TRUE}$$

Rewriting using dt_char_exists, matching in *,
This completes the proof of dt_char_TCC1.
Q.E.D.

C.49.2 Cpp_char.value_bitmask_char_TCC1

Terse proof for value_bitmask_char_TCC1.

value_bitmask_char_TCC1:

$$\{1\} \quad \exists (x_1: \{l: \text{list}[\text{Byte}] \mid \text{length}[\text{Byte}](l) = \text{size}(\text{uidt}(\text{dt_char}))\}): \text{TRUE}$$

Using lemma list_all_length[Byte],
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of value_bitmask_char_TCC1.
Q.E.D.

C.49.3 Cpp_char.min_char

Terse proof for min_char.

min_char:

$$\{1\} \quad \text{min}(\text{range_char}) \leq 0$$

Using lemma char_is_uchar_or_schar,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
we get 2 subgoals:

min_char.1:

$$\begin{array}{l} \{-1\} \quad \text{range_char} = \text{range_schar} \\ \{1\} \quad \text{min}(\text{range_char}) \leq 0 \end{array}$$

Replacing using formula -1,

Using lemma min_schar,
 Using lemma dt_schar_bits,
 Using lemma min_bits_per_byte,
 Using lemma bits_per_byte_minimum,
 Rewriting using dt_schar_size, matching in *,
 Rewriting using dt_uchar_size, matching in *,
 Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of `min_char.1`.

`min_char.2:`

$$\frac{\begin{array}{l} \{-1\} \text{ range_char} = \text{extend}[\text{int}, \text{nat}, \text{bool}, \text{FALSE}](\text{range_uchar}) \\ \{1\} \text{ min}(\text{range_char}) \leq 0 \end{array}}{\quad}$$

Using lemma min_bits_per_byte,
 Using lemma bits_per_byte_minimum,
 Using lemma binary_range_uchar,
 Using lemma dt_uchar_bits,
 Rewriting using dt_uchar_size, matching in *,
 Using lemma expt_ge1,
 we get 2 subgoals:

`min_char.2.1:`

$$\frac{\begin{array}{l} \{-1\} 2^{(\text{max_value_bits_uchar} - 8)} \geq 1 \\ \{-2\} \text{max_value_bits_uchar} = 1 \times \text{bits_per_byte} \\ \{-3\} \text{range_uchar}(0) \equiv 0 \leq \text{max}(\text{range_uchar}) \\ \{-4\} \text{bits_per_byte} \geq \text{min_bits_per_byte} \\ \{-5\} \text{min_bits_per_byte} = 8 \\ \{-6\} \text{range_char} = \text{extend}[\text{int}, \text{nat}, \text{bool}, \text{FALSE}](\text{range_uchar}) \end{array}}{\begin{array}{l} \{1\} \text{min}(\text{range_char}) \leq 0 \end{array}}$$

Trying repeated skolemization, instantiation, and if-lifting,
 Keeping (-3 1) and hiding *,
 Adding type constraints for `min(extend[int, nat, bool, FALSE](range_uchar))`,
 Instantiating the top quantifier in -2 with the terms: (0),
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 This completes the proof of `min_char.2.1`.

`min_char.2.2:`

$$\frac{\begin{array}{l} \{-1\} \text{max_value_bits_uchar} = 1 \times \text{bits_per_byte} \\ \{-2\} \text{range_uchar}(0) \equiv 0 \leq \text{max}(\text{range_uchar}) \\ \{-3\} \text{bits_per_byte} \geq \text{min_bits_per_byte} \\ \{-4\} \text{min_bits_per_byte} = 8 \\ \{-5\} \text{range_char} = \text{extend}[\text{int}, \text{nat}, \text{bool}, \text{FALSE}](\text{range_uchar}) \end{array}}{\begin{array}{l} \{1\} \text{max_value_bits_uchar} - 8 \geq 0 \\ \{2\} \text{min}(\text{range_char}) \leq 0 \end{array}}$$

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 This completes the proof of `min_char.2.2`.
 Q.E.D.

C.49.4 Cpp_char.max_char

Terse proof for `max_char`.

C Proof scripts

`max_char`:

{1}	$\max(\text{range_char}) \geq 127$
-----	-------------------------------------

Using lemma `char_is_uchar_or_schar`,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 we get 2 subgoals:

`max_char.1`:

{-1}	$\text{range_char} = \text{range_schar}$
{1}	$\max(\text{range_char}) \geq 127$

Replacing using formula -1,
 Using lemma `max_schar`,
 Using lemma `dt_schar_bits`,
 Using lemma `min_bits_per_byte`,
 Using lemma `bits_per_byte_minimum`,
 Rewriting using `dt_schar_size`, matching in *,
 Rewriting using `dt_uchar_size`, matching in *,
 Using lemma `expt_ge1`,
 we get 2 subgoals:

`max_char.1.1`:

{-1}	$2^{(\max_value_bits_schar - 7)} \geq 1$
{-2}	$\text{bits_per_byte} \geq \text{min_bits_per_byte}$
{-3}	$\text{min_bits_per_byte} = 8$
{-4}	$\max_value_bits_schar \geq 1 \times \text{bits_per_byte} - 1$
{-5}	$\max(\text{range_schar}) \geq (2^{\max_value_bits_schar}) - 1$
{-6}	$\text{range_char} = \text{range_schar}$
{1}	$\max(\text{range_schar}) \geq 127$

Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of `max_char.1.1`.

`max_char.1.2`:

{-1}	$\text{bits_per_byte} \geq \text{min_bits_per_byte}$
{-2}	$\text{min_bits_per_byte} = 8$
{-3}	$\max_value_bits_schar \geq 1 \times \text{bits_per_byte} - 1$
{-4}	$\max(\text{range_schar}) \geq (2^{\max_value_bits_schar}) - 1$
{-5}	$\text{range_char} = \text{range_schar}$
{1}	$\max_value_bits_schar - 7 \geq 0$
{2}	$\max(\text{range_schar}) \geq 127$

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 This completes the proof of `max_char.1.2`.

`max_char.2`:

{-1}	$\text{range_char} = \text{extend}[\text{int}, \text{nat}, \text{bool}, \text{FALSE}](\text{range_uchar})$
{1}	$\max(\text{range_char}) \geq 127$

Using lemma `max_uchar`,
 Using lemma `dt_uchar_bits`,
 Using lemma `min_bits_per_byte`,
 Using lemma `bits_per_byte_minimum`,
 Rewriting using `dt_uchar_size`, matching in *,
 Using lemma `expt_ge1`,

we get 2 subgoals:

`max_char.2.1:`

{-1}	$2^{(\text{max_value_bits_uchar} - 8)} \geq 1$
{-2}	$\text{bits_per_byte} \geq \text{min_bits_per_byte}$
{-3}	$\text{min_bits_per_byte} = 8$
{-4}	$\text{max_value_bits_uchar} = 1 \times \text{bits_per_byte}$
{-5}	$\text{max}(\text{range_uchar}) \geq (2^{\text{max_value_bits_uchar}}) - 1$
{-6}	$\text{range_char} = \text{extend}[\text{int}, \text{nat}, \text{bool}, \text{FALSE}](\text{range_uchar})$
{1}	
	$\text{max}(\text{range_char}) \geq 127$

Replacing using formula -6,

Trying repeated skolemization, instantiation, and if-lifting,

Adding type constraints for $\text{max}(\text{extend}[\text{int}, \text{nat}, \text{bool}, \text{FALSE}](\text{range_uchar}))$,

Instantiating the top quantifier in -2 with the terms: (127),

Expanding the definition of extend,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Rewriting using `binary_range_uchar`, matching in * where n gets 127,

This completes the proof of `max_char.2.1`.

`max_char.2.2:`

{-1}	$\text{bits_per_byte} \geq \text{min_bits_per_byte}$
{-2}	$\text{min_bits_per_byte} = 8$
{-3}	$\text{max_value_bits_uchar} = 1 \times \text{bits_per_byte}$
{-4}	$\text{max}(\text{range_uchar}) \geq (2^{\text{max_value_bits_uchar}}) - 1$
{-5}	$\text{range_char} = \text{extend}[\text{int}, \text{nat}, \text{bool}, \text{FALSE}](\text{range_uchar})$
{1}	
	$\text{max_value_bits_uchar} - 8 \geq 0$
{2}	$\text{max}(\text{range_char}) \geq 127$

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `max_char.2.2`.

Q.E.D.

C.49.5 Cpp_char.char_binary

Terse proof for `char_binary`.

`char_binary:`

{1}	$\forall (i: \text{int}): \text{range_char}(i) \equiv (\text{min}(\text{range_char}) \leq i \wedge i \leq \text{max}(\text{range_char}))$
-----	--

Installing automatic rewrites from: extend

Repeatedly Skolemizing and flattening,

Using lemma `char_is_uchar_or_schar`,

Splitting conjunctions,

we get 2 subgoals:

`char_binary.1:`

{-1}	$\text{range_char} = \text{range_schar}$
{-2}	$\text{integer_pred}(i')$
{1}	
	$\text{range_char}(i') \equiv (\text{min}(\text{range_char}) \leq i' \wedge i' \leq \text{max}(\text{range_char}))$

Replacing using formula -1,

Using lemma `schar_binary`,

This completes the proof of `char_binary.1`.

C Proof scripts

`char_binary.2:`

$$\frac{\begin{array}{l} \{-1\} \text{ range_char} = \text{extend}[\text{int}, \text{nat}, \text{bool}, \text{FALSE}](\text{range_uchar}) \\ \{-2\} \text{ integer_pred}(i') \end{array}}{\{-1\} \text{ range_char}(i') \equiv (\min(\text{range_char}) \leq i' \wedge i' \leq \max(\text{range_char}))}$$

Replacing using formula -1,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 3 subgoals:

`char_binary.2.1:`

$$\frac{\begin{array}{l} \{-1\} \text{ range_char} = \text{extend}[\text{int}, \text{nat}, \text{bool}, \text{FALSE}](\text{range_uchar}) \\ \{-2\} \text{ integer_pred}(i') \\ \{-3\} \text{ extend}[\text{int}, \text{nat}, \text{bool}, \text{FALSE}](\text{range_uchar})(i') \end{array}}{\{-1\} i' \leq \max(\text{extend}[\text{int}, \text{nat}, \text{bool}, \text{FALSE}](\text{range_uchar}))}$$

Adding type constraints for $\max(\text{extend}[\text{int}, \text{nat}, \text{bool}, \text{FALSE}](\text{range_uchar}))$,

Instantiating the top quantifier in -2 with the terms: $(i!1)$,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `char_binary.2.1`.

`char_binary.2.2:`

$$\frac{\begin{array}{l} \{-1\} \text{ range_char} = \text{extend}[\text{int}, \text{nat}, \text{bool}, \text{FALSE}](\text{range_uchar}) \\ \{-2\} \text{ integer_pred}(i') \\ \{-3\} \text{ extend}[\text{int}, \text{nat}, \text{bool}, \text{FALSE}](\text{range_uchar})(i') \end{array}}{\{-1\} \min(\text{extend}[\text{int}, \text{nat}, \text{bool}, \text{FALSE}](\text{range_uchar})) \leq i'}$$

Adding type constraints for $\min(\text{extend}[\text{int}, \text{nat}, \text{bool}, \text{FALSE}](\text{range_uchar}))$,

Instantiating the top quantifier in -2 with the terms: $(i!1)$,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `char_binary.2.2`.

`char_binary.2.3:`

$$\frac{\begin{array}{l} \{-1\} \text{ range_char} = \text{extend}[\text{int}, \text{nat}, \text{bool}, \text{FALSE}](\text{range_uchar}) \\ \{-2\} \text{ integer_pred}(i') \\ \{-3\} \min(\text{extend}[\text{int}, \text{nat}, \text{bool}, \text{FALSE}](\text{range_uchar})) \leq i' \\ \{-4\} i' \leq \max(\text{extend}[\text{int}, \text{nat}, \text{bool}, \text{FALSE}](\text{range_uchar})) \end{array}}{\{-1\} \text{ extend}[\text{int}, \text{nat}, \text{bool}, \text{FALSE}](\text{range_uchar})(i')}$$

Adding type constraints for $\max(\text{extend}[\text{int}, \text{nat}, \text{bool}, \text{FALSE}](\text{range_uchar}))$,

Adding type constraints for $\min(\text{extend}[\text{int}, \text{nat}, \text{bool}, \text{FALSE}](\text{range_uchar}))$,

Using lemma `binary_range_uchar`,

we get 2 subgoals:

char_binary.2.3.1:

{-1}	$\text{range_uchar}(i') \equiv i' \leq \max(\text{range_uchar})$
{-2}	$\text{extend}[\text{int}, \text{nat}, \text{bool}, \text{FALSE}]$ $(\text{range_uchar})(\min(\text{extend}[\text{int}, \text{nat}, \text{bool}, \text{FALSE}](\text{range_uchar})))$
{-3}	$\forall (x: \text{int}):$ $\text{extend}[\text{int}, \text{nat}, \text{bool}, \text{FALSE}](\text{range_uchar})(x) \supset$ $\min(\text{extend}[\text{int}, \text{nat}, \text{bool}, \text{FALSE}](\text{range_uchar})) \leq x$
{-4}	$\text{extend}[\text{int}, \text{nat}, \text{bool}, \text{FALSE}]$ $(\text{range_uchar})(\max(\text{extend}[\text{int}, \text{nat}, \text{bool}, \text{FALSE}](\text{range_uchar})))$
{-5}	$\forall (x: \text{int}):$ $\text{extend}[\text{int}, \text{nat}, \text{bool}, \text{FALSE}](\text{range_uchar})(x) \supset$ $x \leq \max(\text{extend}[\text{int}, \text{nat}, \text{bool}, \text{FALSE}](\text{range_uchar}))$
{-6}	$\text{range_char} = \text{extend}[\text{int}, \text{nat}, \text{bool}, \text{FALSE}](\text{range_uchar})$
{-7}	$\text{integer_pred}(i')$
{-8}	$\min(\text{extend}[\text{int}, \text{nat}, \text{bool}, \text{FALSE}](\text{range_uchar})) \leq i'$
{-9}	$i' \leq \max(\text{extend}[\text{int}, \text{nat}, \text{bool}, \text{FALSE}](\text{range_uchar}))$
{1}	$\text{extend}[\text{int}, \text{nat}, \text{bool}, \text{FALSE}](\text{range_uchar})(i')$

Using lemma binary_range_uchar,

we get 2 subgoals:

char_binary.2.3.1.1:

{-1}	$\text{range_uchar}(\max(\text{extend}[\text{int}, \text{nat}, \text{bool}, \text{FALSE}](\text{range_uchar}))) \equiv$ $\max(\text{extend}[\text{int}, \text{nat}, \text{bool}, \text{FALSE}](\text{range_uchar})) \leq \max(\text{range_uchar})$
{-2}	$\text{range_uchar}(i') \equiv i' \leq \max(\text{range_uchar})$
{-3}	$\text{extend}[\text{int}, \text{nat}, \text{bool}, \text{FALSE}]$ $(\text{range_uchar})(\min(\text{extend}[\text{int}, \text{nat}, \text{bool}, \text{FALSE}](\text{range_uchar})))$
{-4}	$\forall (x: \text{int}):$ $\text{extend}[\text{int}, \text{nat}, \text{bool}, \text{FALSE}](\text{range_uchar})(x) \supset$ $\min(\text{extend}[\text{int}, \text{nat}, \text{bool}, \text{FALSE}](\text{range_uchar})) \leq x$
{-5}	$\text{extend}[\text{int}, \text{nat}, \text{bool}, \text{FALSE}]$ $(\text{range_uchar})(\max(\text{extend}[\text{int}, \text{nat}, \text{bool}, \text{FALSE}](\text{range_uchar})))$
{-6}	$\forall (x: \text{int}):$ $\text{extend}[\text{int}, \text{nat}, \text{bool}, \text{FALSE}](\text{range_uchar})(x) \supset$ $x \leq \max(\text{extend}[\text{int}, \text{nat}, \text{bool}, \text{FALSE}](\text{range_uchar}))$
{-7}	$\text{range_char} = \text{extend}[\text{int}, \text{nat}, \text{bool}, \text{FALSE}](\text{range_uchar})$
{-8}	$\text{integer_pred}(i')$
{-9}	$\min(\text{extend}[\text{int}, \text{nat}, \text{bool}, \text{FALSE}](\text{range_uchar})) \leq i'$
{-10}	$i' \leq \max(\text{extend}[\text{int}, \text{nat}, \text{bool}, \text{FALSE}](\text{range_uchar}))$
{1}	$\text{extend}[\text{int}, \text{nat}, \text{bool}, \text{FALSE}](\text{range_uchar})(i')$

Expanding the definition of extend,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of char_binary.2.3.1.1.

char_binary.2.3.1.2:

{-1}	range_uchar(i') $\equiv i' \leq \max(\text{range_uchar})$
{-2}	extend[int, nat, bool, FALSE] (range_uchar)(min(extend[int, nat, bool, FALSE](range_uchar)))
{-3}	$\forall (x: \text{int}):$ extend[int, nat, bool, FALSE](range_uchar)(x) \supset min(extend[int, nat, bool, FALSE](range_uchar)) $\leq x$
{-4}	extend[int, nat, bool, FALSE] (range_uchar)(max(extend[int, nat, bool, FALSE](range_uchar)))
{-5}	$\forall (x: \text{int}):$ extend[int, nat, bool, FALSE](range_uchar)(x) \supset $x \leq \max(\text{extend}[\text{int}, \text{nat}, \text{bool}, \text{FALSE}](\text{range_uchar}))$
{-6}	range_char = extend[int, nat, bool, FALSE](range_uchar)
{-7}	integer_pred(i')
{-8}	min(extend[int, nat, bool, FALSE](range_uchar)) $\leq i'$
{-9}	$i' \leq \max(\text{extend}[\text{int}, \text{nat}, \text{bool}, \text{FALSE}](\text{range_uchar}))$
{1}	max[int, restrict[[real, real], [int, int], boolean](\leq) (extend[int, nat, bool, FALSE](range_uchar)) ≥ 0
{2}	extend[int, nat, bool, FALSE](range_uchar)(i')

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of char_binary.2.3.1.2.

char_binary.2.3.2:

{-1}	extend[int, nat, bool, FALSE] (range_uchar)(min(extend[int, nat, bool, FALSE](range_uchar)))
{-2}	$\forall (x: \text{int}):$ extend[int, nat, bool, FALSE](range_uchar)(x) \supset min(extend[int, nat, bool, FALSE](range_uchar)) $\leq x$
{-3}	extend[int, nat, bool, FALSE] (range_uchar)(max(extend[int, nat, bool, FALSE](range_uchar)))
{-4}	$\forall (x: \text{int}):$ extend[int, nat, bool, FALSE](range_uchar)(x) \supset $x \leq \max(\text{extend}[\text{int}, \text{nat}, \text{bool}, \text{FALSE}](\text{range_uchar}))$
{-5}	range_char = extend[int, nat, bool, FALSE](range_uchar)
{-6}	integer_pred(i')
{-7}	min(extend[int, nat, bool, FALSE](range_uchar)) $\leq i'$
{-8}	$i' \leq \max(\text{extend}[\text{int}, \text{nat}, \text{bool}, \text{FALSE}](\text{range_uchar}))$
{1}	$i' \geq 0$
{2}	extend[int, nat, bool, FALSE](range_uchar)(i')

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of char_binary.2.3.2.

Q.E.D.

C.50 Proofs for Cpp_class (types.pvs)

C.50.1 Cpp_class.uidt_class_TCC1

Terse proof for uidt_class_TCC1.

```
uidt_class_TCC1:
```

```
{1}  ∃ (x: [Cpp_Subtype(class?) → {uidt: (uninterpreted_data_type?) | size(uidt) > 0}]):
      TRUE
```

Instantiating the top quantifier in 1 with the terms: $(\lambda (c: \text{Cpp_Subtype}(\text{class?})): \text{choose}(\{\text{uidt}: (\text{uninterpreted_data_type?}) \mid \text{size}(\text{uidt}) > 0\}))$,

we get 2 subgoals:

```
uidt_class_TCC1.1:
```

```
{1}  TRUE
```

which is trivially true.

This completes the proof of uidt_class_TCC1.1.

```
uidt_class_TCC1.2:
```

```
{1}  ∃ (c: Cpp_Subtype(class?):
      nonempty?[(uninterpreted_data_type?)
                {uidt: (uninterpreted_data_type?) | size(uidt) > 0}])
```

Trying repeated skolemization, instantiation, and if-lifting,

Instantiating the top quantifier in -3 with the terms: $((\# \text{size} := 1, \text{valid?} := \text{LAMBDA (l: list[Byte], \text{a: Address}): \text{FALSE} \#))$,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of uidt_class_TCC1.2.

Q.E.D.

C.51 Proofs for Cpp_const_volatile (types.pvs)

C.51.1 Cpp_const_volatile.dt_volatile_TCC1

Terse proof for dt_volatile_TCC1.

```
dt_volatile_TCC1:
```

```
{1}  ∃ (x: [(interpreted_data_type?[Data]) → (interpreted_data_type?[Data])]): TRUE
```

Instantiating the top quantifier in 1 with the terms: $(\lambda (\text{idt}: (\text{interpreted_data_type?}[\text{Data}])): \text{idt})$, which is trivially true.

This completes the proof of dt_volatile_TCC1.

Q.E.D.

C.52 Proofs for Cpp_double (types.pvs)

C.52.1 Cpp_double.dt_double_TCC1

Terse proof for dt_double_TCC1.

```
dt_double_TCC1:
```

```
{1}  ∃ (x1: (pod_data_type?[(range_double)])): TRUE
```


Rewriting using `dt_double_exists`, matching in `*`,
 This completes the proof of `dt_double_TCC1`.
 Q.E.D.

C.53 Proofs for Cpp_enum (types.pvs)

C.53.1 Cpp_enum.valid_range?_TCC1

Terse proof for `valid_range?_TCC1`.

`valid_range?_TCC1`:

$\{1\} \quad \forall (e: \text{Cpp_Subtype}(\text{enum?}), r: \text{non_empty_finite_set}[\text{int}]):$ $\text{array?}(e) \vee$ $\text{pointer?}(e) \vee$ $\text{reference?}(e) \vee$ $\text{bitfield?}(e) \vee \text{enum?}(e) \vee \text{pointer_to_member?}(e) \vee \text{const?}(e) \vee \text{volatile?}(e)$
--

Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of `valid_range?_TCC1`.
 Q.E.D.

C.53.2 Cpp_enum.valid_range?_TCC2

Terse proof for `valid_range?_TCC2`.

`valid_range?_TCC2`:

$\{1\} \quad \forall (e: \text{Cpp_Subtype}(\text{enum?}), r: \text{non_empty_finite_set}[\text{int}]): \text{non_bool_integral?}(\text{typ}(e))$

Repeatedly Skolemizing and flattening,
 Expanding the definition of `Cpp_Type?`,
 Instantiating quantified variables,
 Expanding the definition of subterm,
 Expanding the definition of `enum_underlying_integral?`,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 This completes the proof of `valid_range?_TCC2`.
 Q.E.D.

C.53.3 Cpp_enum.range_enum_TCC1

Terse proof for `range_enum_TCC1`.

`range_enum_TCC1`:

$\{1\} \quad \exists (x: [e: \text{Cpp_Subtype}(\text{enum?}) \rightarrow (\text{valid_range?}(e))]): \text{TRUE}$
--

Instantiating the top quantifier in 1 with the terms: $(\lambda (e: \text{Cpp_Subtype}(\text{enum?})): \text{range_integral}(\text{typ}(e)))$,
 we get 3 subgoals:

`range_enum_TCC1.1`:

$\{1\} \quad \text{TRUE}$

which is trivially true.

This completes the proof of `range_enum_TCC1.1`.

`range_enum_TCC1.2`:

{1} $\forall (e: \text{Cpp_Subtype}(\text{enum?})): \text{valid_range?}(e)(\text{range_integral}(\text{typ}(e)))$

Repeatedly Skolemizing and flattening,
 Expanding the definition of `Cpp_Type?`,
 Instantiating quantified variables,
 Expanding the definition of `subterm`,
 Applying disjunctive simplification to flatten sequent,
 Keeping (-18 -19 1) and hiding *,
 Expanding the definition of `enum_underlying_integral?`,
 Expanding the definition of `enum_constants?`,
 Expanding the definition of `valid_range?`,
 Rewriting using `subset_reflexive`, matching in *,
 This completes the proof of `range_enum_TCC1.2`.

`range_enum_TCC1.3`:

{1} $\forall (e: \text{Cpp_Subtype}(\text{enum?})): \text{non_bool_integral?}(\text{typ}(e))$

Repeatedly Skolemizing and flattening,
 Expanding the definition of `Cpp_Type?`,
 Instantiating quantified variables,
 Expanding the definition of `subterm`,
 Applying disjunctive simplification to flatten sequent,
 Keeping (-18 1) and hiding *,
 Expanding the definition of `enum_underlying_integral?`,
 which is trivially true.
 This completes the proof of `range_enum_TCC1.3`.
 Q.E.D.

C.53.4 Cpp_enum.dt_enum_TCC1

Terse proof for `dt_enum_TCC1`.

`dt_enum_TCC1`:

{1} $\exists (x_1: [e: \text{Cpp_Subtype}(\text{enum?}) \rightarrow (\text{pod_data_type?}[\text{((range_enum}(e))])]): \text{TRUE}$

Instantiating the top quantifier in 1 with the terms: $(\lambda (e: \text{Cpp_Subtype}(\text{enum?})): \text{choose}(\text{pod_data_type?}[\text{((range_enum}(e)))]))$,

we get 2 subgoals:

`dt_enum_TCC1.1`:

{1} TRUE

which is trivially true.

This completes the proof of `dt_enum_TCC1.1`.

dt_enum_TCC1.2:

```
{1}  ∀ (e: Cpp_Subtype(enum?):
      nonempty?[Interpreted_data_type[((range_enum(e)))]
      (pod_data_type?[((range_enum(e)))]
```

Repeatedly Skolemizing and flattening,
 Expanding the definition of nonempty?,
 Expanding the definition of empty?,
 Expanding the definition of member,
 Using lemma dt_enum_exists,
 Repeatedly Skolemizing and flattening,
 Instantiating quantified variables,
 This completes the proof of dt_enum_TCC1.2.
 Q.E.D.

C.53.5 Cpp_enum.value_bitmask_enum_TCC1

Terse proof for value_bitmask_enum_TCC1.

value_bitmask_enum_TCC1:

```
{1}  ∃ (x1:
      [e: Cpp_Subtype(enum?) →
      {l: list[Byte] | length[Byte](l) = size(uidt(dt_enum(e)))}]):
      TRUE
```

Instantiating the top quantifier in 1 with the terms: $(\lambda (e: \text{Cpp_Subtype}(\text{enum?})): \text{choose}(\{l: \text{list}[\text{Byte}] \mid \text{length}[\text{Byte}](l) = \text{size}(\text{uidt}(\text{dt_enum}(e)))\}))$,
 we get 2 subgoals:

value_bitmask_enum_TCC1.1:

```
{1}  TRUE
```

which is trivially true.

This completes the proof of value_bitmask_enum_TCC1.1.

value_bitmask_enum_TCC1.2:

```
{1}  ∀ (e: Cpp_Subtype(enum?):
      nonempty?[list[Byte]]
      ({l: list[Byte] | length[Byte](l) = size(uidt(dt_enum(e)))})
```

Repeatedly Skolemizing and flattening,
 Expanding the definition of nonempty?,
 Expanding the definition of empty?,
 Expanding the definition of member,
 Using lemma list_all_length[Byte],
 Repeatedly Skolemizing and flattening,
 Instantiating the top quantifier in -5 with the terms: (!1),
 which is trivially true.
 This completes the proof of value_bitmask_enum_TCC1.2.
 Q.E.D.

C.53.6 Cpp_enum.enum_size_TCC1

Terse proof for enum_size_TCC1.

enum_size_TCC1:

$\{1\} \quad \forall (e: \text{Cpp_Subtype}(\text{enum?})): \\ \text{array?}(e) \vee \\ \text{pointer?}(e) \vee \\ \text{reference?}(e) \vee \\ \text{bitfield?}(e) \vee \text{enum?}(e) \vee \text{pointer_to_member?}(e) \vee \text{const?}(e) \vee \text{volatile?}(e)$

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of enum_size_TCC1.

Q.E.D.

C.53.7 Cpp_enum.enum_size_TCC2

Terse proof for enum_size_TCC2.

enum_size_TCC2:

$\{1\} \quad \forall (e: \text{Cpp_Subtype}(\text{enum?})): \text{non_bool_integral?}(\text{typ}(e))$
--

Repeatedly Skolemizing and flattening,

Expanding the definition of Cpp_Type?,

Instantiating quantified variables,

Expanding the definition of subterm,

Applying disjunctive simplification to flatten sequent,

Keeping (-18 -27 1) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of enum_size_TCC2.

Q.E.D.

C.54 Proofs for Cpp_float (types.pvs)

C.54.1 Cpp_float.range_float_TCC1

Terse proof for range_float_TCC1.

range_float_TCC1:

$\{1\} \quad \exists (x: \text{non_empty_finite_set}[\text{extended_real}]): \text{TRUE}$

Instantiating the top quantifier in 1 with the terms: $(\lambda (n: \text{extended_real}): n = 0)$,

we get 2 subgoals:

range_float_TCC1.1:

$\{1\} \quad \text{TRUE}$

which is trivially true.

This completes the proof of range_float_TCC1.1.

range_float_TCC1.2:

{1} is_finite[extended_real](λ (n: extended_real): n = 0) ∧
 ¬ empty?[extended_real](λ (n: extended_real): n = 0)

Trying repeated skolemization, instantiation, and if-lifting,
 Instantiating the top quantifier in 1 with the terms: (1 LAMBDA (P: (LAMBDA (n: extended_real): n = 0)): 0),
 which is trivially true.
 This completes the proof of range_float_TCC1.2.
 Q.E.D.

C.54.2 Cpp_float.dt_float_TCC1

Terse proof for dt_float_TCC1.

dt_float_TCC1:

{1} ∃ (x₁: (pod_data_type?(range_float))): TRUE

Rewriting using dt_float_exists, matching in *,
 This completes the proof of dt_float_TCC1.
 Q.E.D.

C.55 Proofs for Cpp_function (types.pvs)

C.55.1 Cpp_function.uidt_function_TCC1

Terse proof for uidt_function_TCC1.

uidt_function_TCC1:

{1} ∀ (f: Cpp_Subtype(function?):
 uninterpreted_data_type?((#size := 0, valid? := λ (l: list[Byte], a: Ad-
 dress): FALSE#))

Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of uidt_function_TCC1.
 Q.E.D.

C.55.2 Cpp_function.dt_vtable_TCC1

Terse proof for dt_vtable_TCC1.

dt_vtable_TCC1:

{1} ∃ (x₁:
 [c: Cpp_Subtype(class?) →
 (interpreted_data_type?([(singleton[Vtable](vtable(c))]))]):
 TRUE

Instantiating the top quantifier in 1 with the terms: (λ (c: Cpp_Subtype(class?):
 choose(interpreted_data_type?([(singleton[Vtable](vtable(c))])))),
 we get 2 subgoals:

dt_vtable_TCC1.1:

{1} TRUE

which is trivially true.

This completes the proof of dt_vtable_TCC1.1.

dt_vtable_TCC1.2:

{1} $\forall (c: \text{Cpp_Subtype}(\text{class?})): \text{nonempty?}[\text{Interpreted_data_type}[\text{((singleton[Vtable](vtable(c)))}]]]$
 $(\text{interpreted_data_type?}[\text{((singleton[Vtable](vtable(c)))}])])$

Repeatedly Skolemizing and flattening,
 Expanding the definition of nonempty?,
 Expanding the definition of empty?,
 Expanding the definition of member,
 Using lemma vtable_idt_exists,
 Repeatedly Skolemizing and flattening,
 Instantiating quantified variables,
 This completes the proof of dt_vtable_TCC1.2.
 Q.E.D.

C.56 Proofs for Cpp_int (types.pvs)

C.56.1 Cpp_int.range_int_TCC1

Terse proof for range_int_TCC1.

range_int_TCC1:

{1} $\exists (x: \text{non_empty_finite_set}[\text{int}]): \text{TRUE}$

Instantiating the top quantifier in 1 with the terms: $(\lambda (n: \text{int}): n = 0)$,

we get 2 subgoals:

range_int_TCC1.1:

{1} TRUE

which is trivially true.

This completes the proof of range_int_TCC1.1.

range_int_TCC1.2:

{1} $\text{is_finite}[\text{int}](\lambda (n: \text{int}): n = 0) \wedge \neg \text{empty?}[\text{int}](\lambda (n: \text{int}): n = 0)$

Trying repeated skolemization, instantiation, and if-lifting,

Instantiating the top quantifier in 1 with the terms: $(1 \text{ LAMBDA } (P: (\text{LAMBDA } (n: \text{int}): n = 0)):$
 $0)$,

which is trivially true.

This completes the proof of range_int_TCC1.2.

Q.E.D.

C.56.2 Cpp_int.dt_int_TCC1

Terse proof for dt_int_TCC1.

dt_int_TCC1:

{1} $\exists (x_1: (\text{pod_data_type?}[(\text{range_int})])): \text{TRUE}$

Rewriting using dt_int_exists, matching in *,
This completes the proof of dt_int_TCC1.
Q.E.D.

C.56.3 Cpp_int.value_bitmask_int_TCC1

Terse proof for value_bitmask_int_TCC1.

value_bitmask_int_TCC1:

{1} $\exists (x_1: \{\!| l: \text{list}[\text{Byte}] \mid \text{length}[\text{Byte}](l) = \text{size}(\text{uidt}(\text{dt_int})) \!\}) : \text{TRUE}$

Using lemma list_all_length[Byte],
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of value_bitmask_int_TCC1.
Q.E.D.

C.57 Proofs for Cpp_long (types.pvs)

C.57.1 Cpp_long.range_long_TCC1

Terse proof for range_long_TCC1.

range_long_TCC1:

{1} $\exists (x: \text{non_empty_finite_set}[\text{int}]): \text{TRUE}$

Instantiating the top quantifier in 1 with the terms: $(\lambda (n: \text{int}): n = 0)$,
we get 2 subgoals:
range_long_TCC1.1:

{1} TRUE

which is trivially true.
This completes the proof of range_long_TCC1.1.
range_long_TCC1.2:

{1} $\text{is_finite}[\text{int}](\lambda (n: \text{int}): n = 0) \wedge \neg \text{empty?}[\text{int}](\lambda (n: \text{int}): n = 0)$

Trying repeated skolemization, instantiation, and if-lifting,
Instantiating the top quantifier in 1 with the terms: $(1 \text{ LAMBDA } (P: (\text{LAMBDA } (n: \text{int}): n = 0)):$
 $0)$,
which is trivially true.
This completes the proof of range_long_TCC1.2.
Q.E.D.

C.57.2 Cpp_long.dt_long_TCC1

Terse proof for dt_long_TCC1.

dt_long_TCC1:

{1} $\exists (x_1: (\text{pod_data_type?}[(\text{range_long})])): \text{TRUE}$

Rewriting using dt_long_exists, matching in *,
This completes the proof of dt_long_TCC1.
Q.E.D.

C.57.3 Cpp_long.value_bitmask_long_TCC1

Terse proof for value_bitmask_long_TCC1.

value_bitmask_long_TCC1:

{1} $\exists (x_1: \{l: \text{list}[\text{Byte}] \mid \text{length}[\text{Byte}](l) = \text{size}(\text{uidt}(\text{dt_long}))\}): \text{TRUE}$

Using lemma list_all_length[Byte],
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of value_bitmask_long_TCC1.
Q.E.D.

C.58 Proofs for Cpp_longdouble (types.pvs)

C.58.1 Cpp_longdouble.dt_longdouble_TCC1

Terse proof for dt_longdouble_TCC1.

dt_longdouble_TCC1:

{1} $\exists (x_1: (\text{pod_data_type?}[(\text{range_longdouble})])): \text{TRUE}$

Rewriting using dt_longdouble_exists, matching in *,
This completes the proof of dt_longdouble_TCC1.
Q.E.D.

C.59 Proofs for Cpp_longlong (types.pvs)

C.59.1 Cpp_longlong.range_longlong_TCC1

Terse proof for range_longlong_TCC1.

range_longlong_TCC1:

{1} $\exists (x: \text{non_empty_finite_set}[\text{int}]): \text{TRUE}$

Instantiating the top quantifier in 1 with the terms: $(\lambda (n: \text{int}): n = 0)$,
we get 2 subgoals:
range_longlong_TCC1.1:

{1} TRUE

which is trivially true.

This completes the proof of `range_longlong_TCC1.1`.

`range_longlong_TCC1.2`:

$$\{1\} \text{ is_finite}[\text{int}](\lambda (n: \text{int}): n = 0) \wedge \neg \text{empty?}[\text{int}](\lambda (n: \text{int}): n = 0)$$

Trying repeated skolemization, instantiation, and if-lifting,

Instantiating the top quantifier in 1 with the terms: (1 LAMBDA (P: (LAMBDA (n: int): n = 0)):

0),

which is trivially true.

This completes the proof of `range_longlong_TCC1.2`.

Q.E.D.

C.59.2 Cpp_longlong.dt_longlong_TCC1

Terse proof for `dt_longlong_TCC1`.

`dt_longlong_TCC1`:

$$\{1\} \exists (x_1: (\text{pod_data_type?}[(\text{range_longlong})])): \text{TRUE}$$

Rewriting using `dt_longlong_exists`, matching in *,

This completes the proof of `dt_longlong_TCC1`.

Q.E.D.

C.59.3 Cpp_longlong.value_bitmask_longlong_TCC1

Terse proof for `value_bitmask_longlong_TCC1`.

`value_bitmask_longlong_TCC1`:

$$\{1\} \exists (x_1: \{l: \text{list}[\text{Byte}] \mid \text{length}[\text{Byte}](l) = \text{size}(\text{uidt}(\text{dt_longlong}))\}): \text{TRUE}$$

Using lemma `list_all_length[Byte]`,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `value_bitmask_longlong_TCC1`.

Q.E.D.

C.60 Proofs for Cpp_pointer (types.pvs)

C.60.1 Cpp_pointer.dt_pointer_TCC1

Terse proof for `dt_pointer_TCC1`.

`dt_pointer_TCC1`:

$$\{1\} \exists (x_1: [p: \text{Cpp_Subtype}(\text{pointer?}) \rightarrow (\text{pod_data_type?}[(\text{range_pointer}(p))])]): \text{TRUE}$$

Using lemma `dt_pointer_exists`,

Instantiating the top quantifier in 1 with the terms: (LAMBDA (p: Cpp_Subtype(pointer?): choose(pod_data_type?((range_pointer(p))))),

we get 2 subgoals:

dt_pointer_TCC1.1:

{-1}	$\forall (p: \text{Cpp_Subtype}(\text{pointer?})): \\ \exists (\text{dt}: (\text{pod_data_type?}[\text{((range_pointer(p)))]]): \text{TRUE}$
{1}	TRUE

which is trivially true.

This completes the proof of dt_pointer_TCC1.1.

dt_pointer_TCC1.2:

{-1}	$\forall (p: \text{Cpp_Subtype}(\text{pointer?})): \\ \exists (\text{dt}: (\text{pod_data_type?}[\text{((range_pointer(p)))]]): \text{TRUE}$
{1}	$\forall (p: \text{Cpp_Subtype}(\text{pointer?})): \\ \text{nonempty?}[\text{Interpreted_data_type}[\text{((range_pointer(p)))]}] \\ (\text{pod_data_type?}[\text{((range_pointer(p)))]})$

Repeatedly Skolemizing and flattening,
 Expanding the definition of nonempty?,
 Expanding the definition of empty?,
 Expanding the definition of member,
 Instantiating quantified variables,
 Repeatedly Skolemizing and flattening,
 Instantiating quantified variables,
 This completes the proof of dt_pointer_TCC1.2.
 Q.E.D.

C.60.2 Cpp_pointer.pointers_to_layout_compatible_range_TCC1

Terse proof for pointers_to_layout_compatible_range_TCC1.

pointers_to_layout_compatible_range_TCC1:

{1}	$\forall (p_1, p_2: \text{Cpp_Subtype}(\text{pointer?})): \\ \text{array?}(p_1) \vee \\ \text{pointer?}(p_1) \vee \\ \text{reference?}(p_1) \vee \\ \text{bitfield?}(p_1) \vee \text{enum?}(p_1) \vee \text{pointer_to_member?}(p_1) \vee \text{const?}(p_1) \vee \text{volatile?}(p_1)$
-----	---

Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of pointers_to_layout_compatible_range_TCC1.
 Q.E.D.

C.60.3 Cpp_pointer.pointers_to_layout_compatible_range_TCC2

Terse proof for pointers_to_layout_compatible_range_TCC2.

pointers_to_layout_compatible_range_TCC2:

{1}	$\forall (p_1, p_2: \text{Cpp_Subtype}(\text{pointer?})): \text{Cpp_Type?}(\text{typ}(p_1))$
-----	--

Repeatedly Skolemizing and flattening,
 Keeping (-1 1) and hiding *,
 Expanding the definition of Cpp_Type?,
 Repeatedly Skolemizing and flattening,
 Instantiating quantified variables,

Expanding the definition of subterm,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 This completes the proof of `pointers_to_layout_compatible_range_TCC2`.
 Q.E.D.

C.60.4 Cpp_pointer.pointers_to_layout_compatible_range_TCC3

Terse proof for `pointers_to_layout_compatible_range_TCC3`.
`pointers_to_layout_compatible_range_TCC3`:

$$\{1\} \quad \forall (p_1, p_2: \text{Cpp_Subtype}(\text{pointer?})): \\
\text{array?}(p_2) \vee \\
\text{pointer?}(p_2) \vee \\
\text{reference?}(p_2) \vee \\
\text{bitfield?}(p_2) \vee \text{enum?}(p_2) \vee \text{pointer_to_member?}(p_2) \vee \text{const?}(p_2) \vee \text{volatile?}(p_2)$$

Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of `pointers_to_layout_compatible_range_TCC3`.
 Q.E.D.

C.60.5 Cpp_pointer.pointers_to_layout_compatible_range_TCC4

Terse proof for `pointers_to_layout_compatible_range_TCC4`.
`pointers_to_layout_compatible_range_TCC4`:

$$\{1\} \quad \forall (p_1, p_2: \text{Cpp_Subtype}(\text{pointer?})): \text{Cpp_Type?}(\text{typ}(p_2))$$

Repeatedly Skolemizing and flattening,
 Keeping (-3 1) and hiding *,
 Expanding the definition of `Cpp_Type?`,
 Repeatedly Skolemizing and flattening,
 Instantiating quantified variables,
 Expanding the definition of subterm,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 This completes the proof of `pointers_to_layout_compatible_range_TCC4`.
 Q.E.D.

C.60.6 Cpp_pointer.pointers_to_layout_compatible_dt_TCC1

Terse proof for `pointers_to_layout_compatible_dt_TCC1`.
`pointers_to_layout_compatible_dt_TCC1`:

$$\{1\} \quad \forall (p_1, p_2: \text{Cpp_Subtype}(\text{pointer?})): \\
\text{layout_compatible?}(\text{typ}(p_1), \text{typ}(p_2)) \supset \\
(\forall (x: \text{lift}[\text{Pointer_Base_Type}]): \text{range_pointer}(p_2)(x) \equiv \text{range_pointer}(p_1)(x))$$

Repeatedly Skolemizing and flattening,
 Using lemma `pointers_to_layout_compatible_range`,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 This completes the proof of `pointers_to_layout_compatible_dt_TCC1`.
 Q.E.D.

C.60.7 Cpp_pointer.void_pointer_char_pointer_dt_TCC1

Terse proof for void_pointer_char_pointer_dt_TCC1.

void_pointer_char_pointer_dt_TCC1:

$$\frac{\{1\} \quad \forall (p_1, p_2: \text{Cpp_Subtype}(\text{pointer?})): \quad \text{void?}(\text{typ}(p_1)) \wedge \text{char?}(\text{typ}(p_2)) \supset (\forall (x: \text{lift}[\text{Pointer_Base_Type}]): \text{range_pointer}(p_2)(x) \equiv \text{range_pointer}(p_1)(x))}{}$$

Repeatedly Skolemizing and flattening,

Using lemma void_pointer_char_pointer_range,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of void_pointer_char_pointer_dt_TCC1.

Q.E.D.

C.60.8 Cpp_pointer.add_TCC1

Terse proof for add_TCC1.

add_TCC1:

$$\frac{\{1\} \quad \exists (x: \quad \begin{array}{l} [\text{ptyp}: \text{Cpp_Subtype}(\text{cv}(\text{pointer?})) \rightarrow \\ \quad [[(\text{range_pointer}(\text{cv_base}(\text{ptyp}))), \text{int}] \rightarrow \\ \quad \quad (\text{range_pointer}(\text{cv_base}(\text{ptyp})))]]): \\ \text{TRUE} \end{array}}{}$$

Instantiating the top quantifier in 1 with the terms: $(\lambda (p: \text{Cpp_Subtype}(\text{cv}(\text{pointer?}))) : \lambda (y: (\text{range_pointer}(\text{cv_base}(p))), z: \text{int}): y)$,

which is trivially true.

This completes the proof of add_TCC1.

Q.E.D.

C.60.9 Cpp_pointer.add_spec2_TCC1

Terse proof for add_spec2_TCC1.

add_spec2_TCC1:

$$\frac{\{1\} \quad \forall (i: \text{int}, \text{ptyp}: \text{Cpp_Subtype}(\text{cv}(\text{pointer?})), \text{ptr_val}: (\text{range_pointer}(\text{cv_base}(\text{ptyp})))): \quad \begin{array}{l} \text{not_null?}(\text{ptyp})(\text{ptr_val}) \supset \\ \quad \text{up?}[\text{Memory_Address}](\text{address_of}(\text{ptyp})(\text{add}(\text{ptyp})(\text{ptr_val}, i))) \end{array}}{}$$

Repeatedly Skolemizing and flattening,

Using lemma add_spec,

Using lemma address_of_spec,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of add_spec2_TCC1.

Q.E.D.

C.60.10 Cpp_pointer.add_spec2_TCC2

Terse proof for add_spec2_TCC2.

add_spec2_TCC2:

$\{1\} \quad \forall (\text{ptyp}: \text{Cpp_Subtype}(\text{cv}(\text{pointer?})), \text{ptr_val}: (\text{range_pointer}(\text{cv_base}(\text{ptyp})))):$ $\text{not_null?}(\text{ptyp})(\text{ptr_val}) \supset \text{up?}[\text{Memory_Address}](\text{address_of}(\text{ptyp})(\text{ptr_val}))$
--

Repeatedly Skolemizing and flattening,

Using lemma address_of_spec,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of add_spec2_TCC2.

Q.E.D.

C.60.11 Cpp_pointer.add_simplification

Terse proof for add_simplification.

add_simplification:

$\{1\} \quad \forall (\text{ptyp}: \text{Cpp_Subtype}(\text{cv}(\text{pointer?})), \text{ptr_val}: (\text{range_pointer}(\text{cv_base}(\text{ptyp}))),$ $i, j: \text{int}):$ $\text{address_of}(\text{ptyp})(\text{add}(\text{ptyp})(\text{add}(\text{ptyp})(\text{ptr_val}, i), j)) =$ $\text{address_of}(\text{ptyp})(\text{add}(\text{ptyp})(\text{ptr_val}, i + j))$

Repeatedly Skolemizing and flattening,

Case splitting on not_null?(ptyp!1)(ptr_val!1),

we get 2 subgoals:

add_simplification.1:

$\{-1\} \quad \text{not_null?}(\text{ptyp}')(\text{ptr_val}')$ $\{-2\} \quad \text{Cpp_Type?}(\text{ptyp}')$ $\{-3\} \quad \text{cv}(\text{pointer?})(\text{ptyp}')$ $\{-4\} \quad \text{range_pointer}(\text{cv_base}(\text{ptyp}'))(\text{ptr_val}')$ $\{-5\} \quad \text{integer_pred}(i')$ $\{-6\} \quad \text{integer_pred}(j')$
$\{1\} \quad \text{address_of}(\text{ptyp}')(\text{add}(\text{ptyp}')(\text{add}(\text{ptyp}')(\text{ptr_val}', i'), j')) =$ $\text{address_of}(\text{ptyp}')(\text{add}(\text{ptyp}')(\text{ptr_val}', i' + j'))$

Using lemma add_spec,

Using lemma add_spec,

Using lemma add_spec,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Using lemma add_spec2,

Using lemma add_spec2,

Using lemma add_spec2,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Replacing using formula -5,

Case splitting on down(address_of(ptyp!1)(ptr_val!1)) + i!1 + j!1 = down(address_of(ptyp!1)(ptr_val!1)) + (i!1 + j!1),

we get 2 subgoals:

add_simplification.1.1:

{-1}	$\text{down}(\text{address_of}(\text{ptyp}')(\text{ptr_val}')) + i' + j' =$ $\text{down}(\text{address_of}(\text{ptyp}')(\text{ptr_val}')) + (i' + j')$
{-2}	$\text{not_null?}(\text{ptyp}')(\text{ptr_val}')$
{-3}	$\text{down}(\text{address_of}(\text{ptyp}')(\text{add}(\text{ptyp}')(\text{ptr_val}', i' + j')) =$ $\text{down}(\text{address_of}(\text{ptyp}')(\text{ptr_val}')) + i' + j'$
{-4}	$\text{not_null?}(\text{ptyp}')(\text{add}(\text{ptyp}')(\text{ptr_val}', i'))$
{-5}	$\text{down}(\text{address_of}(\text{ptyp}')(\text{add}(\text{ptyp}')(\text{add}(\text{ptyp}')(\text{ptr_val}', i'), j))) =$ $\text{down}(\text{address_of}(\text{ptyp}')(\text{ptr_val}')) + i' + j'$
{-6}	$\text{down}(\text{address_of}(\text{ptyp}')(\text{add}(\text{ptyp}')(\text{ptr_val}', i))) =$ $\text{down}(\text{address_of}(\text{ptyp}')(\text{ptr_val}')) + i'$
{-7}	$\text{not_null?}(\text{ptyp}')(\text{add}(\text{ptyp}')(\text{ptr_val}', i' + j'))$
{-8}	$\text{not_null?}(\text{ptyp}')(\text{add}(\text{ptyp}')(\text{add}(\text{ptyp}')(\text{ptr_val}', i'), j'))$
{-9}	$\text{Cpp_Type?}(\text{ptyp}')$
{-10}	$\text{cv}(\text{pointer?})(\text{ptyp}')$
{-11}	$\text{range_pointer}(\text{cv_base}(\text{ptyp}'))(\text{ptr_val}')$
{-12}	$\text{integer_pred}(i')$
{-13}	$\text{integer_pred}(j')$
{1}	$\text{address_of}(\text{ptyp}')(\text{add}(\text{ptyp}')(\text{add}(\text{ptyp}')(\text{ptr_val}', i'), j')) =$ $\text{address_of}(\text{ptyp}')(\text{add}(\text{ptyp}')(\text{ptr_val}', i' + j'))$

Replacing using formula -1,

Replacing using formula -3,

Keeping (-5 -7 -8 1) and hiding *,

Using lemma address_of_spec,

Using lemma address_of_spec,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Hiding formulas: (-1 -3),

Rewriting using lift_up_extensionality, matching in *,

This completes the proof of add_simplification.1.1.

add_simplification.1.2:

{-1}	$\text{not_null?}(\text{ptyp}')(\text{ptr_val}')$
{-2}	$\text{down}(\text{address_of}(\text{ptyp}')(\text{add}(\text{ptyp}')(\text{ptr_val}', i' + j))) =$ $\text{down}(\text{address_of}(\text{ptyp}')(\text{ptr_val}')) + i' + j'$
{-3}	$\text{not_null?}(\text{ptyp}')(\text{add}(\text{ptyp}')(\text{ptr_val}', i'))$
{-4}	$\text{down}(\text{address_of}(\text{ptyp}')(\text{add}(\text{ptyp}')(\text{add}(\text{ptyp}')(\text{ptr_val}', i'), j))) =$ $\text{down}(\text{address_of}(\text{ptyp}')(\text{ptr_val}')) + i' + j'$
{-5}	$\text{down}(\text{address_of}(\text{ptyp}')(\text{add}(\text{ptyp}')(\text{ptr_val}', i))) =$ $\text{down}(\text{address_of}(\text{ptyp}')(\text{ptr_val}')) + i'$
{-6}	$\text{not_null?}(\text{ptyp}')(\text{add}(\text{ptyp}')(\text{ptr_val}', i' + j'))$
{-7}	$\text{not_null?}(\text{ptyp}')(\text{add}(\text{ptyp}')(\text{add}(\text{ptyp}')(\text{ptr_val}', i'), j'))$
{-8}	$\text{Cpp_Type?}(\text{ptyp}')$
{-9}	$\text{cv}(\text{pointer?})(\text{ptyp}')$
{-10}	$\text{range_pointer}(\text{cv_base}(\text{ptyp}'))(\text{ptr_val}')$
{-11}	$\text{integer_pred}(i')$
{-12}	$\text{integer_pred}(j')$
{1}	$\text{down}(\text{address_of}(\text{ptyp}')(\text{ptr_val}')) + i' + j' =$ $\text{down}(\text{address_of}(\text{ptyp}')(\text{ptr_val}')) + (i' + j')$
{2}	$\text{address_of}(\text{ptyp}')(\text{add}(\text{ptyp}')(\text{add}(\text{ptyp}')(\text{ptr_val}', i'), j)) =$ $\text{address_of}(\text{ptyp}')(\text{add}(\text{ptyp}')(\text{ptr_val}', i' + j'))$

Keeping (1) and hiding *,

Expanding the definition of +,

C Proof scripts

which is trivially true.

This completes the proof of `add_simplification.1.2`.

`add_simplification.2`:

{-1}	<code>Cpp_Type?(ptyp')</code>
{-2}	<code>cv(pointer?)(ptyp')</code>
{-3}	<code>range_pointer(cv_base(ptyp'))(ptr_val')</code>
{-4}	<code>integer_pred(i')</code>
{-5}	<code>integer_pred(j')</code>
{1}	<code>not_null?(ptyp')(ptr_val')</code>
{2}	<code>address_of(ptyp')(add(ptyp')(add(ptyp')(ptr_val', i'), j')) = address_of(ptyp')(add(ptyp')(ptr_val', i' + j'))</code>

Using lemma `add_spec`,

Using lemma `add_spec`,

Using lemma `add_spec`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Expanding the definition of `not_null?`,

Using lemma `address_of_spec2`,

Using lemma `address_of_spec2`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `add_simplification.2`.

Q.E.D.

C.60.12 Cpp_pointer.void_pointer_other_pointer_value_TCC1

Terse proof for `void_pointer_other_pointer_value_TCC1`.

`void_pointer_other_pointer_value_TCC1`:

{1}	$\forall (p_1, p_2: \text{Cpp_Subtype}(\text{cv}(\text{pointer?})), \text{ptr_val1}: (\text{range_pointer}(\text{cv_base}(p_1)))):$ <code>array?(cv_base(p₁))</code> \vee <code>pointer?(cv_base(p₁))</code> \vee <code>reference?(cv_base(p₁))</code> \vee <code>bitfield?(cv_base(p₁))</code> \vee <code>enum?(cv_base(p₁))</code> \vee <code>pointer_to_member?(cv_base(p₁))</code> \vee <code>const?(cv_base(p₁))</code> \vee <code>volatile?(cv_base(p₁))</code>
-----	---

Repeatedly Skolemizing and flattening,

Keeping (-2 2) and hiding *,

Rewriting using `cv_base_result`, matching in *,

Keeping (1) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `void_pointer_other_pointer_value_TCC1`.

Q.E.D.

C.60.13 Cpp_pointer.void_pointer_other_pointer_value_TCC2

Terse proof for `void_pointer_other_pointer_value_TCC2`.

void_pointer_other_pointer_value_TCC2:

$$\frac{\{1\} \quad \forall (p_1, p_2: \text{Cpp_Subtype}(\text{cv}(\text{pointer?})), \text{ptr_val1}: (\text{range_pointer}(\text{cv_base}(p_1)))): \quad \text{void?}(\text{typ}(\text{cv_base}(p_1))) \wedge \text{not_null?}(p_1)(\text{ptr_val1}) \supset (\forall (\text{ptr_val2}: (\text{range_pointer}(\text{cv_base}(p_2)))): \text{not_null?}(p_2)(\text{ptr_val2}) \supset \text{up?}[\text{Memory_Address}] (\text{address_of}(p_2)(\text{ptr_val2}))}}{\quad}$$

Repeatedly Skolemizing and flattening,
 Rewriting using address_of_spec, matching in *,
 This completes the proof of void_pointer_other_pointer_value_TCC2.
 Q.E.D.

C.61 Proofs for Cpp_pointer_to_member (types.pvs)

C.61.1 Cpp_pointer_to_member.dt_ptm_TCC1

Terse proof for dt_ptm_TCC1.

dt_ptm_TCC1:

$$\frac{\{1\} \quad \exists (x_1: \quad [p: \text{Cpp_Subtype}(\text{pointer_to_member?}) \rightarrow (\text{pod_data_type?}[\text{((range_ptm}(p))])])]: \quad \text{TRUE}}{\quad}$$

Instantiating the top quantifier in 1 with the terms: $(\lambda (p: \text{Cpp_Subtype}(\text{pointer_to_member?})): \text{choose}(\text{pod_data_type?}[\text{((range_ptm}(p)))]))$,

we get 2 subgoals:

dt_ptm_TCC1.1:

$$\frac{\{1\} \quad \text{TRUE}}{\quad}$$

which is trivially true.

This completes the proof of dt_ptm_TCC1.1.

dt_ptm_TCC1.2:

$$\frac{\{1\} \quad \forall (p: \text{Cpp_Subtype}(\text{pointer_to_member?})): \quad \text{nonempty?}[\text{Interpreted_data_type}[\text{((range_ptm}(p)))]] (\text{pod_data_type?}[\text{((range_ptm}(p)))])}{\quad}$$

Repeatedly Skolemizing and flattening,
 Expanding the definition of nonempty?,
 Expanding the definition of empty?,
 Expanding the definition of member,
 Using lemma dt_pointer_to_member_exists,
 Repeatedly Skolemizing and flattening,
 Instantiating quantified variables,
 This completes the proof of dt_ptm_TCC1.2.
 Q.E.D.

C.62 Proofs for Cpp_reference (types.pvs)

C.62.1 Cpp_reference.dt_reference_TCC1

Terse proof for dt_reference_TCC1.

dt_reference_TCC1:

$$\frac{}{\{1\} \quad \exists (x_1: [r: \text{Cpp_Subtype}(\text{reference?}) \rightarrow (\text{pod_data_type?}[\text{((alloc_range(r))])])]): \text{TRUE}}$$

Using lemma dt_ref_exists,

Instantiating the top quantifier in 1 with the terms: (LAMBDA (r: Cpp_Subtype(reference?): choose(pod_data_type?(((alloc_range(r)))))),

we get 2 subgoals:

dt_reference_TCC1.1:

$$\frac{\{1\} \quad \forall (r: \text{Cpp_Subtype}(\text{reference?})): \exists (dt: (\text{pod_data_type?}[\text{((alloc_range(r))])}): \text{TRUE}}{\{1\} \quad \text{TRUE}}$$

which is trivially true.

This completes the proof of dt_reference_TCC1.1.

dt_reference_TCC1.2:

$$\frac{\{1\} \quad \forall (r: \text{Cpp_Subtype}(\text{reference?})): \exists (dt: (\text{pod_data_type?}[\text{((alloc_range(r))])}): \text{TRUE}}{\{1\} \quad \forall (r: \text{Cpp_Subtype}(\text{reference?})): \text{nonempty?}[\text{Interpreted_data_type}[\text{((alloc_range(r)))]}](\text{pod_data_type?}[\text{((alloc_range(r)))])}$$

Repeatedly Skolemizing and flattening,

Instantiating quantified variables,

Expanding the definition of nonempty?,

Expanding the definition of empty?,

Repeatedly Skolemizing and flattening,

Instantiating quantified variables,

Expanding the definition of member,

which is trivially true.

This completes the proof of dt_reference_TCC1.2.

Q.E.D.

C.63 Proofs for Cpp_schar (types.pvs)

C.63.1 Cpp_schar.range_schar_TCC1

Terse proof for range_schar_TCC1.

range_schar_TCC1:

$$\frac{}{\{1\} \quad \exists (x: \text{non_empty_finite_set}[\text{int}]): \text{TRUE}}$$

Instantiating the top quantifier in 1 with the terms: ($\lambda (n: \text{int}): n = 0$),

we get 2 subgoals:

range_schar_TCC1.1:

{1} TRUE

which is trivially true.

This completes the proof of range_schar_TCC1.1.

range_schar_TCC1.2:

{1} is_finite[int]($\lambda (n: \text{int}): n = 0$) \wedge \neg empty?[int]($\lambda (n: \text{int}): n = 0$)

Trying repeated skolemization, instantiation, and if-lifting,

Instantiating the top quantifier in 1 with the terms: (1 LAMBDA (P: (LAMBDA (n: int): n = 0)): 0),

which is trivially true.

This completes the proof of range_schar_TCC1.2.

Q.E.D.

C.63.2 Cpp_schar.dt_schar_TCC1

Terse proof for dt_schar_TCC1.

dt_schar_TCC1:

{1} $\exists (x_1: (\text{pod_data_type?}[(\text{range_schar})])): \text{TRUE}$

Rewriting using dt_schar_exists, matching in *,

This completes the proof of dt_schar_TCC1.

Q.E.D.

C.63.3 Cpp_schar.schar_value_eq_uchar_value_TCC1

Terse proof for schar_value_eq_uchar_value_TCC1.

schar_value_eq_uchar_value_TCC1:

{1} $\forall (i: (\text{range_schar})): i \geq 0 \supset (\text{range_uchar})(i)$

Repeatedly Skolemizing and flattening,

Rewriting using schar_nonneg_is_uchar, matching in *,

This completes the proof of schar_value_eq_uchar_value_TCC1.

Q.E.D.

C.63.4 Cpp_schar.value_bitmask_schar_TCC1

Terse proof for value_bitmask_schar_TCC1.

value_bitmask_schar_TCC1:

{1} $\exists (x_1: \{l: \text{list}[\text{Byte}] \mid \text{length}[\text{Byte}](l) = \text{size}(\text{uidt}(\text{dt_schar}))\}): \text{TRUE}$

Using lemma list_all_length[Byte],

Instantiating the top quantifier in -1 with the terms: (size(uidt(dt_schar))),

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of value_bitmask_schar_TCC1.

Q.E.D.

C.64 Proofs for Cpp_short (types.pvs)

C.64.1 Cpp_short.range_short_TCC1

Terse proof for range_short_TCC1.

range_short_TCC1:

{1} $\exists (x: \text{non_empty_finite_set}[\text{int}]): \text{TRUE}$

Instantiating the top quantifier in 1 with the terms: $(\lambda (n: \text{int}): n = 0)$,
we get 2 subgoals:

range_short_TCC1.1:

{1} TRUE

which is trivially true.

This completes the proof of range_short_TCC1.1.

range_short_TCC1.2:

{1} $\text{is_finite}[\text{int}](\lambda (n: \text{int}): n = 0) \wedge \neg \text{empty?}[\text{int}](\lambda (n: \text{int}): n = 0)$

Trying repeated skolemization, instantiation, and if-lifting,

Instantiating the top quantifier in 1 with the terms: $(1 \text{ LAMBDA } (P: (\text{LAMBDA } (n: \text{int}): n = 0)):$
0),

which is trivially true.

This completes the proof of range_short_TCC1.2.

Q.E.D.

C.64.2 Cpp_short.dt_short_TCC1

Terse proof for dt_short_TCC1.

dt_short_TCC1:

{1} $\exists (x_1: (\text{pod_data_type?}[(\text{range_short})])): \text{TRUE}$

Rewriting using dt_short_exists, matching in *,

This completes the proof of dt_short_TCC1.

Q.E.D.

C.64.3 Cpp_short.value_bitmask_short_TCC1

Terse proof for value_bitmask_short_TCC1.

value_bitmask_short_TCC1:

{1} $\exists (x_1: \{l: \text{list}[\text{Byte}] \mid \text{length}[\text{Byte}](l) = \text{size}(\text{uidt}(\text{dt_short}))\}): \text{TRUE}$

Using lemma list_all_length[Byte],

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `value_bitmask_short_TCC1`.
Q.E.D.

C.65 Proofs for Cpp_uchar (types.pvs)

C.65.1 Cpp_uchar.range_uchar_TCC1

Terse proof for `range_uchar_TCC1`.

`range_uchar_TCC1`:

$\{1\} \quad \exists (x: \text{non_empty_finite_set}[\text{nat}]): \text{TRUE}$

Instantiating the top quantifier in 1 with the terms: $(\lambda (n: \text{nat}): n = 0)$,
we get 2 subgoals:

`range_uchar_TCC1.1`:

$\{1\} \quad \text{TRUE}$

which is trivially true.

This completes the proof of `range_uchar_TCC1.1`.

`range_uchar_TCC1.2`:

$\{1\} \quad \text{is_finite}[\text{nat}](\lambda (n: \text{nat}): n = 0) \wedge \neg \text{empty?}[\text{nat}](\lambda (n: \text{nat}): n = 0)$

Trying repeated skolemization, instantiation, and if-lifting,

Instantiating the top quantifier in 1 with the terms: $(1 \text{ LAMBDA } (P: (\text{LAMBDA } (n: \text{nat}): n = 0)):$
0),

which is trivially true.

This completes the proof of `range_uchar_TCC1.2`.

Q.E.D.

C.65.2 Cpp_uchar.dt_uchar_TCC1

Terse proof for `dt_uchar_TCC1`.

`dt_uchar_TCC1`:

$\{1\} \quad \exists (x_1: (\text{pod_data_type?}[(\text{range_uchar})])): \text{TRUE}$

Using lemma `dt_uchar_exists`,

This completes the proof of `dt_uchar_TCC1`.

Q.E.D.

C.65.3 Cpp_uchar.value_bitmask_uchar_TCC1

Terse proof for `value_bitmask_uchar_TCC1`.

`value_bitmask_uchar_TCC1`:

$\{1\} \quad \exists (x_1: \{l: \text{list}[\text{Byte}] \mid \text{length}[\text{Byte}](l) = \text{size}(\text{uidt}(\text{dt_uchar}))\}): \text{TRUE}$

Using lemma `list_all_length[Byte]`,

Instantiating the top quantifier in -1 with the terms: (size(uidt(dt_uchar))),
 Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of value_bitmask_uchar_TCC1.
 Q.E.D.

C.66 Proofs for Cpp_uint (types.pvs)

C.66.1 Cpp_uint.range_uint_TCC1

Terse proof for range_uint_TCC1.

range_uint_TCC1:

{1} $\exists (x: \text{non_empty_finite_set}[\text{nat}]): \text{TRUE}$

Instantiating the top quantifier in 1 with the terms: ($\lambda (n: \text{nat}): n = 0$),
 we get 2 subgoals:

range_uint_TCC1.1:

{1} TRUE

which is trivially true.

This completes the proof of range_uint_TCC1.1.

range_uint_TCC1.2:

{1} $\text{is_finite}[\text{nat}](\lambda (n: \text{nat}): n = 0) \wedge \neg \text{empty?}[\text{nat}](\lambda (n: \text{nat}): n = 0)$

Trying repeated skolemization, instantiation, and if-lifting,

Instantiating the top quantifier in 1 with the terms: (1 LAMBDA (p: (LAMBDA (n: nat): n = 0)):
 0),

which is trivially true.

This completes the proof of range_uint_TCC1.2.

Q.E.D.

C.66.2 Cpp_uint.dt_uint_TCC1

Terse proof for dt_uint_TCC1.

dt_uint_TCC1:

{1} $\exists (x_1: (\text{pod_data_type?}[(\text{range_uint})])): \text{TRUE}$

Rewriting using dt_uint_exists, matching in *,

This completes the proof of dt_uint_TCC1.

Q.E.D.

C.66.3 Cpp_uint.uint_representation_same_for_positive_int_TCC1

Terse proof for uint_representation_same_for_positive_int_TCC1.

uint_representation_same_for_positive_int_TCC1:

{1} $\forall (i: (\text{range_int})): i \geq 0 \supset (\text{range_uint})(i)$

Repeatedly Skolemizing and flattening,
 Rewriting using uint_value_supset_int_value, matching in *,
 This completes the proof of uint_representation_same_for_positive_int_TCC1.
 Q.E.D.

C.66.4 Cpp_uint.value_bitmask_uint_TCC1

Terse proof for value_bitmask_uint_TCC1.

value_bitmask_uint_TCC1:

$$\frac{}{\{1\} \exists (x_1: \{l: \text{list}[\text{Byte}] \mid \text{length}[\text{Byte}](l) = \text{size}(\text{uidt}(\text{dt_uint}))\}): \text{TRUE}}$$

Using lemma list_all_length[Byte],
 Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of value_bitmask_uint_TCC1.
 Q.E.D.

C.67 Proofs for Cpp_ulong (types.pvs)

C.67.1 Cpp_ulong.range_ulong_TCC1

Terse proof for range_ulong_TCC1.

range_ulong_TCC1:

$$\frac{}{\{1\} \exists (x: \text{non_empty_finite_set}[\text{nat}]): \text{TRUE}}$$

Instantiating the top quantifier in 1 with the terms: $(\lambda (n: \text{nat}): n = 0)$,
 we get 2 subgoals:

range_ulong_TCC1.1:

$$\frac{}{\{1\} \text{TRUE}}$$

which is trivially true.

This completes the proof of range_ulong_TCC1.1.

range_ulong_TCC1.2:

$$\frac{}{\{1\} \text{is_finite}[\text{nat}](\lambda (n: \text{nat}): n = 0) \wedge \neg \text{empty?}[\text{nat}](\lambda (n: \text{nat}): n = 0)}}$$

Trying repeated skolemization, instantiation, and if-lifting,

Instantiating the top quantifier in 1 with the terms: $(1 \text{ LAMBDA } (P: (\text{LAMBDA } (n: \text{nat}): n = 0)):$
 $0)$,

which is trivially true.

This completes the proof of range_ulong_TCC1.2.

Q.E.D.

C.67.2 Cpp_ulong.dt_ulong_TCC1

Terse proof for dt_ulong_TCC1.

dt_ulong_TCC1:

{1} $\exists (x_1: (\text{pod_data_type?}[(\text{range_ulong})])): \text{TRUE}$

Rewriting using dt_ulong_exists, matching in *,
 This completes the proof of dt_ulong_TCC1.
 Q.E.D.

C.67.3

Cpp_ulong.ulong_representation_same_for_positive_long_TCC1

Terse proof for ulong_representation_same_for_positive_long_TCC1.

ulong_representation_same_for_positive_long_TCC1:

{1} $\forall (i: (\text{range_long})): i \geq 0 \supset (\text{range_ulong})(i)$

Repeatedly Skolemizing and flattening,
 Rewriting using ulong_value_supset_long_value, matching in *,
 This completes the proof of ulong_representation_same_for_positive_long_TCC1.
 Q.E.D.

C.67.4 Cpp_ulong.value_bitmask_ulong_TCC1

Terse proof for value_bitmask_ulong_TCC1.

value_bitmask_ulong_TCC1:

{1} $\exists (x_1: \{l: \text{list}[\text{Byte}] \mid \text{length}[\text{Byte}](l) = \text{size}(\text{uidt}(\text{dt_ulong}))\}): \text{TRUE}$

Using lemma list_all_length[Byte],
 Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of value_bitmask_ulong_TCC1.
 Q.E.D.

C.68 Proofs for Cpp_ulonglong (types.pvs)

C.68.1 Cpp_ulonglong.range_ulonglong_TCC1

Terse proof for range_ulonglong_TCC1.

range_ulonglong_TCC1:

{1} $\exists (x: \text{non_empty_finite_set}[\text{nat}]): \text{TRUE}$

Instantiating the top quantifier in 1 with the terms: $(\lambda (n: \text{nat}): n = 0)$,
 we get 2 subgoals:

range_ulonglong_TCC1.1:

{1} TRUE

which is trivially true.
 This completes the proof of range_ulonglong_TCC1.1.

range_ulonglong_TCC1.2:

$$\frac{}{\{1\} \text{ is_finite}[\text{nat}] (\lambda (n: \text{nat}): n = 0) \wedge \neg \text{empty?}[\text{nat}] (\lambda (n: \text{nat}): n = 0)}$$

Trying repeated skolemization, instantiation, and if-lifting,

Instantiating the top quantifier in 1 with the terms: (1 LAMBDA (P: (LAMBDA (n: nat): n = 0)): 0),

which is trivially true.

This completes the proof of range_ulonglong_TCC1.2.

Q.E.D.

C.68.2 Cpp_ulonglong.dt_ulonglong_TCC1

Terse proof for dt_ulonglong_TCC1.

dt_ulonglong_TCC1:

$$\frac{}{\{1\} \exists (x_1: (\text{pod_data_type?}[(\text{range_ulonglong})])): \text{TRUE}}$$

Rewriting using dt_ulonglong_exists, matching in *,

This completes the proof of dt_ulonglong_TCC1.

Q.E.D.

C.68.3

Cpp_ulonglong.ulonglong_representation_same_for_positive_longlong_TCC1

Terse proof for ulonglong_representation_same_for_positive_longlong_TCC1.

ulonglong_representation_same_for_positive_longlong_TCC1:

$$\frac{}{\{1\} \forall (i: (\text{range_longlong})): i \geq 0 \supset (\text{range_ulonglong})(i)}$$

Repeatedly Skolemizing and flattening,

Rewriting using ulonglong_value_supset_longlong_value, matching in *,

This completes the proof of ulonglong_representation_same_for_positive_longlong_TCC1.

Q.E.D.

C.68.4 Cpp_ulonglong.value_bitmask_ulonglong_TCC1

Terse proof for value_bitmask_ulonglong_TCC1.

value_bitmask_ulonglong_TCC1:

$$\frac{}{\{1\} \exists (x_1: \{l: \text{list}[\text{Byte}] \mid \text{length}[\text{Byte}](l) = \text{size}(\text{uidt}(\text{dt_ulonglong}))\}): \text{TRUE}}$$

Using lemma list_all_length[Byte],

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of value_bitmask_ulonglong_TCC1.

Q.E.D.

C.69 Proofs for Cpp_union (types.pvs)

C.69.1 Cpp_union.uidt_union_TCC1

Terse proof for uidt_union_TCC1.

uidt_union_TCC1:

{1}	$\exists (x:$ $\quad [Cpp_Subtype(union?) \rightarrow \{uidt: (uninterpreted_data_type?) \mid size(uidt) > 0\}]):$ $\quad \text{TRUE}$
-----	---

Instantiating the top quantifier in 1 with the terms: $(\lambda (c: Cpp_Subtype(union?)): \text{choose}(\{uidt: (uninterpreted_data_type?) \mid size(uidt) > 0\}))$,

we get 2 subgoals:

uidt_union_TCC1.1:

{1}	TRUE
-----	------

which is trivially true.

This completes the proof of uidt_union_TCC1.1.

uidt_union_TCC1.2:

{1}	$\forall (c: Cpp_Subtype(union?)):$ $\quad \text{nonempty?}[(uninterpreted_data_type?)$ $\quad \quad (\{uidt: (uninterpreted_data_type?) \mid size(uidt) > 0\})$
-----	---

Trying repeated skolemization, instantiation, and if-lifting,

Instantiating the top quantifier in -3 with the terms: $((\# \text{ size} := 1, \text{ valid?} := \text{LAMBDA (l: list[Byte], a: Address): FALSE \#}))$,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of uidt_union_TCC1.2.

Q.E.D.

C.70 Proofs for Cpp_ushort (types.pvs)

C.70.1 Cpp_ushort.range_ushort_TCC1

Terse proof for range_ushort_TCC1.

range_ushort_TCC1:

{1}	$\exists (x: \text{non_empty_finite_set}[\text{nat}]): \text{TRUE}$
-----	--

Instantiating the top quantifier in 1 with the terms: $(\lambda (n: \text{nat}): n = 0)$,

we get 2 subgoals:

range_ushort_TCC1.1:

{1}	TRUE
-----	------

which is trivially true.

This completes the proof of range_ushort_TCC1.1.

range_ushort_TCC1.2:

$$\frac{}{\{1\} \text{ is_finite}[\text{nat}](\lambda (n: \text{nat}): n = 0) \wedge \neg \text{empty?}[\text{nat}](\lambda (n: \text{nat}): n = 0)}$$

Trying repeated skolemization, instantiation, and if-lifting,

Instantiating the top quantifier in 1 with the terms: (1 LAMBDA (P: (LAMBDA (n: nat): n = 0)):
0),

which is trivially true.

This completes the proof of range_ushort_TCC1.2.

Q.E.D.

C.70.2 Cpp_ushort.dt_ushort_TCC1

Terse proof for dt_ushort_TCC1.

dt_ushort_TCC1:

$$\frac{}{\{1\} \exists (x_1: (\text{pod_data_type?}[(\text{range_ushort})])): \text{TRUE}}$$

Rewriting using dt_ushort_exists, matching in *,

This completes the proof of dt_ushort_TCC1.

Q.E.D.

C.70.3

Cpp_ushort.ushort_representation_same_for_positive_short_TCC1

Terse proof for ushort_representation_same_for_positive_short_TCC1.

ushort_representation_same_for_positive_short_TCC1:

$$\frac{}{\{1\} \forall (i: (\text{range_short})): i \geq 0 \supset (\text{range_ushort})(i)}$$

Repeatedly Skolemizing and flattening,

Rewriting using ushort_value_supset_short_value, matching in *,

This completes the proof of ushort_representation_same_for_positive_short_TCC1.

Q.E.D.

C.70.4 Cpp_ushort.value_bitmask_ushort_TCC1

Terse proof for value_bitmask_ushort_TCC1.

value_bitmask_ushort_TCC1:

$$\frac{}{\{1\} \exists (x_1: \{l: \text{list}[\text{Byte}] \mid \text{length}[\text{Byte}](l) = \text{size}(\text{uidt}(\text{dt_ushort}))\}): \text{TRUE}}$$

Using lemma list_all_length[Byte],

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of value_bitmask_ushort_TCC1.

Q.E.D.

C.71 Proofs for Cpp_wchar_t (types.pvs)

C.71.1 Cpp_wchar_t.range_wchar_t_TCC1

Terse proof for range_wchar_t_TCC1.

range_wchar_t_TCC1:

{1} $\exists (x: \text{non_empty_finite_set}[\text{int}]): \text{TRUE}$

Instantiating the top quantifier in 1 with the terms: $(\lambda (n: \text{int}): n = 0)$,
we get 2 subgoals:

range_wchar_t_TCC1.1:

{1} TRUE

which is trivially true.

This completes the proof of range_wchar_t_TCC1.1.

range_wchar_t_TCC1.2:

{1} $\text{is_finite}[\text{int}](\lambda (n: \text{int}): n = 0) \wedge \neg \text{empty}[\text{int}](\lambda (n: \text{int}): n = 0)$

Trying repeated skolemization, instantiation, and if-lifting,

Instantiating the top quantifier in 1 with the terms: $(1 \text{ LAMBDA } (P: (\text{LAMBDA } (n: \text{int}): n = 0)):$
0),

which is trivially true.

This completes the proof of range_wchar_t_TCC1.2.

Q.E.D.

C.71.2 Cpp_wchar_t.dt_wchar_t_TCC1

Terse proof for dt_wchar_t_TCC1.

dt_wchar_t_TCC1:

{1} $\exists (x_1: (\text{pod_data_type}[(\text{range_wchar_t})])): \text{TRUE}$

Rewriting using dt_wchar_t_exists, matching in *,

This completes the proof of dt_wchar_t_TCC1.

Q.E.D.

C.71.3 Cpp_wchar_t.value_bitmask_wchar_t_TCC1

Terse proof for value_bitmask_wchar_t_TCC1.

value_bitmask_wchar_t_TCC1:

{1} $\exists (x_1: \{l: \text{list}[\text{Byte}] \mid \text{length}[\text{Byte}](l) = \text{size}(\text{uidt}(\text{dt_wchar_t}))\}): \text{TRUE}$

Using lemma list_all_length[Byte],

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of value_bitmask_wchar_t_TCC1.

Q.E.D.

C.72 Proofs for Datatype_Model (datatype_model.pvs)

C.72.1 Datatype_Model.uidt_1_TCC1

Terse proof for uidt_1_TCC1.

uidt_1_TCC1:

{1}	$\text{uninterpreted_data_type?}(\text{(#size := 1, valid? := } \lambda (\text{bl: list[Byte]}, \text{a: Ad- dress): bl = (:0)\#}))$
-----	--

Trying repeated skolemization, instantiation, and if-lifting,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of uidt_1_TCC1.

Q.E.D.

C.72.2 Datatype_Model.pod_1_TCC1

Terse proof for pod_1_TCC1.

pod_1_TCC1:

{1}	$0 < \text{max_byte}$
-----	------------------------

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of pod_1_TCC1.

Q.E.D.

C.72.3 Datatype_Model.pod_1_TCC2

Terse proof for pod_1_TCC2.

pod_1_TCC2:

{1}	$\text{every}_{\text{real}}(\lambda (x: \text{real}): \text{rational_pred}(x) \wedge \text{integer_pred}(x) \wedge x \geq 0 \wedge x < \text{max_byte})$ $((::))$
-----	--

Trying repeated skolemization, instantiation, and if-lifting,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of pod_1_TCC2.

Q.E.D.

C.72.4 Datatype_Model.pod_1_TCC3

Terse proof for pod_1_TCC3.

pod_1_TCC3:

{1}	$\forall (f: [[\text{Unit}, \text{Address}] \rightarrow [\text{list[Byte}] \rightarrow \text{bool}]]):$ $(f = (\lambda (d: \text{Unit}, a: \text{Address}): \text{zero_mask?}(\text{size}(\text{uidt}_1)))) \supset$ $(\forall (x_1: [\text{Unit}, \text{Address}]): \text{singleton?}[\text{list[Byte]}](f(x_1)))$
-----	--

Repeatedly Skolemizing and flattening,

Replacing using formula -1,

C Proof scripts

Using lemma `singleton_zero_mask`,
Expanding the definition of `singleton?`,
which is trivially true.
This completes the proof of `pod_1_TCC3`.
Q.E.D.

C.72.5 Datatype_Model.pod_1_TCC4

Terse proof for `pod_1_TCC4`.

`pod_1_TCC4`:

```
{1} pod_data_type?[Unit]
    ((#uidt := uidt_1,
      to_byte := λ (d: Unit, a: Address): (:0),
      to_mask
        := λ (f: [[Unit, Address] → [list[Byte] → bool]]):
            λ (x1: [Unit, Address]): singleton_elt[list[Byte]](f(x1))
            (λ (d: Unit, a: Address): zero_mask?(size(uidt_1))),
      from_byte
        := λ (bl: list[Byte], a: Address):
            IF bl = (:0) THEN up[Unit](unit) ELSE bottom[Unit] ENDIF#))
```

Expanding the definition of `pod_data_type?`,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `pod_1_TCC4`.
Q.E.D.

C.72.6 Datatype_Model.uidt_2_TCC1

Terse proof for `uidt_2_TCC1`.

`uidt_2_TCC1`:

```
{1} uninterpreted_data_type?((#size := 1,
                              valid? := λ (bl: list[Byte], a: Ad-
                              dress): bl = (:1)#))
```

Trying repeated skolemization, instantiation, and if-lifting,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `uidt_2_TCC1`.
Q.E.D.

C.72.7 Datatype_Model.pod_2_TCC1

Terse proof for `pod_2_TCC1`.

`pod_2_TCC1`:

```
{1} 1 < max_byte
```

Trying repeated skolemization, instantiation, and if-lifting,
Applying `less_than_max_byte`
Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of pod_2_TCC1.

Q.E.D.

C.72.8 Datatype_Model.pod_2_TCC2

Terse proof for pod_2_TCC2.

pod_2_TCC2:

```
{1}  ∀ (f: [[Unit, Address] → [list[Byte] → bool]]):
      (f = (λ (d: Unit, a: Address): zero_mask?(size(uidt_2)))) ⊃
      (∀ (x1: [Unit, Address]): singleton?[list[Byte]](f(x1)))
```

Repeatedly Skolemizing and flattening,

Replacing using formula -1,

Using lemma singleton_zero_mask,

Expanding the definition of singleton?,

which is trivially true.

This completes the proof of pod_2_TCC2.

Q.E.D.

C.72.9 Datatype_Model.pod_2_TCC3

Terse proof for pod_2_TCC3.

pod_2_TCC3:

```
{1}  pod_data_type?[Unit]
      ((#uidt := uidt_2,
        to_byte := λ (d: Unit, a: Address): (:1:),
        to_mask
          := λ (f: [[Unit, Address] → [list[Byte] → bool]]):
              λ (x1: [Unit, Address]): singleton_elt[list[Byte]](f(x1))
              (λ (d: Unit, a: Address): zero_mask?(size(uidt_2))),
        from_byte
          := λ (bl: list[Byte], a: Address):
              IF bl = (:1:) THEN up[Unit](unit) ELSE bottom[Unit] ENDIF#))
```

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of pod_2_TCC3.

Q.E.D.

C.73 Proofs for Datatype_Model_2 (datatype_model.pvs)

C.73.1 Datatype_Model_2.uidt_bool_TCC1

Terse proof for uidt_bool_TCC1.

uidt_bool_TCC1:

{1}	$\text{uninterpreted_data_type?}(\text{\#size := 2, valid?} \\ := \lambda (\text{bl: list[Byte]}, \text{a: Address}): \\ \text{bl} = (:0, 0) \vee \text{bl} = (:0, 1)\#)$
-----	---

Trying repeated skolemization, instantiation, and if-lifting,
we get 2 subgoals:

uidt_bool_TCC1.1:

{-1}	$\text{every}(\lambda (x: \text{number}): \\ \text{number_field_pred}(x) \wedge \\ \text{real_pred}(x) \wedge \\ \text{rational_pred}(x) \wedge \\ \text{integer_pred}(x) \wedge x \geq 0 \wedge x < \text{expt}(2, \text{bits_per_byte})) \\ ((:0, 0:))$
{-2}	$l' = (:0, 0:)$
{1}	$\text{length}((:0, 0:)) = 2$

Expanding the definition of length,
Expanding the definition of length,
Expanding the definition of length,
which is trivially true.

This completes the proof of uidt_bool_TCC1.1.

uidt_bool_TCC1.2:

{-1}	$\text{every}(\lambda (x: \text{number}): \\ \text{number_field_pred}(x) \wedge \\ \text{real_pred}(x) \wedge \\ \text{rational_pred}(x) \wedge \\ \text{integer_pred}(x) \wedge x \geq 0 \wedge x < \text{expt}(2, \text{bits_per_byte})) \\ ((:0, 1:))$
{-2}	$l' = (:0, 1:)$
{1}	$\text{length}((:0, 1:)) = 2$

Expanding the definition of length,
Expanding the definition of length,
Expanding the definition of length,
which is trivially true.

This completes the proof of uidt_bool_TCC1.2.

Q.E.D.

C.73.2 Datatype_Model_2.pod_bool_TCC1

Terse proof for pod_bool_TCC1.

pod_bool_TCC1:

{1}	$\forall (d: \text{bool}): d \supset 0 < \text{max_byte}$
-----	--

Applying less_than_max_byte

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of pod_bool_TCC1.

Q.E.D.

C.73.3 Datatype_Model_2.pod_bool_TCC2

Terse proof for pod_bool_TCC2.

pod_bool_TCC2:

{1}	$\forall (d: \text{bool}):$ $d \supset$ $\text{every}[\text{real}]$ $(\lambda (x: \text{real}): \text{rational_pred}(x) \wedge \text{integer_pred}(x) \wedge x \geq 0 \wedge x < \text{max_byte})$ $((:1:))$
-----	---

Trying repeated skolemization, instantiation, and if-lifting,

Applying less_than_max_byte

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of pod_bool_TCC2.

Q.E.D.

C.73.4 Datatype_Model_2.pod_bool_TCC3

Terse proof for pod_bool_TCC3.

pod_bool_TCC3:

{1}	$\forall (d: \text{bool}): \neg d \supset 0 < \text{max_byte}$
-----	---

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of pod_bool_TCC3.

Q.E.D.

C.73.5 Datatype_Model_2.pod_bool_TCC4

Terse proof for pod_bool_TCC4.

pod_bool_TCC4:

{1}	$\forall (d: \text{bool}):$ $\neg d \supset$ $\text{every}[\text{real}]$ $(\lambda (x: \text{real}): \text{rational_pred}(x) \wedge \text{integer_pred}(x) \wedge x \geq 0 \wedge x < \text{max_byte})$ $((:0:))$
-----	--

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of pod_bool_TCC4.

Q.E.D.

C.73.6 Datatype_Model_2.pod_bool_TCC5

Terse proof for pod_bool_TCC5.

pod_bool_TCC5:

{1}	$\forall (f: [[\text{bool}, \text{Address}] \rightarrow [\text{list}[\text{Byte}] \rightarrow \text{bool}]]):$ $(f = (\lambda (d: \text{bool}, a: \text{Address}): \text{zero_mask}?(size(uidt_bool)))) \supset$ $(\forall (x_1: [\text{bool}, \text{Address}]): \text{singleton}?[\text{list}[\text{Byte}]](f(x_1)))$
-----	--

C Proof scripts

Repeatedly Skolemizing and flattening,
Replacing using formula -1,
Using lemma singleton_zero_mask,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of pod_bool_TCC5.
Q.E.D.

C.73.7 Datatype_Model_2.pod_bool_TCC6

Terse proof for pod_bool_TCC6.

pod_bool_TCC6:

```
{1} pod_data_type?[bool]
    ((#uidt := uidt_bool,
     to_byte := λ (d: bool, a: Address): IF d THEN (:0, 1:) ELSE (:0, 0:) EN-
DIF,
     to_mask
     := λ (f: [[bool, Address] → [list[Byte] → bool]]):
         λ (x₁: [bool, Address]): singleton_elt[list[Byte]](f(x₁))
         (λ (d: bool, a: Address): zero_mask?(size(uidt_bool))),
     from_byte
     := λ (bl: list[Byte], a: Address):
         IF bl = (:0, 1:)
         THEN up[boolean](TRUE)
         ELSE IF bl = (:0, 0:) THEN up[boolean](FALSE) ELSE bot-
tom[boolean] ENDIF
     ENDIF#))
```

Trying repeated skolemization, instantiation, and if-lifting,
Using lemma min_bits_per_byte,
Using lemma bits_per_byte_minimum,
Using lemma pos_expt_gt,
Case splitting on $1 < \text{expt}(2, \text{bits_per_byte})$,
we get 2 subgoals:
pod_bool_TCC6.1:

```
{-1} 1 < expt(2, bits_per_byte)
{-2} bits_per_byte < 2 ^ bits_per_byte
{-3} bits_per_byte ≥ min_bits_per_byte
{-4} min_bits_per_byte = 8
{-5} (:0, 0:) = (:0, 1:)
{1} d'
```

Applying decompose-equality,
Applying decompose-equality,
This completes the proof of pod_bool_TCC6.1.

pod_bool_TCC6.2:

{-1}	bits_per_byte < 2 ^ bits_per_byte
{-2}	bits_per_byte ≥ min_bits_per_byte
{-3}	min_bits_per_byte = 8
{-4}	(:0, 0:) = (:0, 1:)
{1}	1 < expt(2, bits_per_byte)
{2}	d'

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of pod_bool_TCC6.2.
Q.E.D.

C.73.8 Datatype_Model_2.uidt_Byte_TCC1

Terse proof for uidt_Byte_TCC1.

uidt_Byte_TCC1:

{1}	∃ (bl: list[Byte]): length(bl) = 2 ⊃ cons?[Byte](bl)
-----	--

Trying repeated skolemization, instantiation, and if-lifting,
Expanding the definition of length,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of uidt_Byte_TCC1.
Q.E.D.

C.73.9 Datatype_Model_2.uidt_Byte_TCC2

Terse proof for uidt_Byte_TCC2.

uidt_Byte_TCC2:

{1}	uninterpreted_data_type?((#size := 2, valid? := λ (bl: list[Byte], a: Address): length[Byte](bl) = 2 ∧ car[Byte](bl) = 1#))
-----	--

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of uidt_Byte_TCC2.
Q.E.D.

C.73.10 Datatype_Model_2.pod_Byte_TCC1

Terse proof for pod_Byte_TCC1.

pod_Byte_TCC1:

{1}	∃ (bl: list[Byte]): length(bl) = 2 ∧ car(bl) = 1 ⊃ cons?[Byte](cdr[Byte](bl))
-----	---

Trying repeated skolemization, instantiation, and if-lifting,
Expanding the definition of length,
Expanding the definition of length,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of pod_Byte_TCC1.

Q.E.D.

C.73.11 Datatype_Model_2.pod_Byte_TCC2

Terse proof for pod_Byte_TCC2.

pod_Byte_TCC2:

{1} 1 < max_byte

Trying repeated skolemization, instantiation, and if-lifting,
 Applying less_than_max_byte
 Expanding the definition of max_byte,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of pod_Byte_TCC2.
 Q.E.D.

C.73.12 Datatype_Model_2.pod_Byte_TCC3

Terse proof for pod_Byte_TCC3.

pod_Byte_TCC3:

{1} $\forall (d: \text{Byte}):$
 every_[real]
 $(\lambda (x: \text{real}): \text{rational_pred}(x) \wedge \text{integer_pred}(x) \wedge x \geq 0 \wedge x < \text{max_byte})$
 $((:d:))$

Repeatedly Skolemizing and flattening,
 Expanding the definition of every,
 Expanding the definition of every,
 which is trivially true.
 This completes the proof of pod_Byte_TCC3.
 Q.E.D.

C.73.13 Datatype_Model_2.pod_Byte_TCC4

Terse proof for pod_Byte_TCC4.

pod_Byte_TCC4:

{1} $\forall (f: [\text{Byte}, \text{Address}] \rightarrow [\text{list}[\text{Byte}] \rightarrow \text{bool}]):$
 $(f = (\lambda (d: \text{Byte}, a: \text{Address}): \text{zero_mask}?(size(uidt_Byte)))) \supset$
 $(\forall (x_1: [\text{Byte}, \text{Address}]): \text{singleton}?(list[\text{Byte}])(f(x_1)))$

Repeatedly Skolemizing and flattening,
 Replacing using formula -1,
 Using lemma singleton_zero_mask,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 This completes the proof of pod_Byte_TCC4.
 Q.E.D.

C.73.14 Datatype_Model_2.pod_Byte_TCC5

Terse proof for pod_Byte_TCC5.

pod_Byte_TCC5:

```

{1} pod_data_type?[Byte]
  ((#uidt := uidt_Byte,
    to_byte := λ (d: Byte, a: Address): (:1, d),
    to_mask
      := λ (f: [[Byte, Address] → [list[Byte] → bool]]):
        λ (x1: [Byte, Address]): singleton_elt[list[Byte]](f(x1))
        (λ (d: Byte, a: Address): zero_mask?(size(uidt_Byte))),
    from_byte
      := λ (bl: list[Byte], a: Address):
        IF length[Byte](bl) = 2 ∧ car[Byte](bl) = 1
          THEN up[Byte](car[Byte](cdr[Byte](bl)))
          ELSE bottom[Byte]
        ENDIF#))

```

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of pod_Byte_TCC5.

Q.E.D.

C.74 Proofs for DeclarationStatements1 (statements.pvs)

This theory contains no provable formal statements.

C.75 Proofs for DeclarationStatements2 (statements.pvs)

This theory contains no provable formal statements.

C.76 Proofs for Device_Memory (device_memory.pvs)

This theory contains no provable formal statements.

C.77 Proofs for EqualityExpressions (expressions.pvs)

This theory contains no provable formal statements.

C.78 Proofs for Error_Code_Types (result.pvs)

This theory contains no provable formal statements.

C.79 Proofs for Even_More_List_Props (vfiasco-prelude.pvs)

C.79.1 Even_More_List_Props.every_extend

Terse proof for `every_extend`.

`every_extend:`

$$\frac{\{1\} \quad \forall (l: \text{list}[S], P: \text{PRED}[T]): \text{every}(\lambda (t: T): P(t))(l) \supset \text{every}(\lambda (s: S): P(s))(l)}{\quad}$$

Inducting on l on formula 1,

we get 2 subgoals:

`every_extend.1:`

$$\frac{\{1\} \quad \forall (P: \text{PRED}[T]): \text{every}(\lambda (t: T): P(t))(\text{null}) \supset \text{every}(\lambda (s: S): P(s))(\text{null})}{\quad}$$

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `every_extend.1`.

`every_extend.2:`

$$\frac{\{1\} \quad \forall (\text{cons1_var}: S, \text{cons2_var}: \text{list}[S]): (\forall (P: \text{PRED}[T]): \text{every}(\lambda (t: T): P(t))(\text{cons2_var}) \supset \text{every}(\lambda (s: S): P(s))(\text{cons2_var})) \supset (\forall (P: \text{PRED}[T]): \text{every}(\lambda (t: T): P(t))(\text{cons}(\text{cons1_var}, \text{cons2_var})) \supset \text{every}(\lambda (s: S): P(s))(\text{cons}(\text{cons1_var}, \text{cons2_var})))}{\quad}$$

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `every_extend.2`.

Q.E.D.

C.80 Proofs for Exception_type (result.pvs)

This theory contains no provable formal statements.

C.81 Proofs for Exception_type_adt (Exception_type_adt.pvs)

This theory contains no provable formal statements.

C.82 Proofs for *Exception_type_adt_reduce* (*Exception_type_adt.pvs*)

This theory contains no provable formal statements.

C.83 Proofs for *Expand_State* (*device_memory.pvs*)

C.83.1 *Expand_State.em_transformers_ok*

Terse proof for *em_transformers_ok*.

em_transformers_ok:

$\{1\} \quad \forall (\text{states: PRED}[\text{State}], \text{transformers: PRED}[[\text{State} \rightarrow \text{SuperResult}[\text{State}]]]):$ $\text{transformers_ok?}(\text{states}, \text{transformers}) \supset$ $\text{transformers_ok?}(\text{em_lift}(\text{states}), \text{em_lift}(\text{transformers}))$
--

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of *em_transformers_ok*.

Q.E.D.

C.83.2 *Expand_State.em_transformer_invariant*

Terse proof for *em_transformer_invariant*.

em_transformer_invariant:

$\{1\} \quad \forall (\text{states: PRED}[\text{State}], \text{transformers: PRED}[[\text{State} \rightarrow \text{SuperResult}[\text{State}]]]):$ $\text{transformer_invariant?}(\text{states}, \text{transformers}) \supset$ $\text{transformer_invariant?}(\text{em_lift}(\text{states}), \text{em_lift}(\text{transformers}))$
--

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of *em_transformer_invariant*.

Q.E.D.

C.83.3 *Expand_State.em_lift_singleton*

Terse proof for *em_lift_singleton*.

em_lift_singleton:

$\{1\} \quad \forall (q: [\text{State} \rightarrow \text{SuperResult}[\text{State}]]):$ $\text{singleton}(\text{em_lift}(q)) = \text{em_lift}(\text{singleton}(q))$

Repeatedly Skolemizing and flattening,

Applying decompose-equality,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of *em_lift_singleton*.

Q.E.D.

C.84 Proofs for Expand_State2 (device_memory.pvs)

C.84.1 Expand_State2.em_lift_expr_2_super

Terse proof for em_lift_expr_2_super.

em_lift_expr_2_super:

$$\{1\} \quad \forall (\text{stf}: [\text{State} \rightarrow \text{ExprResult}[\text{State}, \text{Data}]]):$$

$$\text{expr_2_super}(\text{em_lift}(\text{stf})) = \text{em_lift}(\text{expr_2_super}(\text{stf}))$$

Repeatedly Skolemizing and flattening,
 Applying decompose-equality,
 Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of em_lift_expr_2_super.
 Q.E.D.

C.85 Proofs for Expand_State3 (device_memory.pvs)

C.85.1 Expand_State3.em_memory_read_transformers

Terse proof for em_memory_read_transformers.

em_memory_read_transformers:

$$\{1\} \quad \forall (\text{addresses}: \text{PRED}[\text{Address}], \text{em_pm}: \text{Memory_struct}[\text{Expand_state}[\text{State}, T]],$$

$$\text{pm}: \text{Memory_struct}[\text{State}]):$$

$$(\text{em_pm}'\text{memory_read} =$$

$$(\lambda (a: \text{Address}): \text{em_lift}[\text{State}, T, \text{Byte}](\text{pm}'\text{memory_read}(a))))$$

$$\supset$$

$$\text{memory_read_transformers}(\text{em_pm}, \text{addresses}) =$$

$$\text{em_lift}(\text{memory_read_transformers}(\text{pm}, \text{addresses}))$$

Repeatedly Skolemizing and flattening,
 Applying decompose-equality,
 Expanding the definition of memory_read_transformers,
 Replacing using formula -1,
 Hiding formulas: -1,
 Expanding the definition of em_lift,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 we get 2 subgoals:

em_memory_read_transformers.1:

$$\{-1\} \quad \exists (a: \text{Address}):$$

$$\text{addresses}'(a) \wedge x' = \text{expr_2_super}(\text{em_lift}[\text{State}, T, \text{Byte}](\text{pm}'\text{memory_read}(a)))$$

$$\{1\} \quad \exists (p:$$

$$(\lambda (q: [\text{State} \rightarrow \text{SuperResult}[\text{State}]]):$$

$$\exists (a: \text{Address}):$$

$$\text{addresses}'(a) \wedge q = \text{expr_2_super}(\text{memory_read}(\text{pm}')(a))))):$$

$$x' = \text{em_lift}(p)$$

Repeatedly Skolemizing and flattening,
 Instantiating the top quantifier in 1 with the terms: (expr_2_super(pm!1'memory_read(a!1))),
 we get 2 subgoals:

em_memory_read_transformers.1.1:

$$\frac{\begin{array}{l} \{-1\} \text{ addresses}'(a') \\ \{-2\} x' = \text{expr_2_super}(\text{em_lift}[\text{State}, T, \text{Byte}] (\text{pm}'\text{'memory_read}(a'))) \end{array}}{\{1\} x' = \text{em_lift}(\text{expr_2_super}(\text{pm}'\text{'memory_read}(a)))}$$

Rewriting using em_lift_expr_2_super[State, T, Byte], matching in *,
This completes the proof of em_memory_read_transformers.1.1.

em_memory_read_transformers.1.2:

$$\frac{\begin{array}{l} \{-1\} \text{ addresses}'(a') \\ \{-2\} x' = \text{expr_2_super}(\text{em_lift}[\text{State}, T, \text{Byte}] (\text{pm}'\text{'memory_read}(a'))) \end{array}}{\{1\} \exists (a: \text{Address}): \\ \text{addresses}'(a) \wedge \\ \text{expr_2_super}[\text{State}, \text{Byte}] (\text{pm}'\text{'memory_read}(a')) = \\ \text{expr_2_super}[\text{State}, \text{Byte}] (\text{memory_read}(\text{pm}') (a))}$$

Instantiating quantified variables,

This completes the proof of em_memory_read_transformers.1.2.

em_memory_read_transformers.2:

$$\frac{\begin{array}{l} \{-1\} \exists (p: \\ \quad (\lambda (q: [\text{State} \rightarrow \text{SuperResult}[\text{State}]]): \\ \quad \quad \exists (a: \text{Address}): \\ \quad \quad \quad \text{addresses}'(a) \wedge q = \text{expr_2_super}(\text{memory_read}(\text{pm}') (a))))): \\ x' = \text{em_lift}(p) \end{array}}{\{1\} \exists (a: \text{Address}): \\ \text{addresses}'(a) \wedge x' = \text{expr_2_super}(\text{em_lift}[\text{State}, T, \text{Byte}] (\text{pm}'\text{'memory_read}(a)))}$$

Repeatedly Skolemizing and flattening,

Instantiating quantified variables,

Rewriting using em_lift_expr_2_super[State, T, Byte], matching in *,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of em_memory_read_transformers.2.

Q.E.D.

C.85.2 Expand_State3.em_memory_write_transformers

Terse proof for em_memory_write_transformers.

em_memory_write_transformers:

$$\frac{\begin{array}{l} \{1\} \forall (\text{addresses}: \text{PRED}[\text{Address}], \text{em_pm}: \text{Memory_struct}[\text{Expand_state}[\text{State}, T]], \\ \quad \text{pm}: \text{Memory_struct}[\text{State}]): \\ (\text{em_pm}'\text{'memory_write} = \\ \quad (\lambda (a: \text{Address}, b: \text{Byte}): \text{em_lift}[\text{State}, T, \text{Unit}] (\text{pm}'\text{'memory_write}(a, b)))) \\ \supset \\ \text{memory_write_transformers}(\text{em_pm}, \text{addresses}) = \\ \quad \text{em_lift}(\text{memory_write_transformers}(\text{pm}, \text{addresses})) \end{array}}$$

Repeatedly Skolemizing and flattening,

Applying decompose-equality,

Expanding the definition of memory_write_transformers,

Replacing using formula -1,

Hiding formulas: -1,

Expanding the definition of em_lift,

C Proof scripts

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
we get 2 subgoals:

`em_memory_write_transformers.1:`

$$\begin{array}{|l}
 \{-1\} \quad \exists (a: \text{Address}, b: \text{Byte}): \\
 \quad \text{addresses}'(a) \wedge \\
 \quad x' = \text{expr_2_super}(\text{em_lift}[\text{State}, T, \text{Unit}] (\text{pm}'\text{'memory_write}(a, b))) \\
 \hline
 \{1\} \quad \exists (p: \\
 \quad (\lambda (q: [\text{State} \rightarrow \text{SuperResult}[\text{State}]]): \\
 \quad \quad \exists (a: \text{Address}, b: \text{Byte}): \\
 \quad \quad \quad \text{addresses}'(a) \wedge q = \text{expr_2_super}(\text{memory_write}(\text{pm}') (a, b))))): \\
 \quad x' = \text{em_lift}(p)
 \end{array}$$

Repeatedly Skolemizing and flattening,

Instantiating the top quantifier in 1 with the terms: $(\text{expr_2_super}(\text{pm}'!1\text{'memory_write}(a!1, b!1)))$,
we get 2 subgoals:

`em_memory_write_transformers.1.1:`

$$\begin{array}{|l}
 \{-1\} \quad b' < \text{max_byte} \\
 \{-2\} \quad \text{addresses}'(a') \\
 \{-3\} \quad x' = \text{expr_2_super}(\text{em_lift}[\text{State}, T, \text{Unit}] (\text{pm}'\text{'memory_write}(a', b'))) \\
 \hline
 \{1\} \quad x' = \text{em_lift}(\text{expr_2_super}(\text{pm}'\text{'memory_write}(a', b')))
 \end{array}$$

Rewriting using `em_lift_expr_2_super`, matching in `*`,

This completes the proof of `em_memory_write_transformers.1.1`.

`em_memory_write_transformers.1.2:`

$$\begin{array}{|l}
 \{-1\} \quad b' < \text{max_byte} \\
 \{-2\} \quad \text{addresses}'(a') \\
 \{-3\} \quad x' = \text{expr_2_super}(\text{em_lift}[\text{State}, T, \text{Unit}] (\text{pm}'\text{'memory_write}(a', b'))) \\
 \hline
 \{1\} \quad \exists (a: \text{Address}, b: \text{Byte}): \\
 \quad \text{addresses}'(a) \wedge \\
 \quad \text{expr_2_super}[\text{State}, \text{Unit}] (\text{pm}'\text{'memory_write}(a', b')) = \\
 \quad \text{expr_2_super}[\text{State}, \text{Unit}] (\text{memory_write}(\text{pm}') (a, b))
 \end{array}$$

Instantiating quantified variables,

This completes the proof of `em_memory_write_transformers.1.2`.

`em_memory_write_transformers.2:`

$$\begin{array}{|l}
 \{-1\} \quad \exists (p: \\
 \quad (\lambda (q: [\text{State} \rightarrow \text{SuperResult}[\text{State}]]): \\
 \quad \quad \exists (a: \text{Address}, b: \text{Byte}): \\
 \quad \quad \quad \text{addresses}'(a) \wedge q = \text{expr_2_super}(\text{memory_write}(\text{pm}') (a, b))))): \\
 \quad x' = \text{em_lift}(p) \\
 \hline
 \{1\} \quad \exists (a: \text{Address}, b: \text{Byte}): \\
 \quad \text{addresses}'(a) \wedge \\
 \quad x' = \text{expr_2_super}(\text{em_lift}[\text{State}, T, \text{Unit}] (\text{pm}'\text{'memory_write}(a, b)))
 \end{array}$$

Repeatedly Skolemizing and flattening,

Instantiating quantified variables,

Rewriting using `em_lift_expr_2_super`, matching in `*`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `em_memory_write_transformers.2`.

Q.E.D.

C.85.3 Expand_State3.em_unchanged_memory_invariant

Terse proof for `em_unchanged_memory_invariant`.

`em_unchanged_memory_invariant`:

<pre>{1} ∃ (addresses: PRED[Address], em_pm: Memory_struct[Expand_state[State, T]], pm: Memory_struct[State], states: PRED[State], transformers: PRED[[State → SuperResult[State]]]): (em_pm'memory_read = (λ (a: Address): em_lift[State, T, Byte](pm'memory_read(a)))) ∧ unchanged_memory_invariant?(pm, states, transformers, addresses) ⊃ unchanged_memory_invariant?(em_pm, em_lift(states), em_lift(transformers), ad- dresses)</pre>
--

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `em_unchanged_memory_invariant`.

Q.E.D.

C.85.4 Expand_State3.em_changed_memory_invariant

Terse proof for `em_changed_memory_invariant`.

`em_changed_memory_invariant`:

<pre>{1} ∃ (addresses: PRED[Address], em_pm: Memory_struct[Expand_state[State, T]], pm: Memory_struct[State], states: PRED[State]): (em_pm'memory_read = (λ (a: Address): em_lift[State, T, Byte](pm'memory_read(a)))) ∧ (em_pm'memory_write = (λ (a: Address, b: Byte): em_lift[State, T, Unit](pm'memory_write(a, b)))) ∧ changed_memory_invariant?(pm, states, addresses) ⊃ changed_memory_invariant?(em_pm, em_lift(states), addresses)</pre>
--

Expanding the definition of `changed_memory_invariant?`,

Repeatedly Skolemizing and flattening,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 3 subgoals:

em_changed_memory_invariant.1:

{-1}	(em_pm' memory_read = (λ (a: Address): em_lift[State, T, Byte](pm' memory_read(a))))
{-2}	(em_pm' memory_write = (λ (a: Address, b: Byte): em_lift[State, T, Unit](pm' memory_write(a, b))))
{-3}	transformer_invariant?(states', memory_read_transformers(pm', addresses'))
{-4}	transformer_invariant?(states', memory_write_transformers(pm', addresses'))
{-5}	$\forall (s: \text{State}, a: \text{Address}, b: \text{Byte}):$ states'(s) ∧ addresses'(a) ∧ OK?(memory_write(pm')(a, b)(s)) ∧ OK?(memory_read(pm')(a)(state(memory_write(pm')(a, b)(s)))) ⊃ data(memory_read(pm')(a)(state(memory_write(pm')(a, b)(s)))) = b
{1}	$\forall (s: \text{Expand_state}[\text{State}, T], a: \text{Address}, b: \text{Byte}):$ em_lift(states')(s) ∧ addresses'(a) ∧ OK?(memory_write(em_pm')(a, b)(s)) ∧ OK?(memory_read(em_pm')(a)(state(memory_write(em_pm')(a, b)(s)))) ⊃ data(memory_read(em_pm')(a)(state(memory_write(em_pm')(a, b)(s)))) = b

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of em_changed_memory_invariant.1.

em_changed_memory_invariant.2:

{-1}	(em_pm' memory_read = (λ (a: Address): em_lift[State, T, Byte](pm' memory_read(a))))
{-2}	(em_pm' memory_write = (λ (a: Address, b: Byte): em_lift[State, T, Unit](pm' memory_write(a, b))))
{-3}	transformer_invariant?(states', memory_read_transformers(pm', addresses'))
{-4}	transformer_invariant?(states', memory_write_transformers(pm', addresses'))
{-5}	$\forall (s: \text{State}, a: \text{Address}, b: \text{Byte}):$ states'(s) ∧ addresses'(a) ∧ OK?(memory_write(pm')(a, b)(s)) ∧ OK?(memory_read(pm')(a)(state(memory_write(pm')(a, b)(s)))) ⊃ data(memory_read(pm')(a)(state(memory_write(pm')(a, b)(s)))) = b
{1}	transformer_invariant?(em_lift(states'), memory_write_transformers(em_pm', addresses'))

Keeping (-2 -4 1) and hiding *,

Using lemma em_memory_write_transformers,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Replacing using formula -2,

Rewriting using em_transformer_invariant, matching in *,

This completes the proof of em_changed_memory_invariant.2.

em_changed_memory_invariant.3:

<pre> {-1} (em_pm' memory_read = (λ (a: Address): em_lift[State, T, Byte](pm' memory_read(a)))) {-2} (em_pm' memory_write = (λ (a: Address, b: Byte): em_lift[State, T, Unit](pm' memory_write(a, b)))) {-3} transformer_invariant?(states', memory_read_transformers(pm', addresses')) {-4} transformer_invariant?(states', memory_write_transformers(pm', addresses')) {-5} ∀ (s: State, a: Address, b: Byte): states'(s) ∧ addresses'(a) ∧ OK?(memory_write(pm')(a, b)(s)) ∧ OK?(memory_read(pm')(a)(state(memory_write(pm')(a, b)(s)))) ⊃ data(memory_read(pm')(a)(state(memory_write(pm')(a, b)(s)))) = b </pre>
<pre> {1} transformer_invariant?(em_lift(states'), memory_read_transformers(em_pm', addresses')) </pre>

Keeping (-1 -3 1) and hiding *,

Using lemma em_memory_read_transformers,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Replacing using formula -2,

Rewriting using em_transformer_invariant, matching in *,

This completes the proof of em_changed_memory_invariant.3.

Q.E.D.

C.85.5

Expand_State3.em_const_unchanged_memory_invariant_TCC1

Terse proof for em_const_unchanged_memory_invariant_TCC1.

em_const_unchanged_memory_invariant_TCC1:

<pre> {1} ∀ (em_pm: Memory_struct[Expand_state[State, T]], pm: Memory_struct[State], q: [Expand_state[State, T] → SuperResult[Expand_state[State, T]]]): (em_pm' memory_read = (λ (a: Address): em_lift[State, T, Byte](pm' memory_read(a)))) ⊃ (∀ (s: Expand_state[State, T]): has_next_state(q(s)) ⊃ OK?[Expand_state[State, T]](q(s)) ∨ abnormal?[Expand_state[State, T]](q(s))) </pre>

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of em_const_unchanged_memory_invariant_TCC1.

Q.E.D.

C.85.6 Expand_State3.em_const_unchanged_memory_invariant

Terse proof for em_const_unchanged_memory_invariant.

em_const_unchanged_memory_invariant:

```
{1}  ∃ (addresses: PRED[Address], em_pm: Memory_struct[Expand_state[State, T]],
      pm: Memory_struct[State], states: PRED[State],
      q: [Expand_state[State, T] → SuperResult[Expand_state[State, T]]]):
  (em_pm' memory_read =
    (λ (a: Address): em_lift[State, T, Byte](pm' memory_read(a)))
    ^
    (∃ (s: Expand_state[State, T]):
      has_next_state(q(s)) ⊃ state(q(s))'state = s'state)
    ⊃ unchanged_memory_invariant?(em_pm, em_lift(states), singleton(q), addresses)
```

Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of `em_const_unchanged_memory_invariant`.
 Q.E.D.

C.85.7 Expand_State3.em_side_effect_content_unchanged

Terse proof for `em_side_effect_content_unchanged`.

em_side_effect_content_unchanged:

```
{1}  ∃ (addresses: PRED[Address],
      se_transformer:
        [[Address, list[Byte], bool] → [State → ExprResult[State, list[Byte]]]],
      states: PRED[State]):
  side_effect_content_unchanged(addresses, states, se_transformer) ⊃
  side_effect_content_unchanged(addresses, em_lift(states),
    λ (a: Address, bl: list[Byte], cp: bool):
      em_lift[State, T, list[Byte]]
        (se_transformer(a, bl, cp)))
```

Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of `em_side_effect_content_unchanged`.
 Q.E.D.

C.86 Proofs for ExprResult (result.pvs)

This theory contains no provable formal statements.

C.87 Proofs for ExprResult_adt (ExprResult_adt.pvs)

This theory contains no provable formal statements.

C.88 Proofs for ExprResult_adt_map (ExprResult_adt.pvs)

This theory contains no provable formal statements.

C.89 Proofs for ExprResult_adt_reduce (ExprResult_adt.pvs)

This theory contains no provable formal statements.

C.90 Proofs for ExpressionStatements (statements.pvs)

This theory contains no provable formal statements.

C.91 Proofs for Expression_Composition_Rewrites (state-transformer.pvs)

C.91.1 Expression_Composition_Rewrites.comp_eval_if_ok_fexpr

Terse proof for comp_eval_if_ok_fexpr.

comp_eval_if_ok_fexpr:

$$\frac{\{1\} \quad \forall (\text{expr}: [\text{State} \rightarrow \text{ExprResult}[\text{State}, \text{Data1}]], \text{fexpr}: [\text{Data1} \rightarrow [\text{State} \rightarrow \text{ExprResult}[\text{State}, \text{Data2}]]], s: \text{State}): \text{OK}?(\text{expr}(s)) \supset (\text{expr} \#\# \text{fexpr}(s) = (\text{expr} \#\# \text{fexpr}(\text{data}(\text{expr}(s))))(s)}$$

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of comp_eval_if_ok_fexpr.

Q.E.D.

C.92 Proofs for Expressions (expressions.pvs)

This theory contains no provable formal statements.

C.93 Proofs for Expt_Lemmas (vfiasco-prelude.pvs)

C.93.1 Expt_Lemmas.both_sides_expt_gt1_le

Terse proof for both_sides_expt_gt1_le.

both_sides_expt_gt1_le:

$$\frac{\{1\} \quad \forall (\text{gt1x}: \{r: \text{posreal} \mid r > 1\}, m, n: \text{nat}): m \leq n \equiv \text{expt}(\text{gt1x}, m) \leq \text{expt}(\text{gt1x}, n)}$$

Repeatedly Skolemizing and flattening,

Using lemma both_sides_expt_gt1_le_aux,

Expanding the definition of expt,

Using lemma both_sides_times_pos_le2,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of both_sides_expt_gt1_le.

Q.E.D.

C.93.2 Expt_Lemmas.expt_plus

Terse proof for `expt_plus`.

`expt_plus`:

$$\{1\} \quad \forall (n0x: \text{nzreal}, m, n: \text{nat}):$$
$$\text{expt}(n0x, m) \times \text{expt}(n0x, n) = \text{expt}(n0x, m + n)$$

Repeatedly Skolemizing and flattening,
Using lemma `expt_plus_aux`,
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `expt_plus`.
Q.E.D.

C.94 Proofs for Extended_Real (vfiasco-prelude.pvs)

C.94.1 Extended_Real.number_field_extended

Terse proof for `number_field_extended`.

`number_field_extended`:

$$\{1\} \quad \forall (x: \text{number_field}): \text{extended_real?}(x)$$

Repeatedly Skolemizing and flattening,
Rewriting using `extended_real_superset_of_number_fields`, matching in *,
This completes the proof of `number_field_extended`.
Q.E.D.

C.94.2 Extended_Real.er_plus_TCC1

Terse proof for `er_plus_TCC1`.

`er_plus_TCC1`:

$$\{1\} \quad \exists (x_1: [\text{extended_real}, \text{extended_real}] \rightarrow \text{extended_real}): \text{TRUE}$$

Instantiating the top quantifier in 1 with the terms: $(\lambda (x, y: \text{extended_real}): x)$,
which is trivially true.
This completes the proof of `er_plus_TCC1`.
Q.E.D.

C.94.3 Extended_Real.er_neg_TCC1

Terse proof for `er_neg_TCC1`.

`er_neg_TCC1`:

$$\{1\} \quad \exists (x_1: [\text{extended_real} \rightarrow \text{extended_real}]): \text{TRUE}$$

Instantiating the top quantifier in 1 with the terms: $(\lambda (x: \text{extended_real}): x)$,
which is trivially true.
This completes the proof of `er_neg_TCC1`.
Q.E.D.

C.94.4 Extended_Real.er_plus_number_field

Terse proof for er_plus_number_field.

er_plus_number_field:

$$\frac{}{\{1\} \quad \forall (a, b: \text{number_field}): \text{number_field_pred}(\text{er_plus}(a, b))}$$

Repeatedly Skolemizing and flattening,
 Rewriting using er_plus_ax, matching in *,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 This completes the proof of er_plus_number_field.
 Q.E.D.

C.94.5 Extended_Real.er_minus_number_field

Terse proof for er_minus_number_field.

er_minus_number_field:

$$\frac{}{\{1\} \quad \forall (a, b: \text{number_field}): \text{number_field_pred}(\text{er_minus}(a, b))}$$

Repeatedly Skolemizing and flattening,
 Rewriting using er_minus_ax, matching in *,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 This completes the proof of er_minus_number_field.
 Q.E.D.

C.94.6 Extended_Real.er_times_number_field

Terse proof for er_times_number_field.

er_times_number_field:

$$\frac{}{\{1\} \quad \forall (a, b: \text{number_field}): \text{number_field_pred}(\text{er_times}(a, b))}$$

Repeatedly Skolemizing and flattening,
 Rewriting using er_times_ax, matching in *,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 This completes the proof of er_times_number_field.
 Q.E.D.

C.94.7 Extended_Real.er_div_number_field

Terse proof for er_div_number_field.

er_div_number_field:

$$\frac{}{\{1\} \quad \forall (a: \text{number_field}, b: \text{nznum}): \text{number_field_pred}(\text{er_div}(a, b))}$$

Repeatedly Skolemizing and flattening,
 Rewriting using er_div_ax, matching in *,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 This completes the proof of er_div_number_field.
 Q.E.D.

C.94.8 Extended_Real.er_neg_number_field

Terse proof for `er_neg_number_field`.

`er_neg_number_field`:

$$\{1\} \quad \forall (a: \text{number_field}): \text{number_field_pred}(\text{er_neg}(a))$$

Repeatedly Skolemizing and flattening,
Rewriting using `er_neg_ax`, matching in `*`,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `er_neg_number_field`.
Q.E.D.

C.94.9 Extended_Real.er_plus_real

Terse proof for `er_plus_real`.

`er_plus_real`:

$$\{1\} \quad \forall (a, b: \text{real}): \text{real_pred}(\text{er_plus}(a, b))$$

Repeatedly Skolemizing and flattening,
Rewriting using `er_plus_ax`, matching in `*`,
Rewriting using `closed_plus`, matching in `*`,
This completes the proof of `er_plus_real`.
Q.E.D.

C.94.10 Extended_Real.er_minus_real

Terse proof for `er_minus_real`.

`er_minus_real`:

$$\{1\} \quad \forall (a, b: \text{real}): \text{real_pred}(\text{er_minus}(a, b))$$

Repeatedly Skolemizing and flattening,
Rewriting using `er_minus_ax`, matching in `*`,
Rewriting using `closed_minus`, matching in `*`,
This completes the proof of `er_minus_real`.
Q.E.D.

C.94.11 Extended_Real.er_times_real

Terse proof for `er_times_real`.

`er_times_real`:

$$\{1\} \quad \forall (a, b: \text{real}): \text{real_pred}(\text{er_times}(a, b))$$

Repeatedly Skolemizing and flattening,
Rewriting using `er_times_ax`, matching in `*`,
Rewriting using `closed_times`, matching in `*`,
This completes the proof of `er_times_real`.
Q.E.D.

C.94.12 Extended_Real.er_div_real

Terse proof for `er_div_real`.

`er_div_real`:

$\{1\} \quad \forall (a: \text{real}, b: \text{nzreal}): \text{real_pred}(\text{er_div}(a, b))$

Repeatedly Skolemizing and flattening,
 Rewriting using `er_div_ax`, matching in `*`,
 Rewriting using `closed_divides`, matching in `*`,
 This completes the proof of `er_div_real`.
 Q.E.D.

C.94.13 Extended_Real.er_neg_real

Terse proof for `er_neg_real`.

`er_neg_real`:

$\{1\} \quad \forall (a: \text{real}): \text{real_pred}(\text{er_neg}(a))$

Repeatedly Skolemizing and flattening,
 Rewriting using `er_neg_ax`, matching in `*`,
 Rewriting using `closed_neg`, matching in `*`,
 This completes the proof of `er_neg_real`.
 Q.E.D.

C.94.14 Extended_Real.er_plus_rational

Terse proof for `er_plus_rational`.

`er_plus_rational`:

$\{1\} \quad \forall (a, b: \text{rational}): \text{rational_pred}(\text{er_plus}(a, b))$

Repeatedly Skolemizing and flattening,
 Rewriting using `er_plus_ax`, matching in `*`,
 Rewriting using `closed_plus`, matching in `*`,
 This completes the proof of `er_plus_rational`.
 Q.E.D.

C.94.15 Extended_Real.er_minus_rational

Terse proof for `er_minus_rational`.

`er_minus_rational`:

$\{1\} \quad \forall (a, b: \text{rational}): \text{rational_pred}(\text{er_minus}(a, b))$

Repeatedly Skolemizing and flattening,
 Rewriting using `er_minus_ax`, matching in `*`,
 Rewriting using `closed_minus`, matching in `*`,
 This completes the proof of `er_minus_rational`.
 Q.E.D.

C.94.16 Extended_Real.er_times_rational

Terse proof for er_times_rational.

er_times_rational:

$$\{1\} \quad \forall (a, b: \text{rational}): \text{rational_pred}(\text{er_times}(a, b))$$

Repeatedly Skolemizing and flattening,
Rewriting using er_times_ax, matching in *,
Rewriting using closed_times, matching in *,
This completes the proof of er_times_rational.
Q.E.D.

C.94.17 Extended_Real.er_div_rational

Terse proof for er_div_rational.

er_div_rational:

$$\{1\} \quad \forall (a: \text{rational}, b: \text{nzrat}): \text{rational_pred}(\text{er_div}(a, b))$$

Repeatedly Skolemizing and flattening,
Rewriting using er_div_ax, matching in *,
Rewriting using closed_divides, matching in *,
This completes the proof of er_div_rational.
Q.E.D.

C.94.18 Extended_Real.er_neg_rational

Terse proof for er_neg_rational.

er_neg_rational:

$$\{1\} \quad \forall (a: \text{rational}): \text{rational_pred}(\text{er_neg}(a))$$

Repeatedly Skolemizing and flattening,
Rewriting using er_neg_ax, matching in *,
Rewriting using closed_neg, matching in *,
This completes the proof of er_neg_rational.
Q.E.D.

C.94.19 Extended_Real.er_plus_integer

Terse proof for er_plus_integer.

er_plus_integer:

$$\{1\} \quad \forall (a, b: \text{int}): \text{integer_pred}(\text{er_plus}(a, b))$$

Repeatedly Skolemizing and flattening,
Rewriting using er_plus_ax, matching in *,
Rewriting using closed_plus, matching in *,
This completes the proof of er_plus_integer.
Q.E.D.

C.94.20 Extended_Real.er_minus_integer

Terse proof for `er_minus_integer`.

`er_minus_integer`:

$$\frac{}{\{1\} \quad \forall (a, b: \text{int}): \text{integer_pred}(\text{er_minus}(a, b))}$$

Repeatedly Skolemizing and flattening,
 Rewriting using `er_minus_ax`, matching in *,
 Rewriting using `closed_minus`, matching in *,
 This completes the proof of `er_minus_integer`.
 Q.E.D.

C.94.21 Extended_Real.er_times_integer

Terse proof for `er_times_integer`.

`er_times_integer`:

$$\frac{}{\{1\} \quad \forall (a, b: \text{int}): \text{integer_pred}(\text{er_times}(a, b))}$$

Repeatedly Skolemizing and flattening,
 Rewriting using `er_times_ax`, matching in *,
 Rewriting using `closed_times`, matching in *,
 This completes the proof of `er_times_integer`.
 Q.E.D.

C.94.22 Extended_Real.er_neg_integer

Terse proof for `er_neg_integer`.

`er_neg_integer`:

$$\frac{}{\{1\} \quad \forall (a: \text{int}): \text{integer_pred}(\text{er_neg}(a))}$$

Repeatedly Skolemizing and flattening,
 Rewriting using `er_neg_ax`, matching in *,
 Rewriting using `closed_neg`, matching in *,
 This completes the proof of `er_neg_integer`.
 Q.E.D.

C.95 Proofs for Finite_Set_Reduce (vfiasco-prelude.pvs)**C.95.1 Finite_Set_Reduce.list_of_finite_set_TCC1**

Terse proof for `list_of_finite_set_TCC1`.

`list_of_finite_set_TCC1`:

$$\frac{}{\{1\} \quad \forall (S: \text{finite_set}[T]): \neg \text{empty?}(S) \supset \text{nonempty?}[T](S)}$$

Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of `list_of_finite_set_TCC1`.
 Q.E.D.

C.95.2 Finite_Set_Reduce.list_of_finite_set_TCC2

Terse proof for list_of_finite_set_TCC2.

list_of_finite_set_TCC2:

$$\frac{}{\{1\} \quad \forall (S: \text{finite_set}[T]): \neg \text{empty?}(S) \supset \text{card}[T](\text{rest}[T](S)) < \text{card}[T](S)}$$

Repeatedly Skolemizing and flattening,
 Rewriting using card_rest, matching in *,
 Simplifying, rewriting, and recording with decision procedures,
 This completes the proof of list_of_finite_set_TCC2.
 Q.E.D.

C.95.3 Finite_Set_Reduce.length_list_of_finite_set

Terse proof for length_list_of_finite_set.

length_list_of_finite_set:

$$\frac{}{\{1\} \quad \forall (S: \text{finite_set}[T]): \text{length}(\text{list_of_finite_set}(S)) = \text{card}(S)}$$

Case splitting on $\forall (i: \text{nat}, S: \text{finite_set}[T]): i = \text{card}(S) \supset \text{length}(\text{list_of_finite_set}(S)) = \text{card}(S)$,
 we get 2 subgoals:

length_list_of_finite_set.1:

$$\frac{\frac{\{1\} \quad \forall (i: \text{nat}, S: \text{finite_set}[T]): \quad i = \text{card}(S) \supset \text{length}(\text{list_of_finite_set}(S)) = \text{card}(S)}{\{1\} \quad \forall (S: \text{finite_set}[T]): \text{length}(\text{list_of_finite_set}(S)) = \text{card}(S)}}{}{}$$

Repeatedly Skolemizing and flattening,
 Instantiating the top quantifier in -2 with the terms: card(S'), S',
 This completes the proof of length_list_of_finite_set.1.

length_list_of_finite_set.2:

$$\frac{\frac{\{1\} \quad \forall (i: \text{nat}, S: \text{finite_set}[T]): \quad i = \text{card}(S) \supset \text{length}(\text{list_of_finite_set}(S)) = \text{card}(S)}{\{2\} \quad \forall (S: \text{finite_set}[T]): \text{length}(\text{list_of_finite_set}(S)) = \text{card}(S)}}{}{}$$

Hiding formulas: 2,
 Inducting on i on formula 1,
 we get 2 subgoals:

length_list_of_finite_set.2.1:

$$\frac{}{\{1\} \quad \forall (S: \text{finite_set}[T]): 0 = \text{card}(S) \supset \text{length}(\text{list_of_finite_set}(S)) = \text{card}(S)}$$

Repeatedly Skolemizing and flattening,
 Using lemma card_empty?[T],
 Simplifying, rewriting, and recording with decision procedures,
 Simplifying, rewriting, and recording with decision procedures,
 Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of length_list_of_finite_set.2.1.

`length_list_of_finite_set.2.2:`

$\{1\} \quad \forall j:$ $\quad (\forall (S: \text{finite_set}[T]):$ $\quad \quad j = \text{card}(S) \supset \text{length}(\text{list_of_finite_set}(S)) = \text{card}(S))$ $\quad \supset$ $\quad (\forall (S: \text{finite_set}[T]):$ $\quad \quad j + 1 = \text{card}(S) \supset \text{length}(\text{list_of_finite_set}(S)) = \text{card}(S))$
--

Repeatedly Skolemizing and flattening,
 Using lemma `card_empty?[T]`,
 Simplifying, rewriting, and recording with decision procedures,
 Simplifying, rewriting, and recording with decision procedures,
 Expanding the definition of `list_of_finite_set`,
 Expanding the definition of `length`,
 Instantiating quantified variables,
 Rewriting using `card_rest`, matching in `*`,
 Simplifying, rewriting, and recording with decision procedures,
 This completes the proof of `length_list_of_finite_set.2.2`.
 Q.E.D.

C.95.4 *Finite_Set_Reduce.member_list_of_finite_set*

Terse proof for `member_list_of_finite_set`.

`member_list_of_finite_set:`

$\{1\} \quad \forall (S: \text{finite_set}[T], x: T): (x \in S) = \text{member}(x, \text{list_of_finite_set}(S))$

Case splitting on $\forall (S: \text{finite_set}[T], k: \text{nat}, x: T): k = \text{card}(S) \supset (\text{member}(x, S) = \text{member}(x, \text{list_of_finite_set}(S)))$,

we get 2 subgoals:

`member_list_of_finite_set.1:`

$\{-1\} \quad \forall (S: \text{finite_set}[T], k: \text{nat}, x: T):$ $\quad k = \text{card}(S) \supset ((x \in S) = \text{member}(x, \text{list_of_finite_set}(S)))$
$\{1\} \quad \forall (S: \text{finite_set}[T], x: T): (x \in S) = \text{member}(x, \text{list_of_finite_set}(S))$

Repeatedly Skolemizing and flattening,
 Instantiating quantified variables,
 Instantiating quantified variables,
 This completes the proof of `member_list_of_finite_set.1`.

`member_list_of_finite_set.2:`

$\{1\} \quad \forall (S: \text{finite_set}[T], k: \text{nat}, x: T):$ $\quad k = \text{card}(S) \supset ((x \in S) = \text{member}(x, \text{list_of_finite_set}(S)))$
$\{2\} \quad \forall (S: \text{finite_set}[T], x: T): (x \in S) = \text{member}(x, \text{list_of_finite_set}(S))$

Hiding formulas: 2,
 Inducting on `k` on formula 1,
 we get 2 subgoals:

member_list_of_finite_set.2.1:

$$\frac{\{1\} \quad \forall (S: \text{finite_set}[T], x: T): \quad 0 = \text{card}(S) \supset ((x \in S) = \text{member}(x, \text{list_of_finite_set}(S)))}{}$$

Repeatedly Skolemizing and flattening,
 Using lemma card_empty?[T],
 Simplifying, rewriting, and recording with decision procedures,
 Simplifying, rewriting, and recording with decision procedures,
 Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of member_list_of_finite_set.2.1.

member_list_of_finite_set.2.2:

$$\frac{\{1\} \quad \forall j: \quad (\forall (S: \text{finite_set}[T], x: T): \quad j = \text{card}(S) \supset ((x \in S) = \text{member}(x, \text{list_of_finite_set}(S)))) \quad \supset \quad (\forall (S: \text{finite_set}[T], x: T): \quad j + 1 = \text{card}(S) \supset ((x \in S) = \text{member}(x, \text{list_of_finite_set}(S))))}{}$$

Repeatedly Skolemizing and flattening,
 Using lemma card_empty?[T],
 Simplifying, rewriting, and recording with decision procedures,
 Simplifying, rewriting, and recording with decision procedures,
 Expanding the definition of list_of_finite_set,
 Expanding the definition of member,
 Case splitting on x!1 = choose(S!1),
 we get 2 subgoals:

member_list_of_finite_set.2.2.1:

$$\frac{\begin{array}{l} \{-1\} \quad x' = \text{choose}(S') \\ \{-2\} \quad \text{is_finite}[T](S') \\ \{-3\} \quad j' \geq 0 \\ \{-4\} \quad \forall (S: \text{finite_set}[T], x: T): \\ \quad \quad j' = \text{card}(S) \supset ((x \in S) = \text{member}(x, \text{list_of_finite_set}(S))) \\ \{-5\} \quad 1 + j' = \text{card}(S') \end{array}}{\begin{array}{l} \{1\} \quad \text{empty?}(S') \\ \{2\} \quad ((x' \in S') = (x' = \text{choose}(S') \vee \text{member}(x', \text{list_of_finite_set}(\text{rest}(S'))))) \end{array}}$$

Simplifying, rewriting, and recording with decision procedures,
 Expanding the definition of member,
 Simplifying, rewriting, and recording with decision procedures,
 This completes the proof of member_list_of_finite_set.2.2.1.

member_list_of_finite_set.2.2.2:

$$\frac{\begin{array}{l} \{-1\} \quad \text{is_finite}[T](S') \\ \{-2\} \quad j' \geq 0 \\ \{-3\} \quad \forall (S: \text{finite_set}[T], x: T): \\ \quad \quad j' = \text{card}(S) \supset ((x \in S) = \text{member}(x, \text{list_of_finite_set}(S))) \\ \{-4\} \quad 1 + j' = \text{card}(S') \end{array}}{\begin{array}{l} \{1\} \quad x' = \text{choose}(S') \\ \{2\} \quad \text{empty?}(S') \\ \{3\} \quad ((x' \in S') = (x' = \text{choose}(S') \vee \text{member}(x', \text{list_of_finite_set}(\text{rest}(S'))))) \end{array}}$$

Simplifying, rewriting, and recording with decision procedures,

Instantiating the top quantifier in -3 with the terms: $\text{rest}(S')$, x' ,
 Installing automatic rewrites from: `card_rest`
 Simplifying, rewriting, and recording with decision procedures,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 we get 2 subgoals:

`member_list_of_finite_set.2.2.2.1:`

{-1}	$\text{is_finite}[T](S')$
{-2}	$j' \geq 0$
{-3}	$(x' \in \text{rest}(S'))$
{-4}	$\text{member}(x', \text{list_of_finite_set}(\text{rest}(S')))$
{-5}	$1 + j' = \text{card}(S')$
{1}	$x' = \text{choose}(S')$
{2}	$\text{empty?}(S')$
{3}	$(x' \in S')$

Using lemma `rest_member`,
 Simplifying, rewriting, and recording with decision procedures,
 This completes the proof of `member_list_of_finite_set.2.2.2.1`.

`member_list_of_finite_set.2.2.2.2:`

{-1}	$\text{is_finite}[T](S')$
{-2}	$j' \geq 0$
{-3}	$1 + j' = \text{card}(S')$
{-4}	$(x' \in S')$
{1}	$x' = \text{choose}(S')$
{2}	$\text{empty?}(S')$
{3}	$(x' \in \text{rest}(S'))$
{4}	$\text{member}(x', \text{list_of_finite_set}(\text{rest}(S')))$

Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of `member_list_of_finite_set.2.2.2.2`.
 Q.E.D.

C.95.5 Finite_Set_Reduce.unique_member_list_of_finite_set

Terse proof for `unique_member_list_of_finite_set`.

`unique_member_list_of_finite_set:`

{1}	$\forall (S: \text{finite_set}[T], i, j: \text{below}(\text{length}(\text{list_of_finite_set}(S)))):$ $\text{nth}(\text{list_of_finite_set}(S), i) = \text{nth}(\text{list_of_finite_set}(S), j) \supset i = j$

Case splitting on $\forall (S: \text{finite_set}[T], k: \text{nat}, i, j: \text{below}(\text{length}(\text{list_of_finite_set}(S)))): k = \text{card}(S) \wedge \text{nth}(\text{list_of_finite_set}(S), i) = \text{nth}(\text{list_of_finite_set}(S), j) \supset i = j$,
 we get 2 subgoals:

`unique_member_list_of_finite_set.1:`

{-1}	$\forall (S: \text{finite_set}[T], k: \text{nat}, i, j: \text{below}(\text{length}(\text{list_of_finite_set}(S)))):$ $k = \text{card}(S) \wedge \text{nth}(\text{list_of_finite_set}(S), i) = \text{nth}(\text{list_of_finite_set}(S), j) \supset$ $i = j$
{1}	$\forall (S: \text{finite_set}[T], i, j: \text{below}(\text{length}(\text{list_of_finite_set}(S)))):$ $\text{nth}(\text{list_of_finite_set}(S), i) = \text{nth}(\text{list_of_finite_set}(S), j) \supset i = j$

Repeatedly Skolemizing and flattening,
 Instantiating the top quantifier in -4 with the terms: S' , $\text{card}(S')$, i' , j' ,

C Proof scripts

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `unique_member_list_of_finite_set.1`.

`unique_member_list_of_finite_set.2`:

$\{1\} \quad \forall (S: \text{finite_set}[T], k: \text{nat}, i, j: \text{below}(\text{length}(\text{list_of_finite_set}(S)))):$ $k = \text{card}(S) \wedge \text{nth}(\text{list_of_finite_set}(S), i) = \text{nth}(\text{list_of_finite_set}(S), j) \supset$ $i = j$
$\{2\} \quad \forall (S: \text{finite_set}[T], i, j: \text{below}(\text{length}(\text{list_of_finite_set}(S)))):$ $\text{nth}(\text{list_of_finite_set}(S), i) = \text{nth}(\text{list_of_finite_set}(S), j) \supset i = j$

Hiding formulas: 2,

Inducting on k on formula 1,

we get 2 subgoals:

`unique_member_list_of_finite_set.2.1`:

$\{1\} \quad \forall (S: \text{finite_set}[T], i, j: \text{below}(\text{length}(\text{list_of_finite_set}(S)))):$ $0 = \text{card}(S) \wedge \text{nth}(\text{list_of_finite_set}(S), i) = \text{nth}(\text{list_of_finite_set}(S), j) \supset$ $i = j$

Repeatedly Skolemizing and flattening,

Using lemma `card_empty?[T]`,

Simplifying, rewriting, and recording with decision procedures,

Simplifying, rewriting, and recording with decision procedures,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `unique_member_list_of_finite_set.2.1`.

`unique_member_list_of_finite_set.2.2`:

$\{1\} \quad \forall (j_1: \text{nat}):$ $(\forall (S: \text{finite_set}[T], i, j: \text{below}(\text{length}(\text{list_of_finite_set}(S))))):$ $j_1 = \text{card}(S) \wedge$ $\text{nth}(\text{list_of_finite_set}(S), i) = \text{nth}(\text{list_of_finite_set}(S), j)$ $\supset i = j$ \supset $(\forall (S: \text{finite_set}[T], i, j: \text{below}(\text{length}(\text{list_of_finite_set}(S))))):$ $j_1 + 1 = \text{card}(S) \wedge$ $\text{nth}(\text{list_of_finite_set}(S), i) = \text{nth}(\text{list_of_finite_set}(S), j)$ $\supset i = j$

Repeatedly Skolemizing and flattening,

Using lemma `card_empty?[T]`,

Simplifying, rewriting, and recording with decision procedures,

Simplifying, rewriting, and recording with decision procedures,

Expanding the definition of `list_of_finite_set`,

Case splitting on `i!1 = 0`,

we get 2 subgoals:

unique_member_list_of_finite_set.2.2.1:

{-1}	$i' = 0$
{-2}	$\text{is_finite}[T](S')$
{-3}	$i' < \text{length}(\text{list_of_finite_set}(S'))$
{-4}	$j'' < \text{length}(\text{list_of_finite_set}(S'))$
{-5}	$j' \geq 0$
{-6}	$\forall (S: \text{finite_set}[T], i, j: \text{below}(\text{length}(\text{list_of_finite_set}(S)))):$ $j' = \text{card}(S) \wedge \text{nth}(\text{list_of_finite_set}(S), i) = \text{nth}(\text{list_of_finite_set}(S), j)$ $\supset i = j$
{-7}	$1 + j' = \text{card}(S')$
{-8}	$\text{nth}(\text{cons}(\text{choose}(S'), \text{list_of_finite_set}(\text{rest}(S'))), i') =$ $\text{nth}(\text{cons}(\text{choose}(S'), \text{list_of_finite_set}(\text{rest}(S'))), j'')$
{1}	
	$\text{empty?}(S')$
{2}	$i' = j''$

Expanding the definition of nth,

Simplifying, rewriting, and recording with decision procedures,

Hiding formulas: -6,

Using lemma member_list_of_finite_set,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 2 subgoals:

unique_member_list_of_finite_set.2.2.1.1:

{-1}	$(\text{choose}(S') \in \text{rest}(S'))$
{-2}	$\text{member}(\text{choose}(S'), \text{list_of_finite_set}(\text{rest}(S')))$
{-3}	$i' = 0$
{-4}	$\text{is_finite}[T](S')$
{-5}	$i' < \text{length}(\text{list_of_finite_set}(S'))$
{-6}	$j'' < \text{length}(\text{list_of_finite_set}(S'))$
{-7}	$j' \geq 0$
{-8}	$1 + j' = \text{card}(S')$
{-9}	$\text{choose}(S') = \text{nth}(\text{list_of_finite_set}(\text{rest}(S')), j'' - 1)$
{1}	
	$\text{empty?}(S')$
{2}	$i' = j''$

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of unique_member_list_of_finite_set.2.2.1.1.

unique_member_list_of_finite_set.2.2.1.2:

{-1}	$i' = 0$
{-2}	$\text{is_finite}[T](S')$
{-3}	$i' < \text{length}(\text{list_of_finite_set}(S'))$
{-4}	$j'' < \text{length}(\text{list_of_finite_set}(S'))$
{-5}	$j' \geq 0$
{-6}	$1 + j' = \text{card}(S')$
{-7}	$\text{choose}(S') = \text{nth}(\text{list_of_finite_set}(\text{rest}(S')), j'' - 1)$
{1}	
	$(\text{choose}(S') \in \text{rest}(S'))$
{2}	$\text{member}(\text{choose}(S'), \text{list_of_finite_set}(\text{rest}(S')))$
{3}	$\text{empty?}(S')$
{4}	$i' = j''$

Hiding formulas: 1,

Using lemma nth_member[T],

Simplifying, rewriting, and recording with decision procedures,

C Proof scripts

Instantiating quantified variables,

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `unique_member_list_of_finite_set.2.2.1.2`.

`unique_member_list_of_finite_set.2.2.2`:

{-1}	$\text{is_finite}[T](S')$
{-2}	$i' < \text{length}(\text{list_of_finite_set}(S'))$
{-3}	$j'' < \text{length}(\text{list_of_finite_set}(S'))$
{-4}	$j' \geq 0$
{-5}	$\forall (S: \text{finite_set}[T], i, j: \text{below}(\text{length}(\text{list_of_finite_set}(S)))):$ $j' = \text{card}(S) \wedge \text{nth}(\text{list_of_finite_set}(S), i) = \text{nth}(\text{list_of_finite_set}(S), j)$ $\supset i = j$
{-6}	$1 + j' = \text{card}(S')$
{-7}	$\text{nth}(\text{cons}(\text{choose}(S'), \text{list_of_finite_set}(\text{rest}(S'))), i') =$ $\text{nth}(\text{cons}(\text{choose}(S'), \text{list_of_finite_set}(\text{rest}(S'))), j'')$
{1}	$i' = 0$
{2}	$\text{empty?}(S')$
{3}	$i' = j''$

Case splitting on $j!2 = 0$,

we get 2 subgoals:

`unique_member_list_of_finite_set.2.2.2.1`:

{-1}	$j'' = 0$
{-2}	$\text{is_finite}[T](S')$
{-3}	$i' < \text{length}(\text{list_of_finite_set}(S'))$
{-4}	$j'' < \text{length}(\text{list_of_finite_set}(S'))$
{-5}	$j' \geq 0$
{-6}	$\forall (S: \text{finite_set}[T], i, j: \text{below}(\text{length}(\text{list_of_finite_set}(S)))):$ $j' = \text{card}(S) \wedge \text{nth}(\text{list_of_finite_set}(S), i) = \text{nth}(\text{list_of_finite_set}(S), j)$ $\supset i = j$
{-7}	$1 + j' = \text{card}(S')$
{-8}	$\text{nth}(\text{cons}(\text{choose}(S'), \text{list_of_finite_set}(\text{rest}(S'))), i') =$ $\text{nth}(\text{cons}(\text{choose}(S'), \text{list_of_finite_set}(\text{rest}(S'))), j'')$
{1}	$i' = 0$
{2}	$\text{empty?}(S')$
{3}	$i' = j''$

Hiding formulas: -6,

Expanding the definition of `nth`,

Simplifying, rewriting, and recording with decision procedures,

Using lemma `member_list_of_finite_set`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 2 subgoals:

unique_member_list_of_finite_set.2.2.2.1.1:

{-1}	$(\text{choose}(S') \in \text{rest}(S'))$
{-2}	$\text{member}(\text{choose}(S'), \text{list_of_finite_set}(\text{rest}(S')))$
{-3}	$j'' = 0$
{-4}	$\text{is_finite}[T](S')$
{-5}	$i' < \text{length}(\text{list_of_finite_set}(S'))$
{-6}	$j'' < \text{length}(\text{list_of_finite_set}(S'))$
{-7}	$j' \geq 0$
{-8}	$1 + j' = \text{card}(S')$
{-9}	$\text{nth}(\text{list_of_finite_set}(\text{rest}(S')), i' - 1) = \text{choose}(S')$
{1}	$i' = 0$
{2}	$\text{empty?}(S')$
{3}	$i' = j''$

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of unique_member_list_of_finite_set.2.2.2.1.1.

unique_member_list_of_finite_set.2.2.2.1.2:

{-1}	$j'' = 0$
{-2}	$\text{is_finite}[T](S')$
{-3}	$i' < \text{length}(\text{list_of_finite_set}(S'))$
{-4}	$j'' < \text{length}(\text{list_of_finite_set}(S'))$
{-5}	$j' \geq 0$
{-6}	$1 + j' = \text{card}(S')$
{-7}	$\text{nth}(\text{list_of_finite_set}(\text{rest}(S')), i' - 1) = \text{choose}(S')$
{1}	$(\text{choose}(S') \in \text{rest}(S'))$
{2}	$\text{member}(\text{choose}(S'), \text{list_of_finite_set}(\text{rest}(S')))$
{3}	$i' = 0$
{4}	$\text{empty?}(S')$
{5}	$i' = j''$

Hiding formulas: 1,

Using lemma nth_member[T],

Simplifying, rewriting, and recording with decision procedures,

Instantiating quantified variables,

This completes the proof of unique_member_list_of_finite_set.2.2.2.1.2.

unique_member_list_of_finite_set.2.2.2.2:

{-1}	$\text{is_finite}[T](S')$
{-2}	$i' < \text{length}(\text{list_of_finite_set}(S'))$
{-3}	$j'' < \text{length}(\text{list_of_finite_set}(S'))$
{-4}	$j' \geq 0$
{-5}	$\forall (S: \text{finite_set}[T], i, j: \text{below}(\text{length}(\text{list_of_finite_set}(S)))):$ $\quad j' = \text{card}(S) \wedge \text{nth}(\text{list_of_finite_set}(S), i) = \text{nth}(\text{list_of_finite_set}(S), j)$ $\quad \supset i = j$
{-6}	$1 + j' = \text{card}(S')$
{-7}	$\text{nth}(\text{cons}(\text{choose}(S'), \text{list_of_finite_set}(\text{rest}(S'))), i') =$ $\quad \text{nth}(\text{cons}(\text{choose}(S'), \text{list_of_finite_set}(\text{rest}(S'))), j'')$
{1}	$j'' = 0$
{2}	$i' = 0$
{3}	$\text{empty?}(S')$
{4}	$i' = j''$

Expanding the definition of nth,

Simplifying, rewriting, and recording with decision procedures,
 Instantiating the top quantifier in -5 with the terms: $\text{rest}(S')$, $i' - 1$, $j'' - 1$,
 Rewriting using `card_rest`, matching in `*`,
 Simplifying, rewriting, and recording with decision procedures,
 This completes the proof of `unique_member_list_of_finite_set.2.2.2.2`.
 Q.E.D.

C.96 Proofs for General (constants.pvs)

C.96.1 General.Byte_TCC1

Terse proof for `Byte_TCC1`.

`Byte_TCC1`:

{1} $0 < \text{max_byte}$

Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of `Byte_TCC1`.
 Q.E.D.

C.96.2 General.less_than_max_byte

Terse proof for `less_than_max_byte`.

`less_than_max_byte`:

{1} $\forall (n: \text{nat}): n < \text{expt}(2, 8) \supset n < \text{max_byte}$

Repeatedly Skolemizing and flattening,
 Expanding the definition of `max_byte`,
 Applying `bits_per_byte_minimum`
 Expanding the definition of `min_bits_per_byte`,
 Applying `both_sides_expt_gt1_le`
 Instantiating the top quantifier in -1 with the terms: 2, 8, `bits_per_byte`,
 Simplifying, rewriting, and recording with decision procedures,
 This completes the proof of `less_than_max_byte`.
 Q.E.D.

C.97 Proofs for Graph (graph.pvs)

C.97.1 Graph.path?_TCC1

Terse proof for `path?_TCC1`.

`path?_TCC1`:

{1} $\forall (g: \text{graph_type}, l: (\text{path_list?}[T])):$
 $(\forall (i: \text{below}(\text{length}(l))): g.\text{nodes}(\text{nth}(l, i))) \supset$
 $(\forall (i_1: \text{below}(\text{length}[T](l) - 1)): i_1 < \text{length}[T](l))$

Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of `path?_TCC1`.

Q.E.D.

C.97.2 Graph.path?_TCC2

Terse proof for path?_TCC2.

path?_TCC2:

$$\{1\} \quad \forall (g: \text{graph_type}, l: (\text{path_list?}[T])): \\ (\forall (i: \text{below}(\text{length}(l))): g' \text{nodes}(\text{nth}(l, i))) \supset \\ (\forall (i_1: \text{below}(\text{length}[T](l) - 1)): g' \text{nodes}(\text{nth}[T](l, i_1)))$$

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of path?_TCC2.

Q.E.D.

C.97.3 Graph.path?_TCC3

Terse proof for path?_TCC3.

path?_TCC3:

$$\{1\} \quad \forall (g: \text{graph_type}, l: (\text{path_list?}[T])): \\ (\forall (i: \text{below}(\text{length}(l))): g' \text{nodes}(\text{nth}(l, i))) \supset \\ (\forall (i_1: \text{below}(\text{length}[T](l) - 1)): i_1 + 1 < \text{length}[T](l))$$

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of path?_TCC3.

Q.E.D.

C.97.4 Graph.path?_TCC4

Terse proof for path?_TCC4.

path?_TCC4:

$$\{1\} \quad \forall (g: \text{graph_type}, l: (\text{path_list?}[T])): \\ (\forall (i: \text{below}(\text{length}(l))): g' \text{nodes}(\text{nth}(l, i))) \supset \\ (\forall (i_1: \text{below}(\text{length}[T](l) - 1)): g' \text{nodes}(\text{nth}[T](l, i_1 + 1)))$$

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of path?_TCC4.

Q.E.D.

C.97.5 Graph.nodes_path_start

Terse proof for nodes_path_start.

nodes_path_start:

$$\{1\} \quad \forall (g: \text{graph_type}, p: (\text{path?}(g))): g' \text{nodes}(\text{path_start}(p))$$

Repeatedly Skolemizing and flattening,
Expanding the definition of path_list?,
Expanding the definition of path_start,

Expanding the definition of path?,
 Applying disjunctive simplification to flatten sequent,
 Instantiating the top quantifier in -2 with the terms: 0,
 we get 2 subgoals:

nodes_path_start.1:

{-1}	$\text{length}(p') \geq 1$
{-2}	$g' \text{ nodes}(\text{nth}(p', 0))$
{-3}	$\forall (i: \text{below}(\text{length}(p') - 1)): g' \text{ edges}(\text{nth}(p', i), \text{nth}(p', 1 + i))$
{1}	$g' \text{ nodes}(\text{car}(p'))$

Expanding the definition of nth,
 which is trivially true.

This completes the proof of nodes_path_start.1.

nodes_path_start.2:

{-1}	$\text{length}(p') \geq 1$
{-2}	$\forall (i: \text{below}(\text{length}(p') - 1)): g' \text{ edges}(\text{nth}(p', i), \text{nth}(p', 1 + i))$
{1}	$0 < \text{length}[T](p')$
{2}	$g' \text{ nodes}(\text{car}(p'))$

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of nodes_path_start.2.

Q.E.D.

C.97.6 Graph.nodes_path_end

Terse proof for nodes_path_end.

nodes_path_end:

{1}	$\forall (g: \text{graph_type}, p: (\text{path?}(g))): g' \text{ nodes}(\text{path_end}(p))$
-----	--

Repeatedly Skolemizing and flattening,
 Expanding the definition of path_list?,
 Expanding the definition of path_end,
 Expanding the definition of path?,
 Applying disjunctive simplification to flatten sequent,
 Instantiating quantified variables,
 Simplifying, rewriting, and recording with decision procedures,
 This completes the proof of nodes_path_end.

Q.E.D.

C.97.7 Graph.path_head_TCC1

Terse proof for path_head_TCC1.

path_head_TCC1:

{1}	$\forall (g: \text{graph_type}, p: (\text{path?}(g)), n: \text{upto}(\text{length}[T](p))):$ $n \geq 1 \supset \text{path_list?}[T](\text{head}[T](p, n))$
-----	---

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of path_head_TCC1.

Q.E.D.

C.97.8 Graph.path_head

Terse proof for path_head.

path_head:

$$\frac{}{\{1\} \quad \forall (g: \text{graph_type}, p: (\text{path?}(g)), n: \text{upto}(\text{length}(p))): \\ n \geq 1 \supset \text{path?}(g)(\text{head}(p, n))}$$

Repeatedly Skolemizing and flattening,

Expanding the definition of path?,

Installing automatic rewrites from: length_head length_tail nth_head nth_tail

Applying propositional simplification,

we get 2 subgoals:

path_head.1:

$$\frac{\begin{array}{l} \{-1\} \quad \text{path_list?}[T](p') \\ \{-2\} \quad \forall (i: \text{below}(\text{length}(p'))): g' \text{'nodes}(\text{nth}(p', i)) \\ \{-3\} \quad \forall (i: \text{below}(\text{length}(p') - 1)): g' \text{'edges}(\text{nth}(p', i), \text{nth}(p', 1 + i)) \\ \{-4\} \quad n' \leq \text{length}(p') \\ \{-5\} \quad n' \geq 1 \end{array}}{\{1\} \quad \forall (i: \text{below}(\text{length}(\text{head}(p', n')))): g' \text{'nodes}(\text{nth}(\text{head}(p', n'), i))}$$

Repeatedly Skolemizing and flattening,

Simplifying, rewriting, and recording with decision procedures,

Instantiating quantified variables,

This completes the proof of path_head.1.

path_head.2:

$$\frac{\begin{array}{l} \{-1\} \quad \text{path_list?}[T](p') \\ \{-2\} \quad \forall (i: \text{below}(\text{length}(p'))): g' \text{'nodes}(\text{nth}(p', i)) \\ \{-3\} \quad \forall (i: \text{below}(\text{length}(p') - 1)): g' \text{'edges}(\text{nth}(p', i), \text{nth}(p', 1 + i)) \\ \{-4\} \quad n' \leq \text{length}(p') \\ \{-5\} \quad n' \geq 1 \end{array}}{\{1\} \quad \forall (i: \text{below}(\text{length}(\text{head}(p', n')) - 1)): \\ g' \text{'edges}(\text{nth}(\text{head}(p', n'), i), \text{nth}(\text{head}(p', n'), 1 + i))}$$

Repeatedly Skolemizing and flattening,

Simplifying, rewriting, and recording with decision procedures,

Instantiating quantified variables,

This completes the proof of path_head.2.

Q.E.D.

C.97.9 Graph.path_tail_TCC1

Terse proof for path_tail_TCC1.

path_tail_TCC1:

$$\frac{}{\{1\} \quad \forall (g: \text{graph_type}, p: (\text{path?}(g)), n: \text{upto}(\text{length}[T](p))): \\ \text{length}(p) - n \geq 1 \supset \text{path_list?}[T](\text{tail}[T](p, n))}$$

Repeatedly Skolemizing and flattening,

Expanding the definition of path_list?,

Rewriting using length_tail, matching in *,

This completes the proof of path_tail_TCC1.

Q.E.D.

C.97.10 Graph.path_tail

Terse proof for path_tail.

path_tail:

$$\frac{}{\{1\} \quad \forall (g: \text{graph_type}, p: (\text{path?}(g)), n: \text{upto}(\text{length}(p))): \\ \text{length}(p) - n \geq 1 \supset \text{path?}(g)(\text{tail}(p, n))}$$

Repeatedly Skolemizing and flattening,

Expanding the definition of path?,

Installing automatic rewrites from: length_head length_tail nth_head nth_tail

Applying propositional simplification,

we get 2 subgoals:

path_tail.1:

$$\frac{\begin{array}{l} \{-1\} \quad \text{path_list?}[T](p') \\ \{-2\} \quad \forall (i: \text{below}(\text{length}(p'))): g' \text{'nodes}(\text{nth}(p', i)) \\ \{-3\} \quad \forall (i: \text{below}(\text{length}(p') - 1)): g' \text{'edges}(\text{nth}(p', i), \text{nth}(p', 1 + i)) \\ \{-4\} \quad n' \leq \text{length}(p') \\ \{-5\} \quad \text{length}(p') - n' \geq 1 \end{array}}{\{1\} \quad \forall (i: \text{below}(\text{length}(\text{tail}(p', n')))): g' \text{'nodes}(\text{nth}(\text{tail}(p', n'), i))}$$

Repeatedly Skolemizing and flattening,

Simplifying, rewriting, and recording with decision procedures,

Instantiating quantified variables,

This completes the proof of path_tail.1.

path_tail.2:

$$\frac{\begin{array}{l} \{-1\} \quad \text{path_list?}[T](p') \\ \{-2\} \quad \forall (i: \text{below}(\text{length}(p'))): g' \text{'nodes}(\text{nth}(p', i)) \\ \{-3\} \quad \forall (i: \text{below}(\text{length}(p') - 1)): g' \text{'edges}(\text{nth}(p', i), \text{nth}(p', 1 + i)) \\ \{-4\} \quad n' \leq \text{length}(p') \\ \{-5\} \quad \text{length}(p') - n' \geq 1 \end{array}}{\{1\} \quad \forall (i: \text{below}(\text{length}(\text{tail}(p', n') - 1)): \\ g' \text{'edges}(\text{nth}(\text{tail}(p', n'), i), \text{nth}(\text{tail}(p', n'), 1 + i))}$$

Repeatedly Skolemizing and flattening,

Simplifying, rewriting, and recording with decision procedures,

Instantiating quantified variables,

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of path_tail.2.

Q.E.D.

C.97.11 Graph.path_cdr_TCC1

Terse proof for path_cdr_TCC1.

path_cdr_TCC1:

$$\frac{}{\{1\} \quad \forall (g: \text{graph_type}, p: (\text{path?}(g))): \text{cons?}[T](p)}$$

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of path_cdr_TCC1.

Q.E.D.

C.97.12 Graph.path_cdr

Terse proof for path_cdr.

path_cdr:

$$\{1\} \quad \forall (g: \text{graph_type}, p: (\text{path?}(g))):$$

$$\text{path_list?}[T](\text{cdr}(p)) \supset \text{path?}(g)(\text{cdr}(p))$$

Repeatedly Skolemizing and flattening,
 Expanding the definition of path_list?,
 Installing automatic rewrites from: length_cdr tail
 Simplifying, rewriting, and recording with decision procedures,
 Using lemma path_tail,
 Simplifying, rewriting, and recording with decision procedures,
 This completes the proof of path_cdr.
 Q.E.D.

C.97.13 Graph.path_concat_path

Terse proof for path_concat_path.

path_concat_path:

$$\{1\} \quad \forall (g: \text{graph_type}, p_1, p_2: (\text{path?}(g))):$$

$$\text{concatable?}(p_1, p_2) \supset \text{path?}(g)(\text{concat_path}(p_1, p_2))$$

Repeatedly Skolemizing and flattening,
 Installing automatic rewrites from: path_list? path? nth_append_left nth_append_right
 length_append length_cdr length_concat_path concat_path
 Simplifying, rewriting, and recording with decision procedures,
 Applying disjunctive simplification to flatten sequent,
 Applying propositional simplification,
 we get 2 subgoals:

path_concat_path.1:

$$\{-1\} \quad \text{length}(p'_1) \geq 1$$

$$\{-2\} \quad \forall (i: \text{below}(\text{length}(p'_1))): g' \text{'nodes}(\text{nth}(p'_1, i))$$

$$\{-3\} \quad \forall (i: \text{below}(\text{length}(p'_1) - 1)):$$

$$g' \text{'edges}(\text{nth}(p'_1, i), \text{nth}(p'_1, 1 + i))$$

$$\{-4\} \quad \text{length}(p'_2) \geq 1$$

$$\{-5\} \quad \forall (i: \text{below}(\text{length}(p'_2))): g' \text{'nodes}(\text{nth}(p'_2, i))$$

$$\{-6\} \quad \forall (i: \text{below}(\text{length}(p'_2) - 1)):$$

$$g' \text{'edges}(\text{nth}(p'_2, i), \text{nth}(p'_2, 1 + i))$$

$$\{-7\} \quad \text{concatable?}(p'_1, p'_2)$$

$$\{1\} \quad \forall (i: \text{below}(\text{length}(\text{append}(p'_1, \text{cdr}(p'_2))))):$$

$$g' \text{'nodes}(\text{nth}(\text{append}(p'_1, \text{cdr}(p'_2)), i))$$

Hiding formulas: -3, -6,
 Repeatedly Skolemizing and flattening,
 Simplifying, rewriting, and recording with decision procedures,
 Case splitting on $i!1 < \text{length}(p!1)$,
 we get 2 subgoals:

path_concat_path.1.1:

{-1}	$i' < \text{length}(p'_1)$
{-2}	$i' < \text{length}(p'_1) + \text{length}(p'_2) - 1$
{-3}	$\text{length}(p'_1) \geq 1$
{-4}	$\forall (i : \text{below}(\text{length}(p'_1))): g' \text{ nodes}(\text{nth}(p'_1, i))$
{-5}	$\text{length}(p'_2) \geq 1$
{-6}	$\forall (i : \text{below}(\text{length}(p'_2))): g' \text{ nodes}(\text{nth}(p'_2, i))$
{-7}	$\text{concatable?}(p'_1, p'_2)$
{1}	
$g' \text{ nodes}(\text{nth}(\text{append}(p'_1, \text{cdr}(p'_2)), i'))$	

Simplifying, rewriting, and recording with decision procedures,

Instantiating quantified variables,

This completes the proof of path_concat_path.1.1.

path_concat_path.1.2:

{-1}	$i' < \text{length}(p'_1) + \text{length}(p'_2) - 1$
{-2}	$\text{length}(p'_1) \geq 1$
{-3}	$\forall (i : \text{below}(\text{length}(p'_1))): g' \text{ nodes}(\text{nth}(p'_1, i))$
{-4}	$\text{length}(p'_2) \geq 1$
{-5}	$\forall (i : \text{below}(\text{length}(p'_2))): g' \text{ nodes}(\text{nth}(p'_2, i))$
{-6}	$\text{concatable?}(p'_1, p'_2)$
{1}	
$i' < \text{length}(p'_1)$	
{2}	
$g' \text{ nodes}(\text{nth}(\text{append}(p'_1, \text{cdr}(p'_2)), i'))$	

Simplifying, rewriting, and recording with decision procedures,

Instantiating the top quantifier in -5 with the terms: $i' - \text{length}(p'_1) + 1$,

Expanding the definition of nth,

which is trivially true.

This completes the proof of path_concat_path.1.2.

path_concat_path.2:

{-1}	$\text{length}(p'_1) \geq 1$
{-2}	$\forall (i : \text{below}(\text{length}(p'_1))): g' \text{ nodes}(\text{nth}(p'_1, i))$
{-3}	$\forall (i : \text{below}(\text{length}(p'_1) - 1)):$ $g' \text{ edges}(\text{nth}(p'_1, i), \text{nth}(p'_1, 1 + i))$
{-4}	$\text{length}(p'_2) \geq 1$
{-5}	$\forall (i : \text{below}(\text{length}(p'_2))): g' \text{ nodes}(\text{nth}(p'_2, i))$
{-6}	$\forall (i : \text{below}(\text{length}(p'_2) - 1)):$ $g' \text{ edges}(\text{nth}(p'_2, i), \text{nth}(p'_2, 1 + i))$
{-7}	$\text{concatable?}(p'_1, p'_2)$
{1}	
$\forall (i : \text{below}(\text{length}(\text{append}(p'_1, \text{cdr}(p'_2))) - 1)):$ $g' \text{ edges}$ $(\text{nth}(\text{append}(p'_1, \text{cdr}(p'_2)), i), \text{nth}(\text{append}(p'_1, \text{cdr}(p'_2)), 1 + i))$	

Repeatedly Skolemizing and flattening,

Simplifying, rewriting, and recording with decision procedures,

Case splitting on $1 + i!1 < \text{length}(p1!1)$,

we get 2 subgoals:

path_concat_path.2.1:

{-1}	$1 + i' < \text{length}(p'_1)$
{-2}	$i' < \text{length}(p'_1) + \text{length}(p'_2) - 2$
{-3}	$\text{length}(p'_1) \geq 1$
{-4}	$\forall (i: \text{below}(\text{length}(p'_1))): g' \text{ nodes}(\text{nth}(p'_1, i))$
{-5}	$\forall (i: \text{below}(\text{length}(p'_1) - 1)):$ $g' \text{ edges}(\text{nth}(p'_1, i), \text{nth}(p'_1, 1 + i))$
{-6}	$\text{length}(p'_2) \geq 1$
{-7}	$\forall (i: \text{below}(\text{length}(p'_2))): g' \text{ nodes}(\text{nth}(p'_2, i))$
{-8}	$\forall (i: \text{below}(\text{length}(p'_2) - 1)):$ $g' \text{ edges}(\text{nth}(p'_2, i), \text{nth}(p'_2, 1 + i))$
{-9}	$\text{concatable?}(p'_1, p'_2)$
{1}	$g' \text{ edges}$ $(\text{nth}(\text{append}(p'_1, \text{cdr}(p'_2)), i'), \text{nth}(\text{append}(p'_1, \text{cdr}(p'_2)), 1 + i'))$

Simplifying, rewriting, and recording with decision procedures,

Instantiating quantified variables,

This completes the proof of path_concat_path.2.1.

path_concat_path.2.2:

{-1}	$i' < \text{length}(p'_1) + \text{length}(p'_2) - 2$
{-2}	$\text{length}(p'_1) \geq 1$
{-3}	$\forall (i: \text{below}(\text{length}(p'_1))): g' \text{ nodes}(\text{nth}(p'_1, i))$
{-4}	$\forall (i: \text{below}(\text{length}(p'_1) - 1)):$ $g' \text{ edges}(\text{nth}(p'_1, i), \text{nth}(p'_1, 1 + i))$
{-5}	$\text{length}(p'_2) \geq 1$
{-6}	$\forall (i: \text{below}(\text{length}(p'_2))): g' \text{ nodes}(\text{nth}(p'_2, i))$
{-7}	$\forall (i: \text{below}(\text{length}(p'_2) - 1)):$ $g' \text{ edges}(\text{nth}(p'_2, i), \text{nth}(p'_2, 1 + i))$
{-8}	$\text{concatable?}(p'_1, p'_2)$
{1}	$1 + i' < \text{length}(p'_1)$
{2}	$g' \text{ edges}$ $(\text{nth}(\text{append}(p'_1, \text{cdr}(p'_2)), i'), \text{nth}(\text{append}(p'_1, \text{cdr}(p'_2)), 1 + i'))$

Case splitting on $1 + i!1 = \text{length}(p!1)$,

we get 2 subgoals:

path_concat_path.2.2.1:

{-1}	$1 + i' = \text{length}(p'_1)$
{-2}	$i' < \text{length}(p'_1) + \text{length}(p'_2) - 2$
{-3}	$\text{length}(p'_1) \geq 1$
{-4}	$\forall (i: \text{below}(\text{length}(p'_1))): g' \text{ nodes}(\text{nth}(p'_1, i))$
{-5}	$\forall (i: \text{below}(\text{length}(p'_1) - 1)):$ $g' \text{ edges}(\text{nth}(p'_1, i), \text{nth}(p'_1, 1 + i))$
{-6}	$\text{length}(p'_2) \geq 1$
{-7}	$\forall (i: \text{below}(\text{length}(p'_2))): g' \text{ nodes}(\text{nth}(p'_2, i))$
{-8}	$\forall (i: \text{below}(\text{length}(p'_2) - 1)):$ $g' \text{ edges}(\text{nth}(p'_2, i), \text{nth}(p'_2, 1 + i))$
{-9}	$\text{concatable?}(p'_1, p'_2)$
{1}	$1 + i' < \text{length}(p'_1)$
{2}	$g' \text{ edges}$ $(\text{nth}(\text{append}(p'_1, \text{cdr}(p'_2)), i'), \text{nth}(\text{append}(p'_1, \text{cdr}(p'_2)), 1 + i'))$

Simplifying, rewriting, and recording with decision procedures,

Expanding the definition of concatable?.

C Proof scripts

Expanding the definition of path_end,
 Expanding the definition of path_start,
 Replacing using formula -1,
 Simplifying, rewriting, and recording with decision procedures,
 Replacing using formula -9,
 Instantiating the top quantifier in -8 with the terms: 0,
 Expanding the definition of nth,
 which is trivially true.
 This completes the proof of path_concat_path.2.2.1.
 path_concat_path.2.2.2:

{-1}	$i' < \text{length}(p'_1) + \text{length}(p'_2) - 2$
{-2}	$\text{length}(p'_1) \geq 1$
{-3}	$\forall (i: \text{below}(\text{length}(p'_1))): g' \text{ nodes}(\text{nth}(p'_1, i))$
{-4}	$\forall (i: \text{below}(\text{length}(p'_1) - 1)):$ $g' \text{ edges}(\text{nth}(p'_1, i), \text{nth}(p'_1, 1 + i))$
{-5}	$\text{length}(p'_2) \geq 1$
{-6}	$\forall (i: \text{below}(\text{length}(p'_2))): g' \text{ nodes}(\text{nth}(p'_2, i))$
{-7}	$\forall (i: \text{below}(\text{length}(p'_2) - 1)):$ $g' \text{ edges}(\text{nth}(p'_2, i), \text{nth}(p'_2, 1 + i))$
{-8}	$\text{concatable?}(p'_1, p'_2)$
{1}	$1 + i' = \text{length}(p'_1)$
{2}	$1 + i' < \text{length}(p'_1)$
{3}	$g' \text{ edges}$ $(\text{nth}(\text{append}(p'_1, \text{cdr}(p'_2)), i'), \text{nth}(\text{append}(p'_1, \text{cdr}(p'_2)), 1 + i'))$

Simplifying, rewriting, and recording with decision procedures,
 Instantiating the top quantifier in -7 with the terms: $i' - \text{length}(p'_1) + 1$,
 Expanding the definition of nth,
 which is trivially true.
 This completes the proof of path_concat_path.2.2.2.
 Q.E.D.

C.97.14 Graph.trans_closure_char_graph

Terse proof for trans_closure_char_graph.

trans_closure_char_graph:

{1}	$\forall (g: \text{graph_type}, x, y: (g' \text{ nodes})):$ $\text{transitive_closure}(g' \text{ edges})(x, y) \equiv$ $(\exists (l: (\text{path?}(g))):$ $\text{length}(l) \geq 2 \wedge \text{path_start}(l) = x \wedge \text{path_end}(l) = y)$
-----	---

Repeatedly Skolemizing and flattening,
 Using lemma trans_closure_char,
 Installing automatic rewrites from: length_stable nth_stable
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 we get 2 subgoals:

trans_closure_char_graph.1:

{-1}	transitive_closure(g' 'edges)(x' , y')
{-2}	$\exists (l: \text{list}[(g'$ 'nodes)]):
	length(l) $\geq 2 \wedge$
	path_start(l) = $x' \wedge$
	path_end(l) = $y' \wedge$
	$(\forall (i: \text{below}(\text{length}(l) - 1)):$
	g' 'edges(nth(l , i), nth(l , $1 + i$)))
{-3}	g' 'nodes(x')
{-4}	g' 'nodes(y')
{1}	$\exists (l: (\text{path?}(g'))): \text{length}(l) \geq 2 \wedge \text{path_start}(l) = x' \wedge \text{path_end}(l) = y'$

Keeping (-2 1) and hiding *,

Repeatedly Skolemizing and flattening,

Using lemma every_nth[T],

Simplifying, rewriting, and recording with decision procedures,

Installing automatic rewrites from: path? path_list?

Instantiating quantified variables,

we get 2 subgoals:

trans_closure_char_graph.1.1:

{-1}	$\forall (i: \text{below}(\text{length}(l'))): g'$ 'nodes(nth(l' , i))
{-2}	every(g' 'nodes)(l')
{-3}	length[T](l') ≥ 2
{-4}	path_start(l') = x'
{-5}	path_end(l') = y'
{-6}	$\forall (i: \text{below}(\text{length}(l') - 1)): g'$ 'edges(nth(l' , i), nth(l' , $1 + i$))
{1}	length(l') $\geq 2 \wedge \text{path_start}(l') = x' \wedge \text{path_end}(l') = y'$

Simplifying, rewriting, and recording with decision procedures,

Expanding the definition of path_start,

Expanding the definition of path_end,

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of trans_closure_char_graph.1.1.

trans_closure_char_graph.1.2:

{-1}	$\forall (i: \text{below}(\text{length}(l'))): g'$ 'nodes(nth(l' , i))
{-2}	every(g' 'nodes)(l')
{-3}	length[T](l') ≥ 2
{-4}	path_start(l') = x'
{-5}	path_end(l') = y'
{-6}	$\forall (i: \text{below}(\text{length}(l') - 1)): g'$ 'edges(nth(l' , i), nth(l' , $1 + i$))
{1}	$(\forall (i: \text{below}(\text{length}(l'))): g'$ 'nodes(nth(l' , i))) \wedge
	$(\forall (i: \text{below}(\text{length}(l') - 1)):$
	g' 'edges(nth(l' , i), nth(l' , $1 + i$)))

Applying propositional simplification,

Repeatedly Skolemizing and flattening,

Instantiating the top quantifier in -7 with the terms: i' ,

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of trans_closure_char_graph.1.2.

trans_closure_char_graph.2:

{-1}	$g' \text{ nodes}(x')$
{-2}	$g' \text{ nodes}(y')$
{-3}	$\exists (l: (\text{path?}(g'))): \text{length}(l) \geq 2 \wedge \text{path_start}(l) = x' \wedge \text{path_end}(l) = y'$
{1}	$\text{transitive_closure}(g' \text{ edges})(x', y')$
{2}	$\exists (l: \text{list}[(g' \text{ nodes})]):$ $\text{length}(l) \geq 2 \wedge$ $\text{path_start}(l) = x' \wedge$ $\text{path_end}(l) = y' \wedge$ $(\forall (i: \text{below}(\text{length}(l) - 1)):$ $g' \text{ edges}(\text{nth}(l, i), \text{nth}(l, 1 + i)))$

Keeping (-3 2) and hiding *,

Repeatedly Skolemizing and flattening,

Installing automatic rewrites from: path? path_list?

Simplifying, rewriting, and recording with decision procedures,

Applying disjunctive simplification to flatten sequent,

Expanding the definition of path_start,

Expanding the definition of path_end,

Using lemma every_nth[T],

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Instantiating quantified variables,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Repeatedly Skolemizing and flattening,

Instantiating the top quantifier in -5 with the terms: i' ,

we get 2 subgoals:

trans_closure_char_graph.2.1:

{-1}	$i' < \text{length}(l') - 1$
{-2}	$\text{every}(g' \text{ nodes})(l')$
{-3}	$\forall (i: \text{below}(\text{length}(l'))): g' \text{ nodes}(\text{nth}(l', i))$
{-4}	$\text{length}(l') \geq 1$
{-5}	$g' \text{ edges}(\text{nth}(l', i'), \text{nth}(l', 1 + i'))$
{-6}	$\text{car}(l') = x'$
{-7}	$\text{length}(l') \geq 2$
{-8}	$\text{nth}(l', \text{length}(l') - 1) = y'$
{1}	$g' \text{ edges}(\text{nth}(l', i'), \text{nth}(l', 1 + i'))$

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of trans_closure_char_graph.2.1.

trans_closure_char_graph.2.2:

{-1}	$i' < \text{length}(l') - 1$
{-2}	$\text{every}(g' \text{ nodes})(l')$
{-3}	$\forall (i: \text{below}(\text{length}(l'))): g' \text{ nodes}(\text{nth}(l', i))$
{-4}	$\text{length}(l') \geq 1$
{-5}	$\text{car}(l') = x'$
{-6}	$\text{length}(l') \geq 2$
{-7}	$\text{nth}(l', \text{length}(l') - 1) = y'$
{1}	$i' < \text{length}[T](l') - 1$
{2}	$g' \text{ edges}(\text{nth}(l', i'), \text{nth}(l', 1 + i'))$

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of trans_closure_char_graph.2.2.

Q.E.D.

C.97.15 Graph.path_cycle_free

Terse proof for path_cycle_free.

path_cycle_free:

{1}	$\forall (g: \text{graph_type}):$ $\text{cycle_free?}(g) \equiv$ $(\forall (l: (\text{path?}(g)), i_1, i_2: \text{below}(\text{length}(l))):$ $\text{nth}(l, i_1) = \text{nth}(l, i_2) \supset i_1 = i_2)$
-----	---

Repeatedly Skolemizing and flattening,

Installing automatic rewrites from: car_tail car_head nth_tail nth_head length_head length_tail path_list? path_head path_tail path_start path_end

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting, we get 2 subgoals:

path_cycle_free.1:

{-1}	cycle_free?(g')
{1}	$\forall (l: (\text{path?}(g')), i_1, i_2: \text{below}(\text{length}(l))):$ $\text{nth}(l, i_1) = \text{nth}(l, i_2) \supset i_1 = i_2$

Repeatedly Skolemizing and flattening,

Expanding the definition of cycle_free?,

Instantiating the top quantifier in -5 with the terms: nth(l', i'_1),

Applying propositional simplification,

we get 2 subgoals:

path_cycle_free.1.1:

{-1}	path_list?[T](l')
{-2}	path?(g')(l')
{-3}	i'_1 < length(l')
{-4}	i'_2 < length(l')
{-5}	nth(l', i'_1) = nth(l', i'_2)
{1}	transitive_closure(g'^edges)(nth(l', i'_1), nth(l', i'_1))
{2}	i'_1 = i'_2

Rewriting using trans_closure_char_graph, matching in *,

Case splitting on i1!1 < i2!1,

we get 2 subgoals:

path_cycle_free.1.1.1:

{-1}	i'_1 < i'_2
{-2}	path_list?[T](l')
{-3}	path?(g')(l')
{-4}	i'_1 < length(l')
{-5}	i'_2 < length(l')
{-6}	nth(l', i'_1) = nth(l', i'_2)
{1}	$\exists (l: (\text{path?}(g'))):$ $\text{length}(l) \geq 2 \wedge$ $\text{car}(l) = \text{nth}(l', i'_1) \wedge \text{nth}(l, \text{length}(l) - 1) = \text{nth}(l', i'_1)$
{2}	i'_1 = i'_2

Instantiating the top quantifier in 1 with the terms: tail(head(l', i'_2 + 1), i'_1),

C Proof scripts

we get 3 subgoals:

`path_cycle_free.1.1.1.1:`

<ul style="list-style-type: none"> {-1} $i'_1 < i'_2$ {-2} <code>path_list?[T](l')</code> {-3} <code>path?(g')(l')</code> {-4} $i'_1 < \text{length}(l')$ {-5} $i'_2 < \text{length}(l')$ {-6} $\text{nth}(l', i'_1) = \text{nth}(l', i'_2)$ 	<hr style="border: 0.5px solid black;"/> <ul style="list-style-type: none"> {1} $\text{length}(\text{tail}(\text{head}(l', i'_2 + 1), i'_1)) \geq 2 \wedge$ $\text{car}(\text{tail}(\text{head}(l', i'_2 + 1), i'_1)) = \text{nth}(l', i'_1) \wedge$ $\text{nth}(\text{tail}(\text{head}(l', i'_2 + 1), i'_1),$ $\quad \text{length}(\text{tail}(\text{head}(l', i'_2 + 1), i'_1)) - 1)$ $= \text{nth}(l', i'_1)$ {2} $i'_1 = i'_2$
---	---

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `path_cycle_free.1.1.1.1`.

`path_cycle_free.1.1.1.2:`

<ul style="list-style-type: none"> {-1} $i'_1 < i'_2$ {-2} <code>path_list?[T](l')</code> {-3} <code>path?(g')(l')</code> {-4} $i'_1 < \text{length}(l')$ {-5} $i'_2 < \text{length}(l')$ {-6} $\text{nth}(l', i'_1) = \text{nth}(l', i'_2)$ 	<hr style="border: 0.5px solid black;"/> <ul style="list-style-type: none"> {1} $\text{length}(\text{tail}[T](\text{head}[T](l', 1 + i'_2), i'_1)) \geq 1 \wedge$ $\text{path?}(g')(\text{tail}[T](\text{head}[T](l', 1 + i'_2), i'_1))$ {2} $i'_1 = i'_2$
---	--

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `path_cycle_free.1.1.1.2`.

`path_cycle_free.1.1.1.3:`

<ul style="list-style-type: none"> {-1} $i'_1 < i'_2$ {-2} <code>path_list?[T](l')</code> {-3} <code>path?(g')(l')</code> {-4} $i'_1 < \text{length}(l')$ {-5} $i'_2 < \text{length}(l')$ {-6} $\text{nth}(l', i'_1) = \text{nth}(l', i'_2)$ 	<hr style="border: 0.5px solid black;"/> <ul style="list-style-type: none"> {1} $i'_1 \leq 1 + i'_2$ {2} $i'_1 = i'_2$
---	--

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `path_cycle_free.1.1.1.3`.

path_cycle_free.1.1.2:

{-1}	path_list?[T](l')
{-2}	path?(g')(l')
{-3}	i'_1 < length(l')
{-4}	i'_2 < length(l')
{-5}	nth(l', i'_1) = nth(l', i'_2)
{1}	i'_1 < i'_2
{2}	$\exists (l: (\text{path?}(g'))):$ length(l) ≥ 2 ∧ car(l) = nth(l', i'_1) ∧ nth(l, length(l) - 1) = nth(l', i'_1)
{3}	i'_1 = i'_2

Instantiating the top quantifier in 2 with the terms: tail(head(l', i'_1 + 1), i'_2), we get 3 subgoals:

path_cycle_free.1.1.2.1:

{-1}	path_list?[T](l')
{-2}	path?(g')(l')
{-3}	i'_1 < length(l')
{-4}	i'_2 < length(l')
{-5}	nth(l', i'_1) = nth(l', i'_2)
{1}	i'_1 < i'_2
{2}	length(tail(head(l', i'_1 + 1), i'_2)) ≥ 2 ∧ car(tail(head(l', i'_1 + 1), i'_2)) = nth(l', i'_1) ∧ nth(tail(head(l', i'_1 + 1), i'_2), length(tail(head(l', i'_1 + 1), i'_2)) - 1) = nth(l', i'_1)
{3}	i'_1 = i'_2

Simplifying, rewriting, and recording with decision procedures,
This completes the proof of path_cycle_free.1.1.2.1.

path_cycle_free.1.1.2.2:

{-1}	path_list?[T](l')
{-2}	path?(g')(l')
{-3}	i'_1 < length(l')
{-4}	i'_2 < length(l')
{-5}	nth(l', i'_1) = nth(l', i'_2)
{1}	length(tail[T](head[T](l', 1 + i'_1), i'_2)) ≥ 1 ∧ path?(g')(tail[T](head[T](l', 1 + i'_1), i'_2))
{2}	i'_1 < i'_2
{3}	i'_1 = i'_2

Simplifying, rewriting, and recording with decision procedures,
This completes the proof of path_cycle_free.1.1.2.2.

path_cycle_free.1.1.2.3:

{-1}	path_list?[T](l')
{-2}	path?(g')(l')
{-3}	i'_1 < length(l')
{-4}	i'_2 < length(l')
{-5}	nth(l', i'_1) = nth(l', i'_2)
{1}	i'_2 ≤ 1 + i'_1
{2}	i'_1 < i'_2
{3}	i'_1 = i'_2

C Proof scripts

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `path_cycle_free.1.1.2.3`.

`path_cycle_free.1.2`:

{-1}	$\text{path_list?}[T](l')$
{-2}	$\text{path?}(g')(l')$
{-3}	$i'_1 < \text{length}(l')$
{-4}	$i'_2 < \text{length}(l')$
{-5}	$\text{nth}(l', i'_1) = \text{nth}(l', i'_2)$
{1}	$g' \text{'nodes}(\text{nth}(l', i'_1))$
{2}	$i'_1 = i'_2$

Expanding the definition of `path?`,

Applying disjunctive simplification to flatten sequent,

Instantiating quantified variables,

This completes the proof of `path_cycle_free.1.2`.

`path_cycle_free.2`:

{-1}	$\forall (l: (\text{path?}(g')), i_1, i_2: \text{below}(\text{length}(l))):$ $\text{nth}(l, i_1) = \text{nth}(l, i_2) \supset i_1 = i_2$
{1}	$\text{cycle_free?}(g')$

Expanding the definition of `cycle_free?`,

Repeatedly Skolemizing and flattening,

Rewriting using `trans_closure_char_graph`, matching in `*`,

Repeatedly Skolemizing and flattening,

Instantiating the top quantifier in -7 with the terms: $l', 0, \text{length}(l') - 1$,

we get 3 subgoals:

`path_cycle_free.2.1`:

{-1}	$\text{path_list?}[T](l')$
{-2}	$\text{path?}(g')(l')$
{-3}	$g' \text{'nodes}(x')$
{-4}	$\text{length}(l') \geq 2$
{-5}	$\text{car}(l') = x'$
{-6}	$\text{nth}(l', \text{length}(l') - 1) = x'$
{-7}	$\text{nth}(l', 0) = \text{nth}(l', \text{length}(l') - 1) \supset 0 = \text{length}(l') - 1$

Simplifying, rewriting, and recording with decision procedures,

Expanding the definition of `nth`,

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `path_cycle_free.2.1`.

`path_cycle_free.2.2`:

{-1}	$\text{path_list?}[T](l')$
{-2}	$\text{path?}(g')(l')$
{-3}	$g' \text{'nodes}(x')$
{-4}	$\text{length}(l') \geq 2$
{-5}	$\text{car}(l') = x'$
{-6}	$\text{nth}(l', \text{length}(l') - 1) = x'$
{1}	$\text{length}[T](l') - 1 \geq 0$

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `path_cycle_free.2.2`.

path_cycle_free.2.3:

{-1}	path_list?[T](l')
{-2}	path?(g')(l')
{-3}	g'nodes(x')
{-4}	length(l') ≥ 2
{-5}	car(l') = x'
{-6}	nth(l', length(l') - 1) = x'
{1}	0 < length[T](l')

Simplifying, rewriting, and recording with decision procedures,
This completes the proof of path_cycle_free.2.3.
Q.E.D.

C.97.16 Graph.different_nodes_on_edge

Terse proof for different_nodes_on_edge.

different_nodes_on_edge:

{1}	$\forall (g: \text{graph_type}, x, y: (g' \text{nodes})): \text{cycle_free?}(g) \wedge g' \text{edges}(x, y) \supset \neg x = y$
-----	--

Repeatedly Skolemizing and flattening,

Using lemma path_cycle_free,

Simplifying, rewriting, and recording with decision procedures,

Installing automatic rewrites from: nth length path_list? path?

Instantiating the top quantifier in -1 with the terms: (:x', y':), 0, 1,
we get 2 subgoals:

different_nodes_on_edge.1:

{-1}	nth((:x', y':), 0) = nth((:x', y':), 1) \supset 0 = 1
{-2}	g'nodes(x')
{-3}	g'nodes(y')
{-4}	cycle_free?(g')
{-5}	g'edges(x', y')
{-6}	x' = y'

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of different_nodes_on_edge.1.

different_nodes_on_edge.2:

{-1}	g'nodes(x')
{-2}	g'nodes(y')
{-3}	cycle_free?(g')
{-4}	g'edges(x', y')
{-5}	x' = y'
{1}	$(\forall (i: \text{below}(\text{length}((:x', y':)))): g' \text{nodes}(\text{nth}((:x', y':), i))) \wedge$ $(\forall (i: \text{below}(\text{length}((:x', y':)) - 1)): g' \text{edges}(\text{nth}((:x', y':), i), \text{nth}((:y':), i)))$

Repeatedly simplifying with decision procedures, rewriting, propositional reasoning, quantifier instantiation, skolemization, if-lifting and equality replacement,

Case splitting on i!1 = 0,

we get 2 subgoals:

C Proof scripts

`different_nodes_on_edge.2.1:`

{-1}	$i' = 0$
{-2}	$i' < 2$
{-3}	$g' \text{ nodes}(y')$
{-4}	$\text{cycle_free?}(g')$
{-5}	$g' \text{ edges}(y', y')$
{-6}	$x' = y'$
{1}	$g' \text{ nodes}(\text{nth}(:y', y':), i')$

Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `different_nodes_on_edge.2.1`.

`different_nodes_on_edge.2.2:`

{-1}	$i' < 2$
{-2}	$g' \text{ nodes}(y')$
{-3}	$\text{cycle_free?}(g')$
{-4}	$g' \text{ edges}(y', y')$
{-5}	$x' = y'$
{1}	$i' = 0$
{2}	$g' \text{ nodes}(\text{nth}(:y', y':), i')$

Case splitting on $i!1 = 1$,
we get 2 subgoals:

`different_nodes_on_edge.2.2.1:`

{-1}	$i' = 1$
{-2}	$i' < 2$
{-3}	$g' \text{ nodes}(y')$
{-4}	$\text{cycle_free?}(g')$
{-5}	$g' \text{ edges}(y', y')$
{-6}	$x' = y'$
{1}	$i' = 0$
{2}	$g' \text{ nodes}(\text{nth}(:y', y':), i')$

Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `different_nodes_on_edge.2.2.1`.

`different_nodes_on_edge.2.2.2:`

{-1}	$i' < 2$
{-2}	$g' \text{ nodes}(y')$
{-3}	$\text{cycle_free?}(g')$
{-4}	$g' \text{ edges}(y', y')$
{-5}	$x' = y'$
{1}	$i' = 1$
{2}	$i' = 0$
{3}	$g' \text{ nodes}(\text{nth}(:y', y':), i')$

Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `different_nodes_on_edge.2.2.2`.
Q.E.D.

C.97.17 Graph.tree_path

Terse proof for `tree_path`.

tree_path:

{1} $\forall (g: \text{graph_type}, p_1, p_2: (\text{path?}(g))):$
 $\text{tree?}(g) \wedge \text{path_end}(p_1) = \text{path_end}(p_2) \wedge \text{length}(p_1) = \text{length}(p_2) \supset p_1 = p_2$

For the top quantifier in 1, we introduce Skolem constants: $(g' \text{ ---})$,
 Inducting on p1 on formula 1 using induction scheme `path_list_induction`[T],
 we get 3 subgoals:

tree_path.1:

{1} $\text{path?}(g')(p'_1)$
 {2} $\forall (p_2: (\text{path?}(g'))):$
 $\text{tree?}(g') \wedge \text{path_end}(p'_1) = \text{path_end}(p_2) \wedge \text{length}(p'_1) = \text{length}(p_2) \supset p'_1 = p_2$

Simplifying, rewriting, and recording with decision procedures,
 This completes the proof of **tree_path.1**.

tree_path.2:

{1} $\forall (t: T):$
 $\text{path?}(g')(\text{cons}(t, \text{null})) \supset$
 $(\forall (p_2: (\text{path?}(g'))):$
 $\text{tree?}(g') \wedge$
 $\text{path_end}(\text{cons}(t, \text{null})) = \text{path_end}(p_2) \wedge$
 $\text{length}(\text{cons}(t, \text{null})) = \text{length}(p_2)$
 $\supset \text{cons}(t, \text{null}) = p_2)$

Repeatedly Skolemizing and flattening,
 Installing automatic rewrites from: `length path_end nth`
 Simplifying, rewriting, and recording with decision procedures,
 Simplifying, rewriting, and recording with decision procedures,
 Applying extensionality,
 Expanding the definition of `length`,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 This completes the proof of **tree_path.2**.

tree_path.3:

{1} $\forall (t: T, \text{tail}: (\text{path_list?})):$
 $(\text{path?}(g')(\text{tail}) \supset$
 $(\forall (p_2: (\text{path?}(g'))):$
 $\text{tree?}(g') \wedge \text{path_end}(\text{tail}) = \text{path_end}(p_2) \wedge \text{length}(\text{tail}) = \text{length}(p_2) \supset$
 $\text{tail} = p_2))$
 \supset
 $\text{path?}(g')(\text{cons}(t, \text{tail})) \supset$
 $(\forall (p_2: (\text{path?}(g'))):$
 $\text{tree?}(g') \wedge$
 $\text{path_end}(\text{cons}(t, \text{tail})) = \text{path_end}(p_2) \wedge$
 $\text{length}(\text{cons}(t, \text{tail})) = \text{length}(p_2)$
 $\supset \text{cons}(t, \text{tail}) = p_2)$

Repeatedly Skolemizing and flattening,
 Using lemma `path_cdr`,
 Simplifying, rewriting, and recording with decision procedures,

C Proof scripts

Expanding the definition of length,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Instantiating the top quantifier in -5 with the terms: $\text{cdr}(p'_2)$,

we get 2 subgoals:

tree_path.3.1:

{-1}	$\text{path?}(g')(\text{tail}')$
{-2}	$\text{path_list?}[T](p'_2)$
{-3}	$\text{path?}(g')(p'_2)$
{-4}	$\text{path_list?}(\text{tail}')$
{-5}	$\text{path_end}(\text{tail}') = \text{path_end}(\text{cdr}(p'_2)) \wedge \text{length}(\text{tail}') = \text{length}(\text{cdr}(p'_2)) \supset$ $\text{tail}' = \text{cdr}(p'_2)$
{-6}	$\text{path?}(g')(\text{cons}(t', \text{tail}'))$
{-7}	$\text{tree?}(g')$
{-8}	$\text{path_end}(\text{cons}(t', \text{tail}')) = \text{path_end}(p'_2)$
{-9}	$\text{length}(\text{tail}') = \text{length}(\text{cdr}(p'_2))$
{1}	$\text{null?}(p'_2)$
{2}	$\text{cons}(t', \text{tail}') = p'_2$

Expanding the definition of path_list?,

Applying propositional simplification,

we get 2 subgoals:

tree_path.3.1.1:

{-1}	$\text{tail}' = \text{cdr}(p'_2)$
{-2}	$\text{path?}(g')(\text{tail}')$
{-3}	$\text{length}(p'_2) \geq 1$
{-4}	$\text{path?}(g')(p'_2)$
{-5}	$\text{length}(\text{tail}') \geq 1$
{-6}	$\text{path?}(g')(\text{cons}(t', \text{tail}'))$
{-7}	$\text{tree?}(g')$
{-8}	$\text{path_end}(\text{cons}(t', \text{tail}')) = \text{path_end}(p'_2)$
{-9}	$\text{length}(\text{tail}') = \text{length}(\text{cdr}(p'_2))$
{1}	$\text{null?}(p'_2)$
{2}	$\text{cons}(t', \text{tail}') = p'_2$

Applying extensionality,

Hiding formulas: -2,

Expanding the definition of path?,

Applying disjunctive simplification to flatten sequent,

Instantiating the top quantifier in -4 with the terms: 0,

we get 2 subgoals:

tree_path.3.1.1.1.1:

{-1}	tail' = cdr(p ₂)
{-2}	length(p ₂) ≥ 1
{-3}	∀ (i: below(length(p ₂))): g' nodes(nth(p ₂ , i))
{-4}	g' edges(nth(p ₂ , 0), nth(p ₂ , 1 + 0))
{-5}	length(tail') ≥ 1
{-6}	∀ (i: below(length(cons(t', tail')))): g' nodes(nth(cons(t', tail'), i))
{-7}	∀ (i: below(length(cons(t', tail')) - 1)): g' edges(nth(cons(t', tail'), i), nth(cons(t', tail'), 1 + i))
{-8}	tree?(g')
{-9}	path_end(cons(t', tail')) = path_end(p ₂)
{-10}	length(tail') = length(cdr(p ₂))
{1}	t' = car(p ₂)
{2}	null?(p ₂)

Instantiating the top quantifier in -7 with the terms: 0,
we get 2 subgoals:

tree_path.3.1.1.1.1.1:

{-1}	tail' = cdr(p ₂)
{-2}	length(p ₂) ≥ 1
{-3}	∀ (i: below(length(p ₂))): g' nodes(nth(p ₂ , i))
{-4}	g' edges(nth(p ₂ , 0), nth(p ₂ , 1 + 0))
{-5}	length(tail') ≥ 1
{-6}	∀ (i: below(length(cons(t', tail')))): g' nodes(nth(cons(t', tail'), i))
{-7}	g' edges(nth(cons(t', tail'), 0), nth(cons(t', tail'), 1 + 0))
{-8}	tree?(g')
{-9}	path_end(cons(t', tail')) = path_end(p ₂)
{-10}	length(tail') = length(cdr(p ₂))
{1}	t' = car(p ₂)
{2}	null?(p ₂)

Installing automatic rewrites from: nth
Simplifying, rewriting, and recording with decision procedures,
Expanding the definition of tree?,
Applying disjunctive simplification to flatten sequent,
Instantiating the top quantifier in -9 with the terms: t', car(p₂), car(tail'),
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of tree_path.3.1.1.1.1.

tree_path.3.1.1.1.1.2:

{-1}	tail' = cdr(p ₂)
{-2}	length(p ₂) ≥ 1
{-3}	∀ (i: below(length(p ₂))): g' nodes(nth(p ₂ , i))
{-4}	g' edges(nth(p ₂ , 0), nth(p ₂ , 1 + 0))
{-5}	length(tail') ≥ 1
{-6}	∀ (i: below(length(cons(t', tail')))): g' nodes(nth(cons(t', tail'), i))
{-7}	tree?(g')
{-8}	path_end(cons(t', tail')) = path_end(p ₂)
{-9}	length(tail') = length(cdr(p ₂))
{1}	0 < length[T](cons[T](t', tail')) - 1
{2}	t' = car(p ₂)
{3}	null?(p ₂)

C Proof scripts

Expanding the definition of length,
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `tree_path.3.1.1.1.2`.

`tree_path.3.1.1.2`:

{-1}	$\text{tail}' = \text{cdr}(p'_2)$
{-2}	$\text{length}(p'_2) \geq 1$
{-3}	$\forall (i : \text{below}(\text{length}(p'_2))) : g' \text{ nodes}(\text{nth}(p'_2, i))$
{-4}	$\text{length}(\text{tail}') \geq 1$
{-5}	$\forall (i : \text{below}(\text{length}(\text{cons}(t', \text{tail}')))) : g' \text{ nodes}(\text{nth}(\text{cons}(t', \text{tail}'), i))$
{-6}	$\forall (i : \text{below}(\text{length}(\text{cons}(t', \text{tail}')) - 1)) :$ $g' \text{ edges}(\text{nth}(\text{cons}(t', \text{tail}'), i), \text{nth}(\text{cons}(t', \text{tail}'), 1 + i))$
{-7}	$\text{tree?}(g')$
{-8}	$\text{path_end}(\text{cons}(t', \text{tail}')) = \text{path_end}(p'_2)$
{-9}	$\text{length}(\text{tail}') = \text{length}(\text{cdr}(p'_2))$
{1}	$0 < \text{length}[T](p'_2) - 1$
{2}	$t' = \text{car}(p'_2)$
{3}	$\text{null?}(p'_2)$

Expanding the definition of length,
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `tree_path.3.1.1.2`.

`tree_path.3.1.2`:

{-1}	$\text{path?}(g')(\text{tail}')$
{-2}	$\text{length}(p'_2) \geq 1$
{-3}	$\text{path?}(g')(p'_2)$
{-4}	$\text{length}(\text{tail}') \geq 1$
{-5}	$\text{path?}(g')(\text{cons}(t', \text{tail}'))$
{-6}	$\text{tree?}(g')$
{-7}	$\text{path_end}(\text{cons}(t', \text{tail}')) = \text{path_end}(p'_2)$
{-8}	$\text{length}(\text{tail}') = \text{length}(\text{cdr}(p'_2))$
{1}	$\text{path_end}(\text{tail}') = \text{path_end}(\text{cdr}(p'_2))$
{2}	$\text{null?}(p'_2)$
{3}	$\text{cons}(t', \text{tail}') = p'_2$

Installing automatic rewrites from: `path_end nth length`
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `tree_path.3.1.2`.

`tree_path.3.2`:

{-1}	$\text{path?}(g')(\text{tail}')$
{-2}	$\text{path_list?}[T](p'_2)$
{-3}	$\text{path?}(g')(p'_2)$
{-4}	$\text{path_list?}(\text{tail}')$
{-5}	$\text{path?}(g')(\text{cons}(t', \text{tail}'))$
{-6}	$\text{tree?}(g')$
{-7}	$\text{path_end}(\text{cons}(t', \text{tail}')) = \text{path_end}(p'_2)$
{-8}	$\text{length}(\text{tail}') = \text{length}(\text{cdr}(p'_2))$
{1}	$\text{path_list?}[T](\text{cdr}[T](p'_2)) \wedge \text{path?}(g')(\text{cdr}[T](p'_2))$
{2}	$\text{null?}(p'_2)$
{3}	$\text{cons}(t', \text{tail}') = p'_2$

Using lemma `path_cdr`,
Expanding the definition of `path_list?`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 This completes the proof of `tree_path.3.2`.
 Q.E.D.

C.97.18 Graph.root_path_TCC1

Terse proof for `root_path_TCC1`.

`root_path_TCC1`:

{1} $\forall (g: \text{graph_type}, n: (g' \text{nodes}), p: (\text{path?}(g))): (g' \text{nodes})(\text{path_start}[T](p))$

Repeatedly Skolemizing and flattening,
 Rewriting using `nodes_path_start`, matching in *,
 This completes the proof of `root_path_TCC1`.
 Q.E.D.

C.97.19 Graph.path_length_bound_TCC1

Terse proof for `path_length_bound_TCC1`.

`path_length_bound_TCC1`:

{1} $\forall (g: \text{graph_type}):$
 $\text{cycle_free?}(g) \wedge \text{finite?}(g) \supset (\forall (p: (\text{path?}(g))): \text{is_finite}[T](g' \text{nodes}))$

Repeatedly Skolemizing and flattening,
 Expanding the definition of `finite?`,
 which is trivially true.
 This completes the proof of `path_length_bound_TCC1`.
 Q.E.D.

C.97.20 Graph.path_length_bound

Terse proof for `path_length_bound`.

`path_length_bound`:

{1} $\forall (g: \text{graph_type}):$
 $\text{finite?}(g) \wedge \text{cycle_free?}(g) \supset$
 $(\forall (p: (\text{path?}(g))): \text{length}(p) \leq \text{card}(g' \text{nodes}))$

Repeatedly Skolemizing and flattening,
 Using lemma `list_pred_card[T]`,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 we get 3 subgoals:

`path_length_bound.1`:

{-1} $\exists (n_1, n_2: \text{below}(\text{length}(p'))): \neg n_1 = n_2 \wedge \text{nth}(p', n_1) = \text{nth}(p', n_2)$
 {-2} $\text{path_list?}[T](p')$
 {-3} $\text{path?}(g')(p')$
 {-4} $\text{finite?}(g')$
 {-5} $\text{cycle_free?}(g')$

 {1} $\text{length}(p') \leq \text{card}(g' \text{nodes})$

C Proof scripts

Repeatedly Skolemizing and flattening,
 Using lemma `path_cycle_free`,
 Simplifying, rewriting, and recording with decision procedures,
 Instantiating quantified variables,
 Simplifying, rewriting, and recording with decision procedures,
 This completes the proof of `path_length_bound.1`.

`path_length_bound.2`:

{-1}	<code>path_list?[T](p')</code>
{-2}	<code>path?(g')(p')</code>
{-3}	<code>finite?(g')</code>
{-4}	<code>cycle_free?(g')</code>
{1}	<code>every(g'nodes)(p')</code>
{2}	<code>length(p') ≤ card(g'nodes)</code>

Rewriting using `every_nth`, matching in `*`,
 Expanding the definition of `path?`,
 Applying disjunctive simplification to flatten sequent,
 This completes the proof of `path_length_bound.2`.

`path_length_bound.3`:

{-1}	<code>path_list?[T](p')</code>
{-2}	<code>path?(g')(p')</code>
{-3}	<code>finite?(g')</code>
{-4}	<code>cycle_free?(g')</code>
{1}	<code>is_finite(g'nodes)</code>
{2}	<code>length(p') ≤ card(g'nodes)</code>

Expanding the definition of `finite?`,
 which is trivially true.
 This completes the proof of `path_length_bound.3`.
 Q.E.D.

C.97.21 Graph.non_root_path

Terse proof for `non_root_path`.

`non_root_path`:

{1}	$\forall (g: \text{graph_type}, n: (g'nodes)):$ $\text{roots}(g)(n) \vee (\exists (p: (\text{path?}(g))): \text{length}(p) = 2 \wedge \text{path_end}(p) = n)$

Repeatedly Skolemizing and flattening,
 Expanding the definition of `roots`,
 Repeatedly Skolemizing and flattening,
 Case splitting on `length[T]((: x!1, n!1 :)) = 2`,
 we get 2 subgoals:

`non_root_path.1`:

{-1}	<code>length[T]((:x', n':)) = 2</code>
{-2}	<code>g'nodes(x')</code>
{-3}	<code>g'nodes(n')</code>
{-4}	<code>g'edges(x', n')</code>
{1}	$\exists (p: (\text{path?}(g'))): \text{length}(p) = 2 \wedge \text{path_end}(p) = n'$

Instantiating the top quantifier in 1 with the terms: `(:x', n':)`,

we get 2 subgoals:

non_root_path.1.1:

$$\left| \begin{array}{l} \{-1\} \text{ length}[T]((:x', n':)) = 2 \\ \{-2\} g' \text{ nodes}(x') \\ \{-3\} g' \text{ nodes}(n') \\ \{-4\} g' \text{ edges}(x', n') \end{array} \right. \hline \{1\} \text{ length}((:x', n':)) = 2 \wedge \text{ path_end}((:x', n':)) = n'$$

Simplifying, rewriting, and recording with decision procedures,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of **non_root_path.1.1**.

non_root_path.1.2:

$$\left| \begin{array}{l} \{-1\} \text{ length}[T]((:x', n':)) = 2 \\ \{-2\} g' \text{ nodes}(x') \\ \{-3\} g' \text{ nodes}(n') \\ \{-4\} g' \text{ edges}(x', n') \end{array} \right. \hline \{1\} \text{ path_list?}[T]((:x', n':)) \wedge \text{ path?}(g')((:x', n':))$$

Expanding the definition of `path_list?`,

Simplifying, rewriting, and recording with decision procedures,

Expanding the definition of `path?`,

Applying propositional simplification,

we get 2 subgoals:

non_root_path.1.2.1:

$$\left| \begin{array}{l} \{-1\} \text{ length}[T]((:x', n':)) = 2 \\ \{-2\} g' \text{ nodes}(x') \\ \{-3\} g' \text{ nodes}(n') \\ \{-4\} g' \text{ edges}(x', n') \end{array} \right. \hline \{1\} \forall (i: \text{ below}(\text{length}((:x', n':)))): g' \text{ nodes}(\text{nth}((:x', n':), i))$$

Repeatedly Skolemizing and flattening,

Case splitting on `i!1 = 0`,

we get 2 subgoals:

non_root_path.1.2.1.1:

$$\left| \begin{array}{l} \{-1\} i' = 0 \\ \{-2\} i' < \text{ length}((:x', n':)) \\ \{-3\} \text{ length}[T]((:x', n':)) = 2 \\ \{-4\} g' \text{ nodes}(x') \\ \{-5\} g' \text{ nodes}(n') \\ \{-6\} g' \text{ edges}(x', n') \end{array} \right. \hline \{1\} g' \text{ nodes}(\text{nth}((:x', n':), i'))$$

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of **non_root_path.1.2.1.1**.

C Proof scripts

`non_root_path.1.2.1.2:`

{-1}	$i' < \text{length}((:x', n':))$
{-2}	$\text{length}[T]((:x', n':)) = 2$
{-3}	$g' \text{ nodes}(x')$
{-4}	$g' \text{ nodes}(n')$
{-5}	$g' \text{ edges}(x', n')$
<hr/>	
{1}	$i' = 0$
{2}	$g' \text{ nodes}(\text{nth}((:x', n':), i'))$

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `non_root_path.1.2.1.2`.

`non_root_path.1.2.2:`

{-1}	$\text{length}[T]((:x', n':)) = 2$
{-2}	$g' \text{ nodes}(x')$
{-3}	$g' \text{ nodes}(n')$
{-4}	$g' \text{ edges}(x', n')$
<hr/>	
{1}	$\forall (i: \text{below}(\text{length}((:x', n':)) - 1)):$ $g' \text{ edges}(\text{nth}((:x', n':), i), \text{nth}((:x', n':), 1 + i))$

Repeatedly Skolemizing and flattening,
Expanding the definition of `nth`,
Expanding the definition of `nth`,
which is trivially true.

This completes the proof of `non_root_path.1.2.2`.

`non_root_path.2:`

{-1}	$g' \text{ nodes}(x')$
{-2}	$g' \text{ nodes}(n')$
{-3}	$g' \text{ edges}(x', n')$
<hr/>	
{1}	$\text{length}[T]((:x', n':)) = 2$
{2}	$\exists (p: (\text{path?}(g'))): \text{length}(p) = 2 \wedge \text{path_end}(p) = n'$

Hiding formulas: 2,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `non_root_path.2`.

Q.E.D.

C.97.22 Graph.non_root_path_extend_TCC1

Terse proof for `non_root_path_extend_TCC1`.

`non_root_path_extend_TCC1:`

{1}	$\forall (g: \text{graph_type}, p: (\text{path?}(g))): (g' \text{ nodes})(\text{path_start}[T](p))$
-----	---

Repeatedly Skolemizing and flattening,

Rewriting using `nodes_path_start`, matching in `*`,

This completes the proof of `non_root_path_extend_TCC1`.

Q.E.D.

C.97.23 Graph.non_root_path_extend

Terse proof for `non_root_path_extend`.

non_root_path_extend:

$\{1\} \quad \forall (g: \text{graph_type}, p: (\text{path?}(g))):$ $\quad \text{roots}(g)(\text{path_start}(p)) \vee$ $\quad (\exists (q: (\text{path?}(g))):$ $\quad \quad \text{length}(q) = \text{length}(p) + 1 \wedge \text{path_end}(q) = \text{path_end}(p))$

Repeatedly Skolemizing and flattening,

Using lemma non_root_path,

Simplifying, rewriting, and recording with decision procedures,

Repeatedly Skolemizing and flattening,

Installing automatic rewrites from: path_concat_path length_concat_path concatable?

Instantiating the top quantifier in 2 with the terms: concat_path(p'' , p'),

we get 2 subgoals:

non_root_path_extend.1:

$\{-1\} \quad \text{path_list?}[T](p'')$ $\{-2\} \quad \text{path?}(g')(p'')$ $\{-3\} \quad \text{length}(p'') = 2$ $\{-4\} \quad \text{path_end}(p'') = \text{path_start}(p')$ $\{-5\} \quad \text{path_list?}[T](p')$ $\{-6\} \quad \text{path?}(g')(p')$
$\{1\} \quad \text{roots}(g')(\text{path_start}(p'))$ $\{2\} \quad \text{length}(\text{concat_path}(p'', p')) = 1 + \text{length}(p') \wedge$ $\quad \text{path_end}(\text{concat_path}(p'', p')) = \text{path_end}(p')$

Simplifying, rewriting, and recording with decision procedures,

Installing automatic rewrites from: path_end path_start nth_append_right nth_append_left concat_path length_append length_cdr path_list? length

Simplifying, rewriting, and recording with decision procedures,

Case splitting on $\text{length}(p!1) = 1$,

we get 2 subgoals:

non_root_path_extend.1.1:

$\{-1\} \quad \text{length}(p') = 1$ $\{-2\} \quad \text{path_list?}[T](p'')$ $\{-3\} \quad \text{path?}(g')(p'')$ $\{-4\} \quad \text{length}(p'') = 2$ $\{-5\} \quad \text{nth}(p'', \text{length}(p'') - 1) = \text{car}(p')$ $\{-6\} \quad \text{length}(p') \geq 1$ $\{-7\} \quad \text{path?}(g')(p')$
$\{1\} \quad \text{roots}(g')(\text{car}(p'))$ $\{2\} \quad \text{nth}(\text{append}(p'', \text{cdr}(p'')), \text{length}(p') + \text{length}(p'') - 2) =$ $\quad \text{nth}(p', \text{length}(p') - 1)$

Simplifying, rewriting, and recording with decision procedures,

Installing automatic rewrites from: nth

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of non_root_path_extend.1.1.

C Proof scripts

`non_root_path_extend.1.2:`

{-1}	<code>path_list?[T](p'')</code>
{-2}	<code>path?(g')(p'')</code>
{-3}	<code>length(p'') = 2</code>
{-4}	<code>nth(p'', length(p'') - 1) = car(p')</code>
{-5}	<code>length(p') ≥ 1</code>
{-6}	<code>path?(g')(p')</code>
{1}	<code>length(p') = 1</code>
{2}	<code>roots(g')(car(p'))</code>
{3}	<code>nth(append(p'', cdr(p')), length(p') + length(p'') - 2) = nth(p', length(p') - 1)</code>

Simplifying, rewriting, and recording with decision procedures,

Installing automatic rewrites from: `nth`

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `non_root_path_extend.1.2`.

`non_root_path_extend.2:`

{-1}	<code>path_list?[T](p'')</code>
{-2}	<code>path?(g')(p'')</code>
{-3}	<code>length(p'') = 2</code>
{-4}	<code>path_end(p'') = path_start(p')</code>
{-5}	<code>path_list?[T](p')</code>
{-6}	<code>path?(g')(p')</code>
{1}	<code>path?(g')(concat_path[T](p'', p'))</code>
{2}	<code>roots(g')(path_start(p'))</code>

Simplifying, rewriting, and recording with decision procedures,

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `non_root_path_extend.2`.

Q.E.D.

C.97.24 Graph.tree_path_to_root

Terse proof for `tree_path_to_root`.

`tree_path_to_root:`

{1}	$\forall (g: \text{graph_type}):$ $\text{finite_tree?}(g) \supset$ $(\forall (n: (g'\text{nodes})): \exists (p: (\text{path?}(g))): \text{root_path}(g, n)(p))$
-----	--

Repeatedly Skolemizing and flattening,

Expanding the definition of `finite_tree?`,

Applying disjunctive simplification to flatten sequent,

Simplifying, rewriting, and recording with decision procedures,

Expanding the definition of `finite?`,

Simplifying, rewriting, and recording with decision procedures,

Letting `max_len` name `min(LAMBDA (i: nat): EXISTS (q: (path?(g!1))): path_end(q) = n!1 AND i = card(g!1'nodes) - length(q))`,

we get 2 subgoals:

tree_path_to_root.1:

{-1}	$\min(\lambda (i: \text{nat}):$ $\quad \exists (q: (\text{path?}(g'))):$ $\quad \text{path_end}(q) = n' \wedge i = \text{card}(g'\text{'nodes}) - \text{length}(q)$
{-2}	$= \text{max_len}$
{-3}	$g'\text{'nodes}(n')$
{-4}	$\text{is_finite}(g'\text{'nodes})$
{-4}	$\text{tree?}(g')$
{1}	$\exists (p: (\text{path?}(g'))): \text{root_path}(g', n')(p)$

Rewriting using min_def, matching in *,

Letting rp name choose(LAMBDA (p: (path?(g!1))): path_end(p) = n!1 AND length(p) = card(g!1'nodes) - max_len),

we get 2 subgoals:

tree_path_to_root.1.1:

{-1}	$\text{choose}(\lambda (p: (\text{path?}(g'))):$ $\quad \text{path_end}(p) = n' \wedge \text{length}(p) = \text{card}(g'\text{'nodes}) - \text{max_len})$
{-2}	$= \text{rp}$
{-2}	$\text{minimum?}(\text{max_len},$ $\quad \lambda (i: \text{nat}):$ $\quad \exists (q: (\text{path?}(g'))):$ $\quad \text{path_end}(q) = n' \wedge i = \text{card}(g'\text{'nodes}) - \text{length}(q)$
{-3}	$g'\text{'nodes}(n')$
{-4}	$\text{is_finite}(g'\text{'nodes})$
{-5}	$\text{tree?}(g')$
{1}	$\exists (p: (\text{path?}(g'))): \text{root_path}(g', n')(p)$

Instantiating the top quantifier in 1 with the terms: rp,

Expanding the definition of root_path,

Simplifying, rewriting, and recording with decision procedures,

Adding type constraints for rp,

Hiding formulas: -1, -3,

Using lemma non_root_path_extend,

Simplifying, rewriting, and recording with decision procedures,

Repeatedly Skolemizing and flattening,

Using lemma path_length_bound,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 2 subgoals:

C Proof scripts

`tree_path_to_root.1.1.1:`

<div style="display: flex; flex-direction: column; gap: 5px;"> <div style="display: flex; align-items: flex-start;"> {-1} $\forall (p: (\text{path?}(g'))): \text{length}(p) \leq \text{card}(g'\text{'nodes})$ </div> <div style="display: flex; align-items: flex-start;"> {-2} $\text{path_list?}[T](q')$ </div> <div style="display: flex; align-items: flex-start;"> {-3} $\text{path?}(g')(q')$ </div> <div style="display: flex; align-items: flex-start;"> {-4} $\text{length}(q') = 1 + \text{length}(\text{rp})$ </div> <div style="display: flex; align-items: flex-start;"> {-5} $\text{path_end}(q') = \text{path_end}(\text{rp})$ </div> <div style="display: flex; align-items: flex-start;"> {-6} $\text{path?}(g')(\text{rp})$ </div> <div style="display: flex; align-items: flex-start;"> {-7} $\text{length}(\text{rp}) = \text{card}(g'\text{'nodes}) - \text{max_len}$ </div> <div style="display: flex; align-items: flex-start;"> {-8} $\text{choose}(\lambda (p: (\text{path?}(g'))):$ $\quad \text{path_end}(p) = n' \wedge \text{length}(p) = \text{card}(g'\text{'nodes}) - \text{max_len})$ $\quad = \text{rp}$ </div> <div style="display: flex; align-items: flex-start;"> {-9} $\text{minimum?}(\text{max_len},$ $\quad \lambda (i: \text{nat}):$ $\quad \exists (q: (\text{path?}(g'))):$ $\quad \text{path_end}(q) = n' \wedge i = \text{card}(g'\text{'nodes}) - \text{length}(q))$ </div> <div style="display: flex; align-items: flex-start;"> {-10} $g'\text{'nodes}(n')$ </div> <div style="display: flex; align-items: flex-start;"> {-11} $\text{is_finite}(g'\text{'nodes})$ </div> <div style="display: flex; align-items: flex-start;"> {-12} $\text{tree?}(g')$ </div> </div>	<div style="display: flex; align-items: flex-start;"> <div style="margin-right: 10px;">{1}</div> $\text{roots}(g')(\text{path_start}(\text{rp}))$ </div>
---	--

Instantiating quantified variables,

Expanding the definition of `minimum?`,

Applying disjunctive simplification to flatten sequent,

Hiding formulas: -8, -9,

Instantiating the top quantifier in -8 with the terms: $\text{card}(g'\text{'nodes}) - \text{length}(q')$,

we get 2 subgoals:

`tree_path_to_root.1.1.1.1:`

<div style="display: flex; flex-direction: column; gap: 5px;"> <div style="display: flex; align-items: flex-start;"> {-1} $\text{length}(q') \leq \text{card}(g'\text{'nodes})$ </div> <div style="display: flex; align-items: flex-start;"> {-2} $\text{path_list?}[T](q')$ </div> <div style="display: flex; align-items: flex-start;"> {-3} $\text{path?}(g')(q')$ </div> <div style="display: flex; align-items: flex-start;"> {-4} $\text{length}(q') = 1 + \text{length}(\text{rp})$ </div> <div style="display: flex; align-items: flex-start;"> {-5} $\text{path_end}(q') = \text{path_end}(\text{rp})$ </div> <div style="display: flex; align-items: flex-start;"> {-6} $\text{path?}(g')(\text{rp})$ </div> <div style="display: flex; align-items: flex-start;"> {-7} $\text{length}(\text{rp}) = \text{card}(g'\text{'nodes}) - \text{max_len}$ </div> <div style="display: flex; align-items: flex-start;"> {-8} $(\exists (q: (\text{path?}(g'))):$ $\quad \text{path_end}(q) = n' \wedge$ $\quad \text{card}(g'\text{'nodes}) - \text{length}(q') = \text{card}(g'\text{'nodes}) - \text{length}(q))$ $\quad \supset \text{max_len} \leq \text{card}(g'\text{'nodes}) - \text{length}(q')$ </div> <div style="display: flex; align-items: flex-start;"> {-9} $g'\text{'nodes}(n')$ </div> <div style="display: flex; align-items: flex-start;"> {-10} $\text{is_finite}(g'\text{'nodes})$ </div> <div style="display: flex; align-items: flex-start;"> {-11} $\text{tree?}(g')$ </div> </div>	<div style="display: flex; align-items: flex-start;"> <div style="margin-right: 10px;">{1}</div> $\text{roots}(g')(\text{path_start}(\text{rp}))$ </div>
--	--

Simplifying, rewriting, and recording with decision procedures,

Instantiating quantified variables,

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `tree_path_to_root.1.1.1.1`.

tree_path_to_root.1.1.1.2:

{-1}	$\text{length}(q') \leq \text{card}(g' \text{'nodes})$
{-2}	$\text{path_list?}[T](q')$
{-3}	$\text{path?}(g')(q')$
{-4}	$\text{length}(q') = 1 + \text{length}(\text{rp})$
{-5}	$\text{path_end}(q') = \text{path_end}(\text{rp})$
{-6}	$\text{path?}(g')(\text{rp})$
{-7}	$\text{length}(\text{rp}) = \text{card}(g' \text{'nodes}) - \text{max_len}$
{-8}	$g' \text{'nodes}(n')$
{-9}	$\text{is_finite}(g' \text{'nodes})$
{-10}	$\text{tree?}(g')$
{1}	$\text{card}[T](g' \text{'nodes}) - \text{length}[T](q') \geq 0$
{2}	$\text{roots}(g')(\text{path_start}(\text{rp}))$

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of tree_path_to_root.1.1.1.2.

tree_path_to_root.1.1.2:

{-1}	$\text{path_list?}[T](q')$
{-2}	$\text{path?}(g')(q')$
{-3}	$\text{length}(q') = 1 + \text{length}(\text{rp})$
{-4}	$\text{path_end}(q') = \text{path_end}(\text{rp})$
{-5}	$\text{path?}(g')(\text{rp})$
{-6}	$\text{length}(\text{rp}) = \text{card}(g' \text{'nodes}) - \text{max_len}$
{-7}	$\text{choose}(\lambda (p: (\text{path?}(g'))):$ $\quad \text{path_end}(p) = n' \wedge \text{length}(p) = \text{card}(g' \text{'nodes}) - \text{max_len}$ $\quad = \text{rp}$
{-8}	$\text{minimum?}(\text{max_len},$ $\quad \lambda (i: \text{nat}):$ $\quad \exists (q: (\text{path?}(g'))):$ $\quad \text{path_end}(q) = n' \wedge i = \text{card}(g' \text{'nodes}) - \text{length}(q)$
{-9}	$g' \text{'nodes}(n')$
{-10}	$\text{is_finite}(g' \text{'nodes})$
{-11}	$\text{tree?}(g')$
{1}	$\text{cycle_free?}(g')$
{2}	$\text{roots}(g')(\text{path_start}(\text{rp}))$

Expanding the definition of tree?,

which is trivially true.

This completes the proof of tree_path_to_root.1.1.2.

tree_path_to_root.1.2:

{-1}	$\text{minimum?}(\text{max_len},$ $\quad \lambda (i: \text{nat}):$ $\quad \exists (q: (\text{path?}(g'))):$ $\quad \text{path_end}(q) = n' \wedge i = \text{card}(g' \text{'nodes}) - \text{length}(q)$
{-2}	$g' \text{'nodes}(n')$
{-3}	$\text{is_finite}(g' \text{'nodes})$
{-4}	$\text{tree?}(g')$
{1}	$\text{nonempty?}[(\text{path?}(g'))]$ $\quad (\lambda (p: (\text{path?}(g'))):$ $\quad \quad \text{path_end}[T](p) = n' \wedge$ $\quad \quad \text{length}[T](p) = \text{card}[T](g' \text{'nodes}) - \text{max_len}$
{2}	$\exists (p: (\text{path?}(g'))): \text{root_path}(g', n')(p)$

C Proof scripts

Expanding the definition of minimum?,
 Applying disjunctive simplification to flatten sequent,
 Hiding formulas: -2, 2,
 Expanding the definition(s) of (nonempty? empty? member),
 Repeatedly Skolemizing and flattening,
 Instantiating quantified variables,
 Simplifying, rewriting, and recording with decision procedures,
 This completes the proof of `tree_path_to_root.1.2`.

`tree_path_to_root.2:`

$\{-1\}$ $g' \text{ nodes}(n')$ $\{-2\}$ $\text{is_finite}(g' \text{ nodes})$ $\{-3\}$ $\text{tree?}(g')$	<hr style="border: 0.5px solid black;"/> $\{1\}$ $\text{nonempty?}[\text{nat}]$ $(\lambda (i: \text{nat}):$ $\quad \exists (q: (\text{path?}(g'))):$ $\quad \quad \text{path_end}[T](q) = n' \wedge i = \text{card}[T](g' \text{ nodes}) - \text{length}[T](q))$ $\{2\}$ $\exists (p: (\text{path?}(g'))): \text{root_path}(g', n')(p)$
--	---

Hiding formulas: 2,
 Installing automatic rewrites from: nonempty? empty? member path_end nth length path_list?
 path?

Simplifying, rewriting, and recording with decision procedures,
 Instantiating the top quantifier in -1 with the terms: $\text{card}(g' \text{ nodes}) - 1$,
 we get 2 subgoals:

`tree_path_to_root.2.1:`

$\{-1\}$ $g' \text{ nodes}(n')$ $\{-2\}$ $\text{is_finite}(g' \text{ nodes})$ $\{-3\}$ $\text{tree?}(g')$	<hr style="border: 0.5px solid black;"/> $\{1\}$ $\exists (q: (\text{path?}(g'))):$ $\quad \text{nth}(q, \text{length}(q) - 1) = n' \wedge$ $\quad \text{card}(g' \text{ nodes}) - 1 = \text{card}[T](g' \text{ nodes}) - \text{length}[T](q)$
--	--

Instantiating the top quantifier in 1 with the terms: $(:n':)$,
 we get 2 subgoals:

`tree_path_to_root.2.1.1:`

$\{-1\}$ $g' \text{ nodes}(n')$ $\{-2\}$ $\text{is_finite}(g' \text{ nodes})$ $\{-3\}$ $\text{tree?}(g')$	<hr style="border: 0.5px solid black;"/> $\{1\}$ $\text{nth}(:n':, \text{length}(:n':) - 1) = n' \wedge$ $\quad \text{card}(g' \text{ nodes}) - 1 = \text{card}[T](g' \text{ nodes}) - \text{length}[T]((:n':))$
--	---

Simplifying, rewriting, and recording with decision procedures,
 This completes the proof of `tree_path_to_root.2.1.1`.

`tree_path_to_root.2.1.2:`

$\{-1\}$ $g' \text{ nodes}(n')$ $\{-2\}$ $\text{is_finite}(g' \text{ nodes})$ $\{-3\}$ $\text{tree?}(g')$	<hr style="border: 0.5px solid black;"/> $\{1\}$ $(\forall (i: \text{below}(\text{length}(:n':))): g' \text{ nodes}(\text{nth}(:n':, i))) \wedge$ $\quad (\forall (i: \text{below}(\text{length}(:n':) - 1)):$ $\quad \quad g' \text{ edges}(\text{nth}(:n':, i), \text{nth}(::, i)))$
--	--

Repeatedly simplifying with decision procedures, rewriting, propositional reasoning, quantifier instantiation, skolemization, if-lifting and equality replacement,

This completes the proof of `tree_path_to_root.2.1.2`.

`tree_path_to_root.2.2`:

{-1}	$g' \text{ nodes}(n')$
{-2}	$\text{is_finite}(g' \text{ nodes})$
{-3}	$\text{tree?}(g')$
{1}	$\text{card}[T](g' \text{ nodes}) - 1 \geq 0$

Using lemma `card_singleton[T]`,

Using lemma `card_subset`,

Simplifying, rewriting, and recording with decision procedures,

Using lemma `singleton_subset[T]`,

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `tree_path_to_root.2.2`.

Q.E.D.

C.97.25 Graph.tree_unique_path_to_root

Terse proof for `tree_unique_path_to_root`.

`tree_unique_path_to_root`:

{1}	$\forall (g: \text{graph_type}, n: (g' \text{ nodes}), p_1, p_2: (\text{path?}(g))):$ $\text{tree?}(g) \wedge \text{root_path}(g, n)(p_1) \wedge \text{root_path}(g, n)(p_2) \supset p_1 = p_2$
-----	---

Repeatedly Skolemizing and flattening,

Installing automatic rewrites from: `path_list? length_tail car_tail cons_tail root_path`

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Case splitting on `length(p1!1) < length(p2!1)`,

we get 2 subgoals:

`tree_unique_path_to_root.1`:

{-1}	$\text{length}(p'_1) < \text{length}(p'_2)$
{-2}	$g' \text{ nodes}(n')$
{-3}	$\text{length}(p'_1) \geq 1$
{-4}	$\text{path?}(g')(p'_1)$
{-5}	$\text{length}(p'_2) \geq 1$
{-6}	$\text{path?}(g')(p'_2)$
{-7}	$\text{tree?}(g')$
{-8}	$\text{roots}(g')(\text{path_start}(p'_1))$
{-9}	$\text{path_end}(p'_1) = n'$
{-10}	$\text{roots}(g')(\text{path_start}(p'_2))$
{-11}	$\text{path_end}(p'_2) = n'$
{1}	$p'_1 = p'_2$

Using lemma `path_tail`,

we get 2 subgoals:

C Proof scripts

tree_unique_path_to_root.1.1:

{-1}	$\text{length}(p'_2) - \text{length}(p'_2) - \text{length}(p'_1) \geq 1 \supset$ $\text{path?}(g')(\text{tail}(p'_2, \text{length}(p'_2) - \text{length}(p'_1)))$
{-2}	$\text{length}(p'_1) < \text{length}(p'_2)$
{-3}	$g' \text{ 'nodes}(n')$
{-4}	$\text{length}(p'_1) \geq 1$
{-5}	$\text{path?}(g')(p'_1)$
{-6}	$\text{length}(p'_2) \geq 1$
{-7}	$\text{path?}(g')(p'_2)$
{-8}	$\text{tree?}(g')$
{-9}	$\text{roots}(g')(\text{path_start}(p'_1))$
{-10}	$\text{path_end}(p'_1) = n'$
{-11}	$\text{roots}(g')(\text{path_start}(p'_2))$
{-12}	$\text{path_end}(p'_2) = n'$
{1}	$p'_1 = p'_2$

Simplifying, rewriting, and recording with decision procedures,

Using lemma tree_path,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 2 subgoals:

tree_unique_path_to_root.1.1.1:

{-1}	$\text{tree?}(g')$
{-2}	$p'_1 = \text{tail}(p'_2, \text{length}(p'_2) - \text{length}(p'_1))$
{-3}	$\text{path?}(g')(\text{tail}(p'_2, \text{length}(p'_2) - \text{length}(p'_1)))$
{-4}	$\text{length}(p'_1) < \text{length}(p'_2)$
{-5}	$g' \text{ 'nodes}(n')$
{-6}	$\text{length}(p'_1) \geq 1$
{-7}	$\text{path?}(g')(p'_1)$
{-8}	$\text{length}(p'_2) \geq 1$
{-9}	$\text{path?}(g')(p'_2)$
{-10}	$\text{roots}(g')(\text{path_start}(p'_1))$
{-11}	$\text{path_end}(p'_1) = n'$
{-12}	$\text{roots}(g')(\text{path_start}(p'_2))$
{-13}	$\text{path_end}(p'_2) = n'$
{1}	$p'_1 = p'_2$

Expanding the definition of roots,

Expanding the definition of path?,

Applying disjunctive simplification to flatten sequent,

Instantiating the top quantifier in -11 with the terms: $\text{nth}(p'_2, \text{length}(p'_2) - \text{length}(p'_1) - 1)$,

we get 2 subgoals:

tree_unique_path_to_root.1.1.1.1.1:

{-1}	tree?(g')
{-2}	$p'_1 = \text{tail}(p'_2, \text{length}(p'_2) - \text{length}(p'_1))$
{-3}	path?(g')(tail(p'_2, length(p'_2) - length(p'_1)))
{-4}	$\text{length}(p'_1) < \text{length}(p'_2)$
{-5}	$g' \text{' nodes}(n')$
{-6}	$\text{length}(p'_1) \geq 1$
{-7}	path?(g')(p'_1)
{-8}	$\text{length}(p'_2) \geq 1$
{-9}	$\forall (i: \text{below}(\text{length}(p'_2))): g' \text{' nodes}(\text{nth}(p'_2, i))$
{-10}	$\forall (i: \text{below}(\text{length}(p'_2) - 1)):$ $g' \text{' edges}(\text{nth}(p'_2, i), \text{nth}(p'_2, 1 + i))$
{-11}	path_end(p'_1) = n'
{-12}	roots(g')(path_start(p'_2))
{-13}	path_end(p'_2) = n'
{1}	$g' \text{' edges}(\text{nth}(p'_2, \text{length}(p'_2) - \text{length}(p'_1) - 1), \text{path_start}(p'_1))$
{2}	$p'_1 = p'_2$

Instantiating the top quantifier in -10 with the terms: $\text{length}(p'_2) - \text{length}(p'_1) - 1$,

Expanding the definition of path_start,

Case splitting on $\text{car}(p1!1) = \text{car}(\text{tail}(p2!1, \text{length}(p2!1) - \text{length}(p1!1)))$,

we get 2 subgoals:

tree_unique_path_to_root.1.1.1.1.1.1:

{-1}	$\text{car}(p'_1) = \text{car}(\text{tail}(p'_2, \text{length}(p'_2) - \text{length}(p'_1)))$
{-2}	tree?(g')
{-3}	$p'_1 = \text{tail}(p'_2, \text{length}(p'_2) - \text{length}(p'_1))$
{-4}	path?(g')(tail(p'_2, length(p'_2) - length(p'_1)))
{-5}	$\text{length}(p'_1) < \text{length}(p'_2)$
{-6}	$g' \text{' nodes}(n')$
{-7}	$\text{length}(p'_1) \geq 1$
{-8}	path?(g')(p'_1)
{-9}	$\text{length}(p'_2) \geq 1$
{-10}	$\forall (i: \text{below}(\text{length}(p'_2))): g' \text{' nodes}(\text{nth}(p'_2, i))$
{-11}	$g' \text{' edges}$ $(\text{nth}(p'_2, \text{length}(p'_2) - \text{length}(p'_1) - 1),$ $\text{nth}(p'_2, 1 + \text{length}(p'_2) - \text{length}(p'_1) - 1))$
{-12}	path_end(p'_1) = n'
{-13}	roots(g')(path_start(p'_2))
{-14}	path_end(p'_2) = n'
{1}	$g' \text{' edges}(\text{nth}(p'_2, \text{length}(p'_2) - 1 - \text{length}(p'_1)), \text{car}(p'_1))$
{2}	$p'_1 = p'_2$

Replacing using formula -1,

Hiding formulas: -1,

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of tree_unique_path_to_root.1.1.1.1.1.1.

C Proof scripts

`tree_unique_path_to_root.1.1.1.1.2:`

{-1}	$\text{tree?}(g')$
{-2}	$p'_1 = \text{tail}(p'_2, \text{length}(p'_2) - \text{length}(p'_1))$
{-3}	$\text{path?}(g')(\text{tail}(p'_2, \text{length}(p'_2) - \text{length}(p'_1)))$
{-4}	$\text{length}(p'_1) < \text{length}(p'_2)$
{-5}	$g' \text{'nodes}(n')$
{-6}	$\text{length}(p'_1) \geq 1$
{-7}	$\text{path?}(g')(p'_1)$
{-8}	$\text{length}(p'_2) \geq 1$
{-9}	$\forall (i: \text{below}(\text{length}(p'_2))): g' \text{'nodes}(\text{nth}(p'_2, i))$
{-10}	$g' \text{'edges}$ $(\text{nth}(p'_2, \text{length}(p'_2) - \text{length}(p'_1) - 1),$ $\text{nth}(p'_2, 1 + \text{length}(p'_2) - \text{length}(p'_1) - 1))$
{-11}	$\text{path_end}(p'_1) = n'$
{-12}	$\text{roots}(g')(\text{path_start}(p'_2))$
{-13}	$\text{path_end}(p'_2) = n'$
{1}	$\text{car}(p'_1) = \text{car}(\text{tail}(p'_2, \text{length}(p'_2) - \text{length}(p'_1)))$
{2}	$g' \text{'edges}(\text{nth}(p'_2, \text{length}(p'_2) - 1 - \text{length}(p'_1)), \text{car}(p'_1))$
{3}	$p'_1 = p'_2$

Replacing using formula -2,

which is trivially true.

This completes the proof of `tree_unique_path_to_root.1.1.1.1.2`.

`tree_unique_path_to_root.1.1.1.2:`

{-1}	$\text{tree?}(g')$
{-2}	$p'_1 = \text{tail}(p'_2, \text{length}(p'_2) - \text{length}(p'_1))$
{-3}	$\text{path?}(g')(\text{tail}(p'_2, \text{length}(p'_2) - \text{length}(p'_1)))$
{-4}	$\text{length}(p'_1) < \text{length}(p'_2)$
{-5}	$g' \text{'nodes}(n')$
{-6}	$\text{length}(p'_1) \geq 1$
{-7}	$\text{path?}(g')(p'_1)$
{-8}	$\text{length}(p'_2) \geq 1$
{-9}	$\forall (i: \text{below}(\text{length}(p'_2))): g' \text{'nodes}(\text{nth}(p'_2, i))$
{-10}	$\forall (i: \text{below}(\text{length}(p'_2) - 1)):$ $g' \text{'edges}(\text{nth}(p'_2, i), \text{nth}(p'_2, 1 + i))$
{-11}	$\text{path_end}(p'_1) = n'$
{-12}	$\text{roots}(g')(\text{path_start}(p'_2))$
{-13}	$\text{path_end}(p'_2) = n'$
{1}	$g' \text{'nodes}(\text{nth}[T](p'_2, \text{length}[T](p'_2) - 1 - \text{length}[T](p'_1)))$
{2}	$p'_1 = p'_2$

Instantiating quantified variables,

This completes the proof of `tree_unique_path_to_root.1.1.1.2`.

tree_unique_path_to_root.1.1.2:

{-1}	tree?(g')
{-2}	path?(g')(tail(p'_2 , length(p'_2) - length(p'_1)))
{-3}	length(p'_1) < length(p'_2)
{-4}	g' 'nodes(n')
{-5}	length(p'_1) \geq 1
{-6}	path?(g')(p'_1)
{-7}	length(p'_2) \geq 1
{-8}	path?(g')(p'_2)
{-9}	roots(g')(path_start(p'_1))
{-10}	path_end(p'_1) = n'
{-11}	roots(g')(path_start(p'_2))
{-12}	path_end(p'_2) = n'
{1}	path_end(p'_1) = path_end(tail(p'_2 , length(p'_2) - length(p'_1)))
{2}	$p'_1 = p'_2$

Installing automatic rewrites from: path_end_nth_tail

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of tree_unique_path_to_root.1.1.2.

tree_unique_path_to_root.1.2:

{-1}	length(p'_1) < length(p'_2)
{-2}	g' 'nodes(n')
{-3}	length(p'_1) \geq 1
{-4}	path?(g')(p'_1)
{-5}	length(p'_2) \geq 1
{-6}	path?(g')(p'_2)
{-7}	tree?(g')
{-8}	roots(g')(path_start(p'_1))
{-9}	path_end(p'_1) = n'
{-10}	roots(g')(path_start(p'_2))
{-11}	path_end(p'_2) = n'
{1}	length[T](p'_2) - length[T](p'_1) \geq 0
{2}	$p'_1 = p'_2$

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of tree_unique_path_to_root.1.2.

tree_unique_path_to_root.2:

{-1}	g' 'nodes(n')
{-2}	length(p'_1) \geq 1
{-3}	path?(g')(p'_1)
{-4}	length(p'_2) \geq 1
{-5}	path?(g')(p'_2)
{-6}	tree?(g')
{-7}	roots(g')(path_start(p'_1))
{-8}	path_end(p'_1) = n'
{-9}	roots(g')(path_start(p'_2))
{-10}	path_end(p'_2) = n'
{1}	length(p'_1) < length(p'_2)
{2}	$p'_1 = p'_2$

Case splitting on length(p_1) > length(p_2),

we get 2 subgoals:

C Proof scripts

tree_unique_path_to_root.2.1:

{-1}	$\text{length}(p'_1) > \text{length}(p'_2)$
{-2}	$g' \text{ nodes}(n')$
{-3}	$\text{length}(p'_1) \geq 1$
{-4}	$\text{path?}(g')(p'_1)$
{-5}	$\text{length}(p'_2) \geq 1$
{-6}	$\text{path?}(g')(p'_2)$
{-7}	$\text{tree?}(g')$
{-8}	$\text{roots}(g')(\text{path_start}(p'_1))$
{-9}	$\text{path_end}(p'_1) = n'$
{-10}	$\text{roots}(g')(\text{path_start}(p'_2))$
{-11}	$\text{path_end}(p'_2) = n'$
{1}	$\text{length}(p'_1) < \text{length}(p'_2)$
{2}	$p'_1 = p'_2$

Hiding formulas: 1,

Using lemma path_tail,

we get 2 subgoals:

tree_unique_path_to_root.2.1.1:

{-1}	$\text{length}(p'_1) - \text{length}(p'_1) - \text{length}(p'_2) \geq 1 \supset$ $\text{path?}(g')(\text{tail}(p'_1, \text{length}(p'_1) - \text{length}(p'_2)))$
{-2}	$\text{length}(p'_1) > \text{length}(p'_2)$
{-3}	$g' \text{ nodes}(n')$
{-4}	$\text{length}(p'_1) \geq 1$
{-5}	$\text{path?}(g')(p'_1)$
{-6}	$\text{length}(p'_2) \geq 1$
{-7}	$\text{path?}(g')(p'_2)$
{-8}	$\text{tree?}(g')$
{-9}	$\text{roots}(g')(\text{path_start}(p'_1))$
{-10}	$\text{path_end}(p'_1) = n'$
{-11}	$\text{roots}(g')(\text{path_start}(p'_2))$
{-12}	$\text{path_end}(p'_2) = n'$
{1}	$p'_1 = p'_2$

Simplifying, rewriting, and recording with decision procedures,

Using lemma tree_path,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 2 subgoals:

tree_unique_path_to_root.2.1.1.1.1:

{-1}	$tree?(g')$
{-2}	$p'_2 = tail(p'_1, length(p'_1) - length(p'_2))$
{-3}	$path?(g')(tail(p'_1, length(p'_1) - length(p'_2)))$
{-4}	$length(p'_1) > length(p'_2)$
{-5}	$g' \text{ nodes}(n')$
{-6}	$length(p'_1) \geq 1$
{-7}	$path?(g')(p'_1)$
{-8}	$length(p'_2) \geq 1$
{-9}	$path?(g')(p'_2)$
{-10}	$roots(g')(path_start(p'_1))$
{-11}	$path_end(p'_1) = n'$
{-12}	$roots(g')(path_start(p'_2))$
{-13}	$path_end(p'_2) = n'$
{1}	$p'_1 = p'_2$

Expanding the definition of roots,

Expanding the definition of path?,

Applying disjunctive simplification to flatten sequent,

Instantiating the top quantifier in -13 with the terms: $nth(p'_1, length(p'_1) - length(p'_2) - 1)$,

we get 2 subgoals:

tree_unique_path_to_root.2.1.1.1.1.1:

{-1}	$tree?(g')$
{-2}	$p'_2 = tail(p'_1, length(p'_1) - length(p'_2))$
{-3}	$path?(g')(tail(p'_1, length(p'_1) - length(p'_2)))$
{-4}	$length(p'_1) > length(p'_2)$
{-5}	$g' \text{ nodes}(n')$
{-6}	$length(p'_1) \geq 1$
{-7}	$\forall (i: \text{below}(length(p'_1))): g' \text{ nodes}(nth(p'_1, i))$
{-8}	$\forall (i: \text{below}(length(p'_1) - 1)):$ $g' \text{ edges}(nth(p'_1, i), nth(p'_1, 1 + i))$
{-9}	$length(p'_2) \geq 1$
{-10}	$path?(g')(p'_2)$
{-11}	$roots(g')(path_start(p'_1))$
{-12}	$path_end(p'_1) = n'$
{-13}	$path_end(p'_2) = n'$
{1}	$g' \text{ edges}(nth(p'_1, length(p'_1) - length(p'_2) - 1), path_start(p'_2))$
{2}	$p'_1 = p'_2$

Instantiating the top quantifier in -8 with the terms: $length(p'_1) - length(p'_2) - 1$,

Expanding the definition of path_start,

Case splitting on $car(p2!1) = car(tail(p1!1, length(p1!1) - length(p2!1)))$,

we get 2 subgoals:

tree_unique_path_to_root.2.1.1.1.1.1:

{-1}	$\text{car}(p'_2) = \text{car}(\text{tail}(p'_1, \text{length}(p'_1) - \text{length}(p'_2)))$
{-2}	$\text{tree?}(g')$
{-3}	$p'_2 = \text{tail}(p'_1, \text{length}(p'_1) - \text{length}(p'_2))$
{-4}	$\text{path?}(g')(\text{tail}(p'_1, \text{length}(p'_1) - \text{length}(p'_2)))$
{-5}	$\text{length}(p'_1) > \text{length}(p'_2)$
{-6}	$g' \text{' nodes}(n')$
{-7}	$\text{length}(p'_1) \geq 1$
{-8}	$\forall (i: \text{below}(\text{length}(p'_1))): g' \text{' nodes}(\text{nth}(p'_1, i))$
{-9}	$g' \text{' edges}$ $(\text{nth}(p'_1, \text{length}(p'_1) - \text{length}(p'_2) - 1),$ $\text{nth}(p'_1, 1 + \text{length}(p'_1) - \text{length}(p'_2) - 1))$
{-10}	$\text{length}(p'_2) \geq 1$
{-11}	$\text{path?}(g')(p'_2)$
{-12}	$\text{roots}(g')(\text{path_start}(p'_1))$
{-13}	$\text{path_end}(p'_1) = n'$
{-14}	$\text{path_end}(p'_2) = n'$
{1}	$g' \text{' edges}(\text{nth}(p'_1, \text{length}(p'_1) - 1 - \text{length}(p'_2)), \text{car}(p'_2))$
{2}	$p'_1 = p'_2$

Replacing using formula -1,

Hiding formulas: -1,

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of tree_unique_path_to_root.2.1.1.1.1.1.

tree_unique_path_to_root.2.1.1.1.1.2:

{-1}	$\text{tree?}(g')$
{-2}	$p'_2 = \text{tail}(p'_1, \text{length}(p'_1) - \text{length}(p'_2))$
{-3}	$\text{path?}(g')(\text{tail}(p'_1, \text{length}(p'_1) - \text{length}(p'_2)))$
{-4}	$\text{length}(p'_1) > \text{length}(p'_2)$
{-5}	$g' \text{' nodes}(n')$
{-6}	$\text{length}(p'_1) \geq 1$
{-7}	$\forall (i: \text{below}(\text{length}(p'_1))): g' \text{' nodes}(\text{nth}(p'_1, i))$
{-8}	$g' \text{' edges}$ $(\text{nth}(p'_1, \text{length}(p'_1) - \text{length}(p'_2) - 1),$ $\text{nth}(p'_1, 1 + \text{length}(p'_1) - \text{length}(p'_2) - 1))$
{-9}	$\text{length}(p'_2) \geq 1$
{-10}	$\text{path?}(g')(p'_2)$
{-11}	$\text{roots}(g')(\text{path_start}(p'_1))$
{-12}	$\text{path_end}(p'_1) = n'$
{-13}	$\text{path_end}(p'_2) = n'$
{1}	$\text{car}(p'_2) = \text{car}(\text{tail}(p'_1, \text{length}(p'_1) - \text{length}(p'_2)))$
{2}	$g' \text{' edges}(\text{nth}(p'_1, \text{length}(p'_1) - 1 - \text{length}(p'_2)), \text{car}(p'_2))$
{3}	$p'_1 = p'_2$

Replacing using formula -2,

which is trivially true.

This completes the proof of tree_unique_path_to_root.2.1.1.1.1.2.

tree_unique_path_to_root.2.1.1.1.2:

{-1}	tree?(g')
{-2}	$p'_2 = \text{tail}(p'_1, \text{length}(p'_1) - \text{length}(p'_2))$
{-3}	$\text{path?}(g')(\text{tail}(p'_1, \text{length}(p'_1) - \text{length}(p'_2)))$
{-4}	$\text{length}(p'_1) > \text{length}(p'_2)$
{-5}	$g' \text{'nodes}(n')$
{-6}	$\text{length}(p'_1) \geq 1$
{-7}	$\forall (i: \text{below}(\text{length}(p'_1))): g' \text{'nodes}(\text{nth}(p'_1, i))$
{-8}	$\forall (i: \text{below}(\text{length}(p'_1) - 1)):$ $g' \text{'edges}(\text{nth}(p'_1, i), \text{nth}(p'_1, 1 + i))$
{-9}	$\text{length}(p'_2) \geq 1$
{-10}	$\text{path?}(g')(p'_2)$
{-11}	$\text{roots}(g')(\text{path_start}(p'_1))$
{-12}	$\text{path_end}(p'_1) = n'$
{-13}	$\text{path_end}(p'_2) = n'$
{1}	$g' \text{'nodes}(\text{nth}[T](p'_1, \text{length}[T](p'_1) - 1 - \text{length}[T](p'_2)))$
{2}	$p'_1 = p'_2$

Instantiating quantified variables,

This completes the proof of tree_unique_path_to_root.2.1.1.1.2.

tree_unique_path_to_root.2.1.1.2:

{-1}	tree?(g')
{-2}	$\text{path?}(g')(\text{tail}(p'_1, \text{length}(p'_1) - \text{length}(p'_2)))$
{-3}	$\text{length}(p'_1) > \text{length}(p'_2)$
{-4}	$g' \text{'nodes}(n')$
{-5}	$\text{length}(p'_1) \geq 1$
{-6}	$\text{path?}(g')(p'_1)$
{-7}	$\text{length}(p'_2) \geq 1$
{-8}	$\text{path?}(g')(p'_2)$
{-9}	$\text{roots}(g')(\text{path_start}(p'_1))$
{-10}	$\text{path_end}(p'_1) = n'$
{-11}	$\text{roots}(g')(\text{path_start}(p'_2))$
{-12}	$\text{path_end}(p'_2) = n'$
{1}	$\text{path_end}(p'_2) = \text{path_end}(\text{tail}(p'_1, \text{length}(p'_1) - \text{length}(p'_2)))$
{2}	$p'_1 = p'_2$

Installing automatic rewrites from: path_end nth_tail

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of tree_unique_path_to_root.2.1.1.2.

tree_unique_path_to_root.2.1.2:

{-1}	$\text{length}(p'_1) > \text{length}(p'_2)$
{-2}	$g' \text{ nodes}(n')$
{-3}	$\text{length}(p'_1) \geq 1$
{-4}	$\text{path?}(g')(p'_1)$
{-5}	$\text{length}(p'_2) \geq 1$
{-6}	$\text{path?}(g')(p'_2)$
{-7}	$\text{tree?}(g')$
{-8}	$\text{roots}(g')(\text{path_start}(p'_1))$
{-9}	$\text{path_end}(p'_1) = n'$
{-10}	$\text{roots}(g')(\text{path_start}(p'_2))$
{-11}	$\text{path_end}(p'_2) = n'$
{1}	$\text{length}[T](p'_1) - \text{length}[T](p'_2) \geq 0$
{2}	$p'_1 = p'_2$

Simplifying, rewriting, and recording with decision procedures,
This completes the proof of tree_unique_path_to_root.2.1.2.

tree_unique_path_to_root.2.2:

{-1}	$g' \text{ nodes}(n')$
{-2}	$\text{length}(p'_1) \geq 1$
{-3}	$\text{path?}(g')(p'_1)$
{-4}	$\text{length}(p'_2) \geq 1$
{-5}	$\text{path?}(g')(p'_2)$
{-6}	$\text{tree?}(g')$
{-7}	$\text{roots}(g')(\text{path_start}(p'_1))$
{-8}	$\text{path_end}(p'_1) = n'$
{-9}	$\text{roots}(g')(\text{path_start}(p'_2))$
{-10}	$\text{path_end}(p'_2) = n'$
{1}	$\text{length}(p'_1) > \text{length}(p'_2)$
{2}	$\text{length}(p'_1) < \text{length}(p'_2)$
{3}	$p'_1 = p'_2$

Rewriting using tree_path, matching in *,
This completes the proof of tree_unique_path_to_root.2.2.
Q.E.D.

C.97.26 Graph.root_path_singleton

Terse proof for root_path_singleton.

root_path_singleton:

	{1} $\forall (g: \text{graph_type}, n: (g' \text{ nodes})): \text{finite_tree?}(g) \supset \text{singleton?}(\text{root_path}(g, n))$
--	--

Repeatedly Skolemizing and flattening,
Expanding the definition of singleton?,
Using lemma tree_path_to_root,
Simplifying, rewriting, and recording with decision procedures,
Instantiating quantified variables,
Simplifying, rewriting, and recording with decision procedures,
Repeatedly Skolemizing and flattening,
Instantiating quantified variables,
Repeatedly Skolemizing and flattening,

Using lemma `tree_unique_path_to_root`,
 Simplifying, rewriting, and recording with decision procedures,
 Expanding the definition of `finite_tree?`,
 which is trivially true.
 This completes the proof of `root_path_singleton`.
 Q.E.D.

C.97.27 Graph.path_to_root_TCC1

Terse proof for `path_to_root_TCC1`.

`path_to_root_TCC1`:

$$\{1\} \quad \forall (g: (\text{finite_tree?}), n: (g' \text{nodes})): \\ \text{singleton?}[(\text{path?}(g))](\text{root_path}(g, n))$$

Repeatedly Skolemizing and flattening,
 Rewriting using `root_path_singleton`, matching in *,
 This completes the proof of `path_to_root_TCC1`.
 Q.E.D.

C.97.28 Graph.path_factor_TCC1

Terse proof for `path_factor_TCC1`.

`path_factor_TCC1`:

$$\{1\} \quad \forall (g: \text{graph_type}, p_1, p_2: (\text{path?}(g))): \\ (\text{path_start}(p_1) = \text{path_start}(p_2) \wedge \\ (\exists (n: \text{nat}): \\ n < \text{length}(p_1) \wedge n < \text{length}(p_2) \wedge \neg \text{nth}(p_1, n) = \text{nth}(p_2, n))) \\ \supset \\ (\forall (q, p1t, p2t: (\text{path?}(g))): \\ p_2 = \text{concat_path}(q, p2t) \wedge \\ p_1 = \text{concat_path}(q, p1t) \wedge \\ \text{length}(p2t) \geq 2 \wedge \\ \text{length}(p1t) \geq 2 \wedge \text{concatable?}(q, p2t) \wedge \text{concatable?}(q, p1t) \\ \supset 0 < \text{length}[T](p1t))$$

Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of `path_factor_TCC1`.
 Q.E.D.

C.97.29 Graph.path_factor_TCC2

Terse proof for `path_factor_TCC2`.

path_factor_TCC2:

$$\begin{array}{l}
 \{1\} \quad \forall (g: \text{graph_type}, p_1, p_2: (\text{path?}(g))): \\
 \quad (\text{path_start}(p_1) = \text{path_start}(p_2) \wedge \\
 \quad \quad (\exists (n: \text{nat}): \\
 \quad \quad \quad n < \text{length}(p_1) \wedge n < \text{length}(p_2) \wedge \neg \text{nth}(p_1, n) = \text{nth}(p_2, n))) \\
 \quad \supset \\
 \quad (\forall (q, p1t, p2t: (\text{path?}(g))): \\
 \quad \quad p_2 = \text{concat_path}(q, p2t) \wedge \\
 \quad \quad p_1 = \text{concat_path}(q, p1t) \wedge \\
 \quad \quad \text{length}(p2t) \geq 2 \wedge \\
 \quad \quad \text{length}(p1t) \geq 2 \wedge \text{concatable?}(q, p2t) \wedge \text{concatable?}(q, p1t) \\
 \quad \quad \supset 0 < \text{length}^{[T]}(p2t))
 \end{array}$$

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of path_factor_TCC2.

Q.E.D.

C.97.30 Graph.path_factor_TCC3

Terse proof for path_factor_TCC3.

path_factor_TCC3:

$$\begin{array}{l}
 \{1\} \quad \forall (g: \text{graph_type}, p_1, p_2: (\text{path?}(g))): \\
 \quad (\text{path_start}(p_1) = \text{path_start}(p_2) \wedge \\
 \quad \quad (\exists (n: \text{nat}): \\
 \quad \quad \quad n < \text{length}(p_1) \wedge n < \text{length}(p_2) \wedge \neg \text{nth}(p_1, n) = \text{nth}(p_2, n))) \\
 \quad \supset \\
 \quad (\forall (q, p1t, p2t: (\text{path?}(g))): \\
 \quad \quad \text{nth}(p1t, 0) = \text{nth}(p2t, 0) \wedge \\
 \quad \quad p_2 = \text{concat_path}(q, p2t) \wedge \\
 \quad \quad p_1 = \text{concat_path}(q, p1t) \wedge \\
 \quad \quad \text{length}(p2t) \geq 2 \wedge \\
 \quad \quad \text{length}(p1t) \geq 2 \wedge \text{concatable?}(q, p2t) \wedge \text{concatable?}(q, p1t) \\
 \quad \quad \supset 1 < \text{length}^{[T]}(p1t))
 \end{array}$$

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of path_factor_TCC3.

Q.E.D.

C.97.31 Graph.path_factor_TCC4

Terse proof for path_factor_TCC4.

path_factor_TCC4:

```

{1}  ∀ (g: graph_type, p1, p2: (path?(g))):
      (path_start(p1) = path_start(p2) ∧
       (∃ (n: nat):
          n < length(p1) ∧ n < length(p2) ∧ ¬ nth(p1, n) = nth(p2, n)))
      ⊃
      (∀ (q, p1t, p2t: (path?(g))):
         nth(p1t, 0) = nth(p2t, 0) ∧
         p2 = concat_path(q, p2t) ∧
         p1 = concat_path(q, p1t) ∧
         length(p2t) ≥ 2 ∧
         length(p1t) ≥ 2 ∧ concatable?(q, p2t) ∧ concatable?(q, p1t)
         ⊃ 1 < length[T](p2t))

```

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of path_factor_TCC4.

Q.E.D.

C.97.32 Graph.path_factor

Terse proof for path_factor.

path_factor:

```

{1}  ∀ (g: graph_type, p1, p2: (path?(g))):
      (path_start(p1) = path_start(p2) ∧
       (∃ (n: nat):
          n < length(p1) ∧ n < length(p2) ∧ ¬ nth(p1, n) = nth(p2, n)))
      ⊃
      (∃ (q, p1t, p2t: (path?(g))):
         concatable?(q, p1t) ∧
         concatable?(q, p2t) ∧
         length(p1t) ≥ 2 ∧
         length(p2t) ≥ 2 ∧
         p1 = concat_path(q, p1t) ∧
         p2 = concat_path(q, p2t) ∧
         nth(p1t, 0) = nth(p2t, 0) ∧ ¬ nth(p1t, 1) = nth(p2t, 1))

```

Repeatedly Skolemizing and flattening,

Installing automatic rewrites from: concatable? concat_path head_tail path_list? length_head length_tail path_end path_start nth_tail car_tail cdr_tail path_tail path_head nth_head

Simplifying, rewriting, and recording with decision procedures,

Letting nd name min(LAMBDA (i: nat): i < length(p1!1) AND i < length(p2!1) AND NOT nth(p1!1, i) = nth(p2!1, i)),

we get 2 subgoals:

C Proof scripts

path_factor.1:

{-1}	$\min(\lambda (i: \text{nat}):$
	$i < \text{length}(p'_1) \wedge i < \text{length}(p'_2) \wedge \neg \text{nth}(p'_1, i) = \text{nth}(p'_2, i))$
	$= \text{nd}$
{-2}	$n' \geq 0$
{-3}	$\text{length}(p'_1) \geq 1$
{-4}	$\text{path?}(g')(p'_1)$
{-5}	$\text{length}(p'_2) \geq 1$
{-6}	$\text{path?}(g')(p'_2)$
{-7}	$\text{car}(p'_1) = \text{car}(p'_2)$
{-8}	$n' < \text{length}(p'_1)$
{-9}	$n' < \text{length}(p'_2)$
{1}	$\text{nth}(p'_1, n') = \text{nth}(p'_2, n')$
{2}	$\exists (q, p1t, p2t: (\text{path?}(g'))):$
	$\text{nth}(q, \text{length}(q) - 1) = \text{car}(p1t) \wedge$
	$\text{nth}(q, \text{length}(q) - 1) = \text{car}(p2t) \wedge$
	$\text{length}(p1t) \geq 2 \wedge$
	$\text{length}(p2t) \geq 2 \wedge$
	$p'_1 = \text{append}(q, \text{cdr}(p1t)) \wedge$
	$p'_2 = \text{append}(q, \text{cdr}(p2t)) \wedge$
	$\text{nth}(p1t, 0) = \text{nth}(p2t, 0) \wedge \neg \text{nth}(p1t, 1) = \text{nth}(p2t, 1)$

Rewriting using min_def, matching in *,

Case splitting on $\text{nd} = 0$,

we get 2 subgoals:

path_factor.1.1:

{-1}	$\text{nd} = 0$
{-2}	$\text{minimum?}(\text{nd},$
	$\lambda (i: \text{nat}):$
	$i < \text{length}(p'_1) \wedge i < \text{length}(p'_2) \wedge \neg \text{nth}(p'_1, i) = \text{nth}(p'_2, i))$
{-3}	$n' \geq 0$
{-4}	$\text{length}(p'_1) \geq 1$
{-5}	$\text{path?}(g')(p'_1)$
{-6}	$\text{length}(p'_2) \geq 1$
{-7}	$\text{path?}(g')(p'_2)$
{-8}	$\text{car}(p'_1) = \text{car}(p'_2)$
{-9}	$n' < \text{length}(p'_1)$
{-10}	$n' < \text{length}(p'_2)$
{1}	$\text{nth}(p'_1, n') = \text{nth}(p'_2, n')$
{2}	$\exists (q, p1t, p2t: (\text{path?}(g'))):$
	$\text{nth}(q, \text{length}(q) - 1) = \text{car}(p1t) \wedge$
	$\text{nth}(q, \text{length}(q) - 1) = \text{car}(p2t) \wedge$
	$\text{length}(p1t) \geq 2 \wedge$
	$\text{length}(p2t) \geq 2 \wedge$
	$p'_1 = \text{append}(q, \text{cdr}(p1t)) \wedge$
	$p'_2 = \text{append}(q, \text{cdr}(p2t)) \wedge$
	$\text{nth}(p1t, 0) = \text{nth}(p2t, 0) \wedge \neg \text{nth}(p1t, 1) = \text{nth}(p2t, 1)$

Hiding formulas: -2, 2,

Adding type constraints for nd,

Installing automatic rewrites from: nth

Simplifying, rewriting, and recording with decision procedures,

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `path_factor.1.1`.

`path_factor.1.2`:

<pre>{-1} minimum?(nd, λ (i: nat): i < length(p'1) ∧ i < length(p'2) ∧ ¬ nth(p'1, i) = nth(p'2, i)) {-2} n' ≥ 0 {-3} length(p'1) ≥ 1 {-4} path?(g')(p'1) {-5} length(p'2) ≥ 1 {-6} path?(g')(p'2) {-7} car(p'1) = car(p'2) {-8} n' < length(p'1) {-9} n' < length(p'2)</pre>	<hr style="border: 0.5px solid black;"/> <pre>{1} nd = 0 {2} nth(p'1, n') = nth(p'2, n') {3} ∃ (q, p1t, p2t: (path?(g'))): nth(q, length(q) - 1) = car(p1t) ∧ nth(q, length(q) - 1) = car(p2t) ∧ length(p1t) ≥ 2 ∧ length(p2t) ≥ 2 ∧ p'1 = append(q, cdr(p1t)) ∧ p'2 = append(q, cdr(p2t)) ∧ nth(p1t, 0) = nth(p2t, 0) ∧ ¬ nth(p1t, 1) = nth(p2t, 1)</pre>
---	---

Instantiating the top quantifier in 3 with the terms: `head(p'1, nd)`, `tail(p'1, nd - 1)`, `tail(p'2, nd - 1)`, we get 6 subgoals:

`path_factor.1.2.1`:

<pre>{-1} minimum?(nd, λ (i: nat): i < length(p'1) ∧ i < length(p'2) ∧ ¬ nth(p'1, i) = nth(p'2, i)) {-2} n' ≥ 0 {-3} length(p'1) ≥ 1 {-4} path?(g')(p'1) {-5} length(p'2) ≥ 1 {-6} path?(g')(p'2) {-7} car(p'1) = car(p'2) {-8} n' < length(p'1) {-9} n' < length(p'2)</pre>	<hr style="border: 0.5px solid black;"/> <pre>{1} nd = 0 {2} nth(p'1, n') = nth(p'2, n') {3} nth(head(p'1, nd), length(head(p'1, nd)) - 1) = car(tail(p'1, nd - 1)) ∧ nth(head(p'1, nd), length(head(p'1, nd)) - 1) = car(tail(p'2, nd - 1)) ∧ length(tail(p'1, nd - 1)) ≥ 2 ∧ length(tail(p'2, nd - 1)) ≥ 2 ∧ p'1 = append(head(p'1, nd), cdr(tail(p'1, nd - 1))) ∧ p'2 = append(head(p'1, nd), cdr(tail(p'2, nd - 1))) ∧ nth(tail(p'1, nd - 1), 0) = nth(tail(p'2, nd - 1), 0) ∧ ¬ nth(tail(p'1, nd - 1), 1) = nth(tail(p'2, nd - 1), 1)</pre>
---	---

C Proof scripts

Simplifying, rewriting, and recording with decision procedures,

Expanding the definition of minimum?,

Applying propositional simplification,

we get 3 subgoals:

`path_factor.1.2.1.1:`

{-1}	$\forall (x: \text{nat}):$
	$x < \text{length}(p'_1) \wedge x < \text{length}(p'_2) \wedge \neg \text{nth}(p'_1, x) = \text{nth}(p'_2, x) \supset \text{nd} \leq x$
{-2}	$n' \geq 0$
{-3}	$\text{length}(p'_1) \geq 1$
{-4}	$\text{path?}(g')(p'_1)$
{-5}	$\text{length}(p'_2) \geq 1$
{-6}	$\text{path?}(g')(p'_2)$
{-7}	$\text{car}(p'_1) = \text{car}(p'_2)$
{-8}	$n' < \text{length}(p'_1)$
{-9}	$n' < \text{length}(p'_2)$
{1}	$\text{nth}(p'_1, \text{nd} - 1) = \text{nth}(p'_2, \text{nd} - 1)$
{2}	$\text{nd} = 0$
{3}	$\text{nth}(p'_1, n') = \text{nth}(p'_2, n')$

Instantiating the top quantifier in -1 with the terms: $\text{nd} - 1$,

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `path_factor.1.2.1.1`.

`path_factor.1.2.1.2:`

{-1}	$\forall (x: \text{nat}):$
	$x < \text{length}(p'_1) \wedge x < \text{length}(p'_2) \wedge \neg \text{nth}(p'_1, x) = \text{nth}(p'_2, x) \supset \text{nd} \leq x$
{-2}	$n' \geq 0$
{-3}	$\text{length}(p'_1) \geq 1$
{-4}	$\text{path?}(g')(p'_1)$
{-5}	$\text{length}(p'_2) \geq 1$
{-6}	$\text{path?}(g')(p'_2)$
{-7}	$\text{car}(p'_1) = \text{car}(p'_2)$
{-8}	$n' < \text{length}(p'_1)$
{-9}	$n' < \text{length}(p'_2)$
{1}	$p'_2 = \text{append}(\text{head}(p'_1, \text{nd}), \text{tail}(p'_2, \text{nd}))$
{2}	$\text{nd} = 0$
{3}	$\text{nth}(p'_1, n') = \text{nth}(p'_2, n')$

Using lemma `list_extensionality`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 2 subgoals:

path_factor.1.2.1.2.1:

{-1}	$\text{head}(p'_1, \text{nd}) = \text{head}(p'_2, \text{nd})$
{-2}	$\text{nd} = \text{nd}$
{-3}	$\forall (i: \text{below}(\text{length}[T](\text{head}(p'_1, \text{nd})))):$ $\text{nth}(\text{head}(p'_1, \text{nd}), i) = \text{nth}(\text{head}(p'_2, \text{nd}), i)$
{-4}	$\forall (x: \text{nat}):$ $x < \text{length}(p'_1) \wedge x < \text{length}(p'_2) \wedge \neg \text{nth}(p'_1, x) = \text{nth}(p'_2, x) \supset \text{nd} \leq x$
{-5}	$n' \geq 0$
{-6}	$\text{length}(p'_1) \geq 1$
{-7}	$\text{path}?(g')(p'_1)$
{-8}	$\text{length}(p'_2) \geq 1$
{-9}	$\text{path}?(g')(p'_2)$
{-10}	$\text{car}(p'_1) = \text{car}(p'_2)$
{-11}	$n' < \text{length}(p'_1)$
{-12}	$n' < \text{length}(p'_2)$
{1}	$p'_2 = \text{append}(\text{head}(p'_1, \text{nd}), \text{tail}(p'_2, \text{nd}))$
{2}	$\text{nd} = 0$
{3}	$\text{nth}(p'_1, n') = \text{nth}(p'_2, n')$

Replacing using formula -1,

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of path_factor.1.2.1.2.1.

path_factor.1.2.1.2.2:

{-1}	$\forall (x: \text{nat}):$ $x < \text{length}(p'_1) \wedge x < \text{length}(p'_2) \wedge \neg \text{nth}(p'_1, x) = \text{nth}(p'_2, x) \supset \text{nd} \leq x$
{-2}	$n' \geq 0$
{-3}	$\text{length}(p'_1) \geq 1$
{-4}	$\text{path}?(g')(p'_1)$
{-5}	$\text{length}(p'_2) \geq 1$
{-6}	$\text{path}?(g')(p'_2)$
{-7}	$\text{car}(p'_1) = \text{car}(p'_2)$
{-8}	$n' < \text{length}(p'_1)$
{-9}	$n' < \text{length}(p'_2)$
{1}	$\text{head}(p'_1, \text{nd}) = \text{head}(p'_2, \text{nd})$
{2}	$\forall (i: \text{below}(\text{length}[T](\text{head}(p'_1, \text{nd})))):$ $\text{nth}(\text{head}(p'_1, \text{nd}), i) = \text{nth}(\text{head}(p'_2, \text{nd}), i)$
{3}	$p'_2 = \text{append}(\text{head}(p'_1, \text{nd}), \text{tail}(p'_2, \text{nd}))$
{4}	$\text{nd} = 0$
{5}	$\text{nth}(p'_1, n') = \text{nth}(p'_2, n')$

Repeatedly Skolemizing and flattening,

Instantiating the top quantifier in -2 with the terms: i' ,

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of path_factor.1.2.1.2.2.

`path_factor.1.2.1.3:`

{-1}	$\forall (x: \text{nat}):$	
	$x < \text{length}(p'_1) \wedge x < \text{length}(p'_2) \wedge \neg \text{nth}(p'_1, x) = \text{nth}(p'_2, x) \supset \text{nd} \leq x$	
{-2}	$n' \geq 0$	
{-3}	$\text{length}(p'_1) \geq 1$	
{-4}	$\text{path?}(g')(p'_1)$	
{-5}	$\text{length}(p'_2) \geq 1$	
{-6}	$\text{path?}(g')(p'_2)$	
{-7}	$\text{car}(p'_1) = \text{car}(p'_2)$	
{-8}	$n' < \text{length}(p'_1)$	
{-9}	$n' < \text{length}(p'_2)$	
{1}	$\text{nth}(p'_1, \text{nd} - 1) = \text{nth}(p'_2, \text{nd} - 1)$	
{2}	$\text{nd} = 0$	
{3}	$\text{nth}(p'_1, n') = \text{nth}(p'_2, n')$	

Instantiating the top quantifier in -1 with the terms: $\text{nd} - 1$,
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `path_factor.1.2.1.3`.

`path_factor.1.2.2:`

{-1}	minimum?(nd,	
	$\lambda (i: \text{nat}):$	
	$i < \text{length}(p'_1) \wedge i < \text{length}(p'_2) \wedge \neg \text{nth}(p'_1, i) = \text{nth}(p'_2, i)$	
{-2}	$n' \geq 0$	
{-3}	$\text{length}(p'_1) \geq 1$	
{-4}	$\text{path?}(g')(p'_1)$	
{-5}	$\text{length}(p'_2) \geq 1$	
{-6}	$\text{path?}(g')(p'_2)$	
{-7}	$\text{car}(p'_1) = \text{car}(p'_2)$	
{-8}	$n' < \text{length}(p'_1)$	
{-9}	$n' < \text{length}(p'_2)$	
{1}	$\text{length}(\text{tail}[T](p'_2, \text{nd} - 1)) \geq 1 \wedge$ $\text{path?}(g')(\text{tail}[T](p'_2, \text{nd} - 1))$	
{2}	$\text{nd} = 0$	
{3}	$\text{nth}(p'_1, n') = \text{nth}(p'_2, n')$	

Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `path_factor.1.2.2`.

`path_factor.1.2.3:`

{-1}	minimum?(nd,	
	$\lambda (i: \text{nat}):$	
	$i < \text{length}(p'_1) \wedge i < \text{length}(p'_2) \wedge \neg \text{nth}(p'_1, i) = \text{nth}(p'_2, i)$	
{-2}	$n' \geq 0$	
{-3}	$\text{length}(p'_1) \geq 1$	
{-4}	$\text{path?}(g')(p'_1)$	
{-5}	$\text{length}(p'_2) \geq 1$	
{-6}	$\text{path?}(g')(p'_2)$	
{-7}	$\text{car}(p'_1) = \text{car}(p'_2)$	
{-8}	$n' < \text{length}(p'_1)$	
{-9}	$n' < \text{length}(p'_2)$	
{1}	$\text{nd} - 1 \geq 0$	
{2}	$\text{nd} = 0$	
{3}	$\text{nth}(p'_1, n') = \text{nth}(p'_2, n')$	

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `path_factor.1.2.3`.

`path_factor.1.2.4`:

{-1}	minimum?(nd,	$\lambda (i: \text{nat}):$
		$i < \text{length}(p'_1) \wedge i < \text{length}(p'_2) \wedge \neg \text{nth}(p'_1, i) = \text{nth}(p'_2, i)$
{-2}	$n' \geq 0$	
{-3}	$\text{length}(p'_1) \geq 1$	
{-4}	$\text{path?}(g')(p'_1)$	
{-5}	$\text{length}(p'_2) \geq 1$	
{-6}	$\text{path?}(g')(p'_2)$	
{-7}	$\text{car}(p'_1) = \text{car}(p'_2)$	
{-8}	$n' < \text{length}(p'_1)$	
{-9}	$n' < \text{length}(p'_2)$	
{1}	$\text{length}(\text{tail}[T](p'_1, \text{nd} - 1)) \geq 1 \wedge$	$\text{path?}(g')(\text{tail}[T](p'_1, \text{nd} - 1))$
{2}	$\text{nd} = 0$	
{3}	$\text{nth}(p'_1, n') = \text{nth}(p'_2, n')$	

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `path_factor.1.2.4`.

`path_factor.1.2.5`:

{-1}	minimum?(nd,	$\lambda (i: \text{nat}):$
		$i < \text{length}(p'_1) \wedge i < \text{length}(p'_2) \wedge \neg \text{nth}(p'_1, i) = \text{nth}(p'_2, i)$
{-2}	$n' \geq 0$	
{-3}	$\text{length}(p'_1) \geq 1$	
{-4}	$\text{path?}(g')(p'_1)$	
{-5}	$\text{length}(p'_2) \geq 1$	
{-6}	$\text{path?}(g')(p'_2)$	
{-7}	$\text{car}(p'_1) = \text{car}(p'_2)$	
{-8}	$n' < \text{length}(p'_1)$	
{-9}	$n' < \text{length}(p'_2)$	
{1}	$\text{nd} - 1 \geq 0$	
{2}	$\text{nd} = 0$	
{3}	$\text{nth}(p'_1, n') = \text{nth}(p'_2, n')$	

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `path_factor.1.2.5`.

`path_factor.1.2.6:`

{-1}	minimum?(nd,	$\lambda (i: \text{nat}):$ $i < \text{length}(p'_1) \wedge i < \text{length}(p'_2) \wedge \neg \text{nth}(p'_1, i) = \text{nth}(p'_2, i)$
{-2}	$n' \geq 0$	
{-3}	$\text{length}(p'_1) \geq 1$	
{-4}	$\text{path?}(g')(p'_1)$	
{-5}	$\text{length}(p'_2) \geq 1$	
{-6}	$\text{path?}(g')(p'_2)$	
{-7}	$\text{car}(p'_1) = \text{car}(p'_2)$	
{-8}	$n' < \text{length}(p'_1)$	
{-9}	$n' < \text{length}(p'_2)$	
{1}	$\text{nd} \geq 1 \wedge \text{path?}(g')(\text{head}[T](p'_1, \text{nd}))$	
{2}	$\text{nd} = 0$	
{3}	$\text{nth}(p'_1, n') = \text{nth}(p'_2, n')$	

Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `path_factor.1.2.6`.

`path_factor.2:`

{-1}	$n' \geq 0$	
{-2}	$\text{length}(p'_1) \geq 1$	
{-3}	$\text{path?}(g')(p'_1)$	
{-4}	$\text{length}(p'_2) \geq 1$	
{-5}	$\text{path?}(g')(p'_2)$	
{-6}	$\text{car}(p'_1) = \text{car}(p'_2)$	
{-7}	$n' < \text{length}(p'_1)$	
{-8}	$n' < \text{length}(p'_2)$	
{1}	$\text{nonempty?}[\text{nat}]$ $(\lambda (i: \text{nat}):$ $\quad i < \text{length}[T](p'_1) \wedge$ $\quad i < \text{length}[T](p'_2) \wedge \neg \text{nth}[T](p'_1, i) = \text{nth}[T](p'_2, i))$	
{2}	$\text{nth}(p'_1, n') = \text{nth}(p'_2, n')$	
{3}	$\exists (q, p1t, p2t: (\text{path?}(g'))):$ $\quad \text{nth}(q, \text{length}(q) - 1) = \text{car}(p1t) \wedge$ $\quad \text{nth}(q, \text{length}(q) - 1) = \text{car}(p2t) \wedge$ $\quad \text{length}(p1t) \geq 2 \wedge$ $\quad \text{length}(p2t) \geq 2 \wedge$ $\quad p'_1 = \text{append}(q, \text{cdr}(p1t)) \wedge$ $\quad p'_2 = \text{append}(q, \text{cdr}(p2t)) \wedge$ $\quad \text{nth}(p1t, 0) = \text{nth}(p2t, 0) \wedge \neg \text{nth}(p1t, 1) = \text{nth}(p2t, 1)$	

Hiding formulas: 3,

Installing automatic rewrites from: `nonempty? empty? member`

Simplifying, rewriting, and recording with decision procedures,

Instantiating quantified variables,

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `path_factor.2`.

Q.E.D.

C.97.33 Graph.node_pair_distinction_TCC1

Terse proof for `node_pair_distinction_TCC1`.

node_pair_distinction_TCC1:

$$\begin{array}{l}
\{1\} \quad \forall (g: \text{graph_type}, P: \text{PRED}[[g'nodes], (g'nodes)]): \\
\quad (\forall (n_1, n_2: (g'nodes), p_1, p_2: (\text{path?}(g))): \\
\quad \quad \neg n_1 = n_2 \wedge \\
\quad \quad \text{root_path}(g, n_1)(p_1) \wedge \\
\quad \quad \text{root_path}(g, n_2)(p_2) \wedge \neg \text{path_start}(p_1) = \text{path_start}(p_2) \\
\quad \quad \supset P(n_1, n_2)) \\
\quad \wedge \\
\quad (\forall (n_1, n_2: (g'nodes), p: (\text{path?}(g))): \\
\quad \quad \neg n_1 = n_2 \wedge \text{path_start}(p) = n_1 \wedge \text{path_end}(p) = n_2 \wedge \text{length}(p) \geq 2 \supset \\
\quad \quad P(n_1, n_2)) \\
\quad \wedge \\
\quad (\forall (n_1, n_2: (g'nodes), p: (\text{path?}(g))): \\
\quad \quad \neg n_1 = n_2 \wedge \text{path_start}(p) = n_2 \wedge \text{path_end}(p) = n_1 \wedge \text{length}(p) \geq 2 \supset \\
\quad \quad P(n_1, n_2)) \\
\quad \wedge (\forall (n: (g'nodes)): P(n, n)) \wedge \text{finite_tree?}(g) \\
\quad \supset \\
\quad (\forall (n_{11}, n_{21}, n_3: (g'nodes), p_{11}, p_{21}: (\text{path?}(g))): \\
\quad \quad g'edges(n_3, \text{path_start}(p_{11})) \wedge \neg n_{21} = n_3 \wedge \neg n_{11} = n_3 \wedge \neg n_{11} = n_{21} \supset \\
\quad \quad g'nodes(\text{path_start}[T](p_{21})))
\end{array}$$

Repeatedly Skolemizing and flattening,

Rewriting using nodes_path_start, matching in *,

This completes the proof of node_pair_distinction_TCC1.

Q.E.D.

C.97.34 Graph.node_pair_distinction

Terse proof for node_pair_distinction.

node_pair_distinction:

$$\begin{array}{l}
 \{1\} \quad \forall (g: \text{graph_type}, P: \text{PRED}[[g' \text{nodes}], [g' \text{nodes}]]) : \\
 \quad (\text{finite_tree?}(g) \wedge \\
 \quad (\forall (n: (g' \text{nodes})): P(n, n)) \wedge \\
 \quad (\forall (n_1, n_2: (g' \text{nodes}), p: (\text{path?}(g))): \\
 \quad \quad \neg n_1 = n_2 \wedge \text{path_start}(p) = n_2 \wedge \text{path_end}(p) = n_1 \wedge \text{length}(p) \geq 2 \supset \\
 \quad \quad P(n_1, n_2)) \\
 \quad \wedge \\
 \quad (\forall (n_1, n_2: (g' \text{nodes}), p: (\text{path?}(g))): \\
 \quad \quad \neg n_1 = n_2 \wedge \text{path_start}(p) = n_1 \wedge \text{path_end}(p) = n_2 \wedge \text{length}(p) \geq 2 \supset \\
 \quad \quad P(n_1, n_2)) \\
 \quad \wedge \\
 \quad (\forall (n_1, n_2: (g' \text{nodes}), p_1, p_2: (\text{path?}(g))): \\
 \quad \quad \neg n_1 = n_2 \wedge \\
 \quad \quad \text{root_path}(g, n_1)(p_1) \wedge \\
 \quad \quad \text{root_path}(g, n_2)(p_2) \wedge \neg \text{path_start}(p_1) = \text{path_start}(p_2) \\
 \quad \quad \supset P(n_1, n_2)) \\
 \quad \wedge \\
 \quad (\forall (n_1, n_2, n_3: (g' \text{nodes}), p_1, p_2: (\text{path?}(g))): \\
 \quad \quad \neg n_1 = n_2 \wedge \\
 \quad \quad \neg n_1 = n_3 \wedge \\
 \quad \quad \neg n_2 = n_3 \wedge \\
 \quad \quad g' \text{edges}(n_3, \text{path_start}(p_1)) \wedge \\
 \quad \quad g' \text{edges}(n_3, \text{path_start}(p_2)) \wedge \\
 \quad \quad \text{path_end}(p_1) = n_1 \wedge \\
 \quad \quad \text{path_end}(p_2) = n_2 \wedge \neg \text{path_start}(p_1) = \text{path_start}(p_2) \\
 \quad \quad \supset P(n_1, n_2))) \\
 \supset (\forall (n_1, n_2: (g' \text{nodes})): P(n_1, n_2))
 \end{array}$$

Repeatedly Skolemizing and flattening,

Case splitting on $n_1 \neq n_2$,

we get 2 subgoals:

node_pair_distinction.1:

{-1}	$n'_1 = n'_2$
{-2}	$g' \text{'nodes}(n'_1)$
{-3}	$g' \text{'nodes}(n'_2)$
{-4}	$\text{finite_tree?}(g')$
{-5}	$\forall (n: (g' \text{'nodes})): P'(n, n)$
{-6}	$\forall (n_1, n_2: (g' \text{'nodes}), p: (\text{path?}(g'))):$ $\neg n_1 = n_2 \wedge \text{path_start}(p) = n_2 \wedge \text{path_end}(p) = n_1 \wedge \text{length}(p) \geq 2 \supset$ $P'(n_1, n_2)$
{-7}	$\forall (n_1, n_2: (g' \text{'nodes}), p: (\text{path?}(g'))):$ $\neg n_1 = n_2 \wedge \text{path_start}(p) = n_1 \wedge \text{path_end}(p) = n_2 \wedge \text{length}(p) \geq 2 \supset$ $P'(n_1, n_2)$
{-8}	$\forall (n_1, n_2: (g' \text{'nodes}), p_1, p_2: (\text{path?}(g'))):$ $\neg n_1 = n_2 \wedge$ $\text{root_path}(g', n_1)(p_1) \wedge$ $\text{root_path}(g', n_2)(p_2) \wedge \neg \text{path_start}(p_1) = \text{path_start}(p_2)$ $\supset P'(n_1, n_2)$
{-9}	$\forall (n_1, n_2, n_3: (g' \text{'nodes}), p_1, p_2: (\text{path?}(g'))):$ $\neg n_1 = n_2 \wedge$ $\neg n_1 = n_3 \wedge$ $\neg n_2 = n_3 \wedge$ $g' \text{'edges}(n_3, \text{path_start}(p_1)) \wedge$ $g' \text{'edges}(n_3, \text{path_start}(p_2)) \wedge$ $\text{path_end}(p_1) = n_1 \wedge \text{path_end}(p_2) = n_2 \wedge \neg \text{path_start}(p_1) = \text{path_start}(p_2)$ $\supset P'(n_1, n_2)$
{1}	$P'(n'_1, n'_2)$

Instantiating the top quantifier in -5 with the terms: n'_1 ,

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of node_pair_distinction.1.

C Proof scripts

node_pair_distinction.2:

<div style="display: flex; flex-direction: column; gap: 5px;"> <div style="display: flex; align-items: flex-start;"> {-1} $g' \text{'nodes}(n'_1)$ </div> <div style="display: flex; align-items: flex-start;"> {-2} $g' \text{'nodes}(n'_2)$ </div> <div style="display: flex; align-items: flex-start;"> {-3} $\text{finite_tree?}(g')$ </div> <div style="display: flex; align-items: flex-start;"> {-4} $\forall (n: (g' \text{'nodes})): P'(n, n)$ </div> <div style="display: flex; align-items: flex-start;"> {-5} $\forall (n_1, n_2: (g' \text{'nodes}), p: (\text{path?}(g'))):$ $\quad \neg n_1 = n_2 \wedge \text{path_start}(p) = n_2 \wedge \text{path_end}(p) = n_1 \wedge \text{length}(p) \geq 2 \supset$ $\quad P'(n_1, n_2)$ </div> <div style="display: flex; align-items: flex-start;"> {-6} $\forall (n_1, n_2: (g' \text{'nodes}), p: (\text{path?}(g'))):$ $\quad \neg n_1 = n_2 \wedge \text{path_start}(p) = n_1 \wedge \text{path_end}(p) = n_2 \wedge \text{length}(p) \geq 2 \supset$ $\quad P'(n_1, n_2)$ </div> <div style="display: flex; align-items: flex-start;"> {-7} $\forall (n_1, n_2: (g' \text{'nodes}), p_1, p_2: (\text{path?}(g'))):$ $\quad \neg n_1 = n_2 \wedge$ $\quad \text{root_path}(g', n_1)(p_1) \wedge$ $\quad \text{root_path}(g', n_2)(p_2) \wedge \neg \text{path_start}(p_1) = \text{path_start}(p_2)$ $\quad \supset P'(n_1, n_2)$ </div> <div style="display: flex; align-items: flex-start;"> {-8} $\forall (n_1, n_2, n_3: (g' \text{'nodes}), p_1, p_2: (\text{path?}(g'))):$ $\quad \neg n_1 = n_2 \wedge$ $\quad \neg n_1 = n_3 \wedge$ $\quad \neg n_2 = n_3 \wedge$ $\quad g' \text{'edges}(n_3, \text{path_start}(p_1)) \wedge$ $\quad g' \text{'edges}(n_3, \text{path_start}(p_2)) \wedge$ $\quad \text{path_end}(p_1) = n_1 \wedge \text{path_end}(p_2) = n_2 \wedge \neg \text{path_start}(p_1) = \text{path_start}(p_2)$ $\quad \supset P'(n_1, n_2)$ </div> </div>	<hr style="border: 0.5px solid black; margin-bottom: 5px;"/> <div style="display: flex; flex-direction: column; gap: 5px;"> <div style="display: flex; align-items: flex-start;"> {1} $n'_1 = n'_2$ </div> <div style="display: flex; align-items: flex-start;"> {2} $P'(n'_1, n'_2)$ </div> </div>
---	---

Hiding formulas: -4,

Simplifying, rewriting, and recording with decision procedures,

Letting rp1 name path_to_root(g!1, n1!1),

Letting rp2 name path_to_root(g!1, n2!1),

Case splitting on EXISTS (i: below(length(rp1))): nth(rp1, i) = n2!1,

we get 2 subgoals:

node_pair_distinction.2.1:

<p>{-1} $\exists (i: \text{below}(\text{length}(\text{rp1}))) : \text{nth}(\text{rp1}, i) = n'_2$</p> <p>{-2} $\text{path_to_root}(g', n'_2) = \text{rp2}$</p> <p>{-3} $\text{path_to_root}(g', n'_1) = \text{rp1}$</p> <p>{-4} $g' \text{ nodes}(n'_1)$</p> <p>{-5} $g' \text{ nodes}(n'_2)$</p> <p>{-6} $\text{finite_tree?}(g')$</p> <p>{-7} $\forall (n_1, n_2 : (g' \text{ nodes}), p : (\text{path?}(g'))):$ $\neg n_1 = n_2 \wedge \text{path_start}(p) = n_2 \wedge \text{path_end}(p) = n_1 \wedge \text{length}(p) \geq 2 \supset$ $P'(n_1, n_2)$</p> <p>{-8} $\forall (n_1, n_2 : (g' \text{ nodes}), p : (\text{path?}(g'))):$ $\neg n_1 = n_2 \wedge \text{path_start}(p) = n_1 \wedge \text{path_end}(p) = n_2 \wedge \text{length}(p) \geq 2 \supset$ $P'(n_1, n_2)$</p> <p>{-9} $\forall (n_1, n_2 : (g' \text{ nodes}), p_1, p_2 : (\text{path?}(g'))):$ $\neg n_1 = n_2 \wedge$ $\text{root_path}(g', n_1)(p_1) \wedge$ $\text{root_path}(g', n_2)(p_2) \wedge \neg \text{path_start}(p_1) = \text{path_start}(p_2)$ $\supset P'(n_1, n_2)$</p> <p>{-10} $\forall (n_1, n_2, n_3 : (g' \text{ nodes}), p_1, p_2 : (\text{path?}(g'))):$ $\neg n_1 = n_2 \wedge$ $\neg n_1 = n_3 \wedge$ $\neg n_2 = n_3 \wedge$ $g' \text{ edges}(n_3, \text{path_start}(p_1)) \wedge$ $g' \text{ edges}(n_3, \text{path_start}(p_2)) \wedge$ $\text{path_end}(p_1) = n_1 \wedge \text{path_end}(p_2) = n_2 \wedge \neg \text{path_start}(p_1) = \text{path_start}(p_2)$ $\supset P'(n_1, n_2)$</p>	<hr style="border: 0.5px solid black;"/> <p>{1} $n'_1 = n'_2$</p> <p>{2} $P'(n'_1, n'_2)$</p>
--	---

Repeatedly Skolemizing and flattening,

Adding type constraints for rp1,

Expanding the definition of root_path,

Applying disjunctive simplification to flatten sequent,

Hiding formulas: -2, -3,

Installing automatic rewrites from: path_start path_end path_tail nth_tail car_tail length_tail path_list?

Instantiating the top quantifier in -10 with the terms: $n'_1, n'_2, \text{tail}(\text{rp1}, i')$,

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of node_pair_distinction.2.1.

C Proof scripts

node_pair_distinction.2.2:

{-1}	$\text{path_to_root}(g', n'_2) = \text{rp2}$
{-2}	$\text{path_to_root}(g', n'_1) = \text{rp1}$
{-3}	$g' \text{'nodes}(n'_1)$
{-4}	$g' \text{'nodes}(n'_2)$
{-5}	$\text{finite_tree?}(g')$
{-6}	$\forall (n_1, n_2: (g' \text{'nodes}), p: (\text{path?}(g'))):$ $\neg n_1 = n_2 \wedge \text{path_start}(p) = n_2 \wedge \text{path_end}(p) = n_1 \wedge \text{length}(p) \geq 2 \supset$ $P'(n_1, n_2)$
{-7}	$\forall (n_1, n_2: (g' \text{'nodes}), p: (\text{path?}(g'))):$ $\neg n_1 = n_2 \wedge \text{path_start}(p) = n_1 \wedge \text{path_end}(p) = n_2 \wedge \text{length}(p) \geq 2 \supset$ $P'(n_1, n_2)$
{-8}	$\forall (n_1, n_2: (g' \text{'nodes}), p_1, p_2: (\text{path?}(g'))):$ $\neg n_1 = n_2 \wedge$ $\text{root_path}(g', n_1)(p_1) \wedge$ $\text{root_path}(g', n_2)(p_2) \wedge \neg \text{path_start}(p_1) = \text{path_start}(p_2)$ $\supset P'(n_1, n_2)$
{-9}	$\forall (n_1, n_2, n_3: (g' \text{'nodes}), p_1, p_2: (\text{path?}(g'))):$ $\neg n_1 = n_2 \wedge$ $\neg n_1 = n_3 \wedge$ $\neg n_2 = n_3 \wedge$ $g' \text{'edges}(n_3, \text{path_start}(p_1)) \wedge$ $g' \text{'edges}(n_3, \text{path_start}(p_2)) \wedge$ $\text{path_end}(p_1) = n_1 \wedge \text{path_end}(p_2) = n_2 \wedge \neg \text{path_start}(p_1) = \text{path_start}(p_2)$ $\supset P'(n_1, n_2)$
{1}	$\exists (i: \text{below}(\text{length}(\text{rp1}))) : \text{nth}(\text{rp1}, i) = n'_2$
{2}	$n'_1 = n'_2$
{3}	$P'(n'_1, n'_2)$

Hiding formulas: -6,

Case splitting on EXISTS (i: below(length(rp2))): nth(rp2, i) = n1!1,

we get 2 subgoals:

node_pair_distinction.2.2.1:

{-1}	$\exists (i: \text{below}(\text{length}(\text{rp2}))) : \text{nth}(\text{rp2}, i) = n'_1$
{-2}	$\text{path_to_root}(g', n'_2) = \text{rp2}$
{-3}	$\text{path_to_root}(g', n'_1) = \text{rp1}$
{-4}	$g' \text{'nodes}(n'_1)$
{-5}	$g' \text{'nodes}(n'_2)$
{-6}	$\text{finite_tree?}(g')$
{-7}	$\forall (n_1, n_2: (g' \text{'nodes}), p: (\text{path?}(g'))):$ $\neg n_1 = n_2 \wedge \text{path_start}(p) = n_1 \wedge \text{path_end}(p) = n_2 \wedge \text{length}(p) \geq 2 \supset$ $P'(n_1, n_2)$
{-8}	$\forall (n_1, n_2: (g' \text{'nodes}), p_1, p_2: (\text{path?}(g'))):$ $\neg n_1 = n_2 \wedge$ $\text{root_path}(g', n_1)(p_1) \wedge$ $\text{root_path}(g', n_2)(p_2) \wedge \neg \text{path_start}(p_1) = \text{path_start}(p_2)$ $\supset P'(n_1, n_2)$
{-9}	$\forall (n_1, n_2, n_3: (g' \text{'nodes}), p_1, p_2: (\text{path?}(g'))):$ $\neg n_1 = n_2 \wedge$ $\neg n_1 = n_3 \wedge$ $\neg n_2 = n_3 \wedge$ $g' \text{'edges}(n_3, \text{path_start}(p_1)) \wedge$ $g' \text{'edges}(n_3, \text{path_start}(p_2)) \wedge$ $\text{path_end}(p_1) = n_1 \wedge \text{path_end}(p_2) = n_2 \wedge \neg \text{path_start}(p_1) = \text{path_start}(p_2)$ $\supset P'(n_1, n_2)$
{1}	$\exists (i: \text{below}(\text{length}(\text{rp1}))) : \text{nth}(\text{rp1}, i) = n'_2$
{2}	$n'_1 = n'_2$
{3}	$P'(n'_1, n'_2)$

Repeatedly Skolemizing and flattening,

Adding type constraints for rp2,

Expanding the definition of root_path,

Applying disjunctive simplification to flatten sequent,

Hiding formulas: -2, -3,

Installing automatic rewrites from: path_start path_end path_tail nth_tail car_tail length_tail path_list?

Instantiating the top quantifier in -10 with the terms: $n'_1, n'_2, \text{tail}(\text{rp2}, i')$,

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of node_pair_distinction.2.2.1.

C Proof scripts

node_pair_distinction.2.2.2:

<div style="display: flex; flex-direction: column; gap: 5px;"> <div style="display: flex; align-items: flex-start;"> {-1} $\text{path_to_root}(g', n'_2) = \text{rp2}$ </div> <div style="display: flex; align-items: flex-start;"> {-2} $\text{path_to_root}(g', n'_1) = \text{rp1}$ </div> <div style="display: flex; align-items: flex-start;"> {-3} $g' \text{'nodes}(n'_1)$ </div> <div style="display: flex; align-items: flex-start;"> {-4} $g' \text{'nodes}(n'_2)$ </div> <div style="display: flex; align-items: flex-start;"> {-5} $\text{finite_tree?}(g')$ </div> <div style="display: flex; align-items: flex-start;"> {-6} $\forall (n_1, n_2: (g' \text{'nodes}), p: (\text{path?}(g'))):$ $\quad \neg n_1 = n_2 \wedge \text{path_start}(p) = n_1 \wedge \text{path_end}(p) = n_2 \wedge \text{length}(p) \geq 2 \supset$ $\quad P'(n_1, n_2)$ </div> <div style="display: flex; align-items: flex-start;"> {-7} $\forall (n_1, n_2: (g' \text{'nodes}), p_1, p_2: (\text{path?}(g'))):$ $\quad \neg n_1 = n_2 \wedge$ $\quad \text{root_path}(g', n_1)(p_1) \wedge$ $\quad \text{root_path}(g', n_2)(p_2) \wedge \neg \text{path_start}(p_1) = \text{path_start}(p_2)$ $\quad \supset P'(n_1, n_2)$ </div> <div style="display: flex; align-items: flex-start;"> {-8} $\forall (n_1, n_2, n_3: (g' \text{'nodes}), p_1, p_2: (\text{path?}(g'))):$ $\quad \neg n_1 = n_2 \wedge$ $\quad \neg n_1 = n_3 \wedge$ $\quad \neg n_2 = n_3 \wedge$ $\quad g' \text{'edges}(n_3, \text{path_start}(p_1)) \wedge$ $\quad g' \text{'edges}(n_3, \text{path_start}(p_2)) \wedge$ $\quad \text{path_end}(p_1) = n_1 \wedge \text{path_end}(p_2) = n_2 \wedge \neg \text{path_start}(p_1) = \text{path_start}(p_2)$ $\quad \supset P'(n_1, n_2)$ </div> </div>	<hr style="border: 0.5px solid black; margin-bottom: 5px;"/> <div style="display: flex; flex-direction: column; gap: 5px;"> <div style="display: flex; align-items: flex-start;"> {1} $\exists (i: \text{below}(\text{length}(\text{rp2}))) : \text{nth}(\text{rp2}, i) = n'_1$ </div> <div style="display: flex; align-items: flex-start;"> {2} $\exists (i: \text{below}(\text{length}(\text{rp1}))) : \text{nth}(\text{rp1}, i) = n'_2$ </div> <div style="display: flex; align-items: flex-start;"> {3} $n'_1 = n'_2$ </div> <div style="display: flex; align-items: flex-start;"> {4} $P'(n'_1, n'_2)$ </div> </div>
---	---

Hiding formulas: -6,

Case splitting on $\text{path_start}(\text{rp1}) = \text{path_start}(\text{rp2})$,

we get 2 subgoals:

node_pair_distinction.2.2.2.1:

<p>{-1} path_start(rp1) = path_start(rp2) {-2} path_to_root(g', n'_2) = rp2 {-3} path_to_root(g', n'_1) = rp1 {-4} g' nodes(n'_1) {-5} g' nodes(n'_2) {-6} finite_tree?(g') {-7} $\forall (n_1, n_2: (g' \text{ nodes}), p_1, p_2: (\text{path?}(g'))):$ $\neg n_1 = n_2 \wedge$ root_path(g', n_1)(p_1) \wedge root_path(g', n_2)(p_2) $\wedge \neg \text{path_start}(p_1) = \text{path_start}(p_2)$ $\supset P'(n_1, n_2)$ {-8} $\forall (n_1, n_2, n_3: (g' \text{ nodes}), p_1, p_2: (\text{path?}(g'))):$ $\neg n_1 = n_2 \wedge$ $\neg n_1 = n_3 \wedge$ $\neg n_2 = n_3 \wedge$ g' edges(n_3, path_start(p_1)) \wedge g' edges(n_3, path_start(p_2)) \wedge path_end(p_1) = n_1 \wedge path_end(p_2) = n_2 $\wedge \neg \text{path_start}(p_1) = \text{path_start}(p_2)$ $\supset P'(n_1, n_2)$</p>	<hr style="border: 0.5px solid black;"/> <p>{1} $\exists (i: \text{below}(\text{length}(\text{rp2}))) : \text{nth}(\text{rp2}, i) = n'_1$ {2} $\exists (i: \text{below}(\text{length}(\text{rp1}))) : \text{nth}(\text{rp1}, i) = n'_2$ {3} $n'_1 = n'_2$ {4} $P'(n'_1, n'_2)$</p>
--	---

Hiding formulas: -7,

Installing automatic rewrites from: path_list? length_cdr path_start nth path_end concat_path
 nth_append_right length_append nth_append_left

Using lemma path_factor,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 2 subgoals:

C Proof scripts

node_pair_distinction.2.2.2.1.1:

<pre> {-1} car(rp1) = car(rp2) {-2} ∃ (q, p1t, p2t: (path?(g'))): concatable?(q, p1t) ∧ concatable?(q, p2t) ∧ length(p1t) ≥ 2 ∧ length(p2t) ≥ 2 ∧ rp1 = append(q, cdr(p1t)) ∧ rp2 = append(q, cdr(p2t)) ∧ car(p1t) = car(p2t) ∧ ¬ car(cdr(p1t)) = car(cdr(p2t)) {-3} path_to_root(g', n'_2) = rp2 {-4} path_to_root(g', n'_1) = rp1 {-5} g' nodes(n'_1) {-6} g' nodes(n'_2) {-7} finite_tree?(g') {-8} ∀ (n1, n2, n3: (g' nodes), p1, p2: (path?(g'))): ¬ n1 = n2 ∧ ¬ n1 = n3 ∧ ¬ n2 = n3 ∧ g' edges(n3, car(p1)) ∧ g' edges(n3, car(p2)) ∧ nth(p1, length(p1) - 1) = n1 ∧ nth(p2, length(p2) - 1) = n2 ∧ ¬ car(p1) = car(p2) ⊃ P'(n1, n2) </pre>	<hr style="border: 0.5px solid black;"/> <pre> {1} ∃ (i: below(length(rp2))): nth(rp2, i) = n'_1 {2} ∃ (i: below(length(rp1))): nth(rp1, i) = n'_2 {3} n'_1 = n'_2 {4} P'(n'_1, n'_2) </pre>
---	--

Repeatedly Skolemizing and flattening,

Simplifying, rewriting, and recording with decision procedures,

Hiding formulas: -3, -5,

Using lemma path_cdr,

Using lemma path_cdr,

Simplifying, rewriting, and recording with decision procedures,

Instantiating the top quantifier in -20 with the terms: n'_1 , n'_2 , $\text{car}(p1t')$, $\text{cdr}(p1t')$, $\text{cdr}(p2t')$,

we get 2 subgoals:

node_pair_distinction.2.2.2.1.1.1:

{-1}	$\text{path?}(g')(\text{cdr}(p2t'))$
{-2}	$\text{path?}(g')(\text{cdr}(p1t'))$
{-3}	$\text{length}(q') \geq 1$
{-4}	$\text{path?}(g')(q')$
{-5}	$\text{path?}(g')(p1t')$
{-6}	$\text{path?}(g')(p2t')$
{-7}	$\text{car}(rp1) = \text{car}(rp2)$
{-8}	$\text{concatable?}(q', p1t')$
{-9}	$\text{concatable?}(q', p2t')$
{-10}	$\text{length}(p1t') \geq 2$
{-11}	$\text{length}(p2t') \geq 2$
{-12}	$rp1 = \text{append}(q', \text{cdr}(p1t'))$
{-13}	$rp2 = \text{append}(q', \text{cdr}(p2t'))$
{-14}	$\text{car}(p1t') = \text{car}(p2t')$
{-15}	$\text{path_to_root}(g', n'_2) = rp2$
{-16}	$\text{path_to_root}(g', n'_1) = rp1$
{-17}	$g' \text{'nodes}(n'_1)$
{-18}	$g' \text{'nodes}(n'_2)$
{-19}	$\text{finite_tree?}(g')$
{-20}	$\neg n'_1 = n'_2 \wedge$ $\neg n'_1 = \text{car}(p1t') \wedge$ $\neg n'_2 = \text{car}(p1t') \wedge$ $g' \text{'edges}(\text{car}(p1t'), \text{car}(\text{cdr}(p1t'))) \wedge$ $g' \text{'edges}(\text{car}(p1t'), \text{car}(\text{cdr}(p2t'))) \wedge$ $\text{nth}(\text{cdr}(p1t'), \text{length}(\text{cdr}(p1t')) - 1) = n'_1 \wedge$ $\text{nth}(\text{cdr}(p2t'), \text{length}(\text{cdr}(p2t')) - 1) = n'_2 \wedge$ $\neg \text{car}(\text{cdr}(p1t')) = \text{car}(\text{cdr}(p2t'))$ $\supset P'(n'_1, n'_2)$
{1}	$\text{car}(\text{cdr}(p1t')) = \text{car}(\text{cdr}(p2t'))$
{2}	$\exists (i: \text{below}(\text{length}(rp2))): \text{nth}(rp2, i) = n'_1$
{3}	$\exists (i: \text{below}(\text{length}(rp1))): \text{nth}(rp1, i) = n'_2$
{4}	$n'_1 = n'_2$
{5}	$P'(n'_1, n'_2)$

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 6 subgoals:

C Proof scripts

node_pair_distinction.2.2.2.1.1.1.1.1:

{-1}	path?(g')(cdr(p2t'))
{-2}	path?(g')(cdr(p1t'))
{-3}	length(q') ≥ 1
{-4}	path?(g')(q')
{-5}	path?(g')(p1t')
{-6}	path?(g')(p2t')
{-7}	car(rp1) = car(rp2)
{-8}	concatable?(q', p1t')
{-9}	concatable?(q', p2t')
{-10}	length(p1t') ≥ 2
{-11}	length(p2t') ≥ 2
{-12}	rp1 = append(q', cdr(p1t'))
{-13}	rp2 = append(q', cdr(p2t'))
{-14}	car(p1t') = car(p2t')
{-15}	path_to_root(g', n ₂ ') = rp2
{-16}	path_to_root(g', n ₁ ') = rp1
{-17}	g'ˆnodes(n ₁ ')
{-18}	g'ˆnodes(n ₂ ')
{-19}	finite_tree?(g')
{-20}	n ₁ ' = car(p1t')
{1}	car(cdr(p1t')) = car(cdr(p2t'))
{2}	∃ (i: below(length(rp2))): nth(rp2, i) = n ₁ '
{3}	∃ (i: below(length(rp1))): nth(rp1, i) = n ₂ '
{4}	n ₁ ' = n ₂ '
{5}	P'(n ₁ ', n ₂ ')

Keeping (-4 -6 -9 -11 -13 -14 -20 2) and hiding *,

Expanding the definition of concatable?,

Replacing using formula -5,

Instantiating the top quantifier in 1 with the terms: length(q') - 1,

we get 2 subgoals:

node_pair_distinction.2.2.2.1.1.1.1.1:

{-1}	path?(g')(q')
{-2}	path?(g')(p2t')
{-3}	path_end(q') = path_start(p2t')
{-4}	length(p2t') ≥ 2
{-5}	rp2 = append(q', cdr(p2t'))
{-6}	car(p1t') = car(p2t')
{-7}	n ₁ ' = car(p1t')
{1}	nth(append(q', cdr(p2t')), length(q') - 1) = n ₁ '

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of node_pair_distinction.2.2.2.1.1.1.1.1.

node_pair_distinction.2.2.2.1.1.1.1.2:

{-1}	path?(g')(q')
{-2}	path?(g')(p2t')
{-3}	path_end(q') = path_start(p2t')
{-4}	length(p2t') ≥ 2
{-5}	rp2 = append(q', cdr(p2t'))
{-6}	car(p1t') = car(p2t')
{-7}	n'_1 = car(p1t')
{1}	length[T](q') - 1 < length[T](rp2)

Replacing using formula -5,

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of node_pair_distinction.2.2.2.1.1.1.1.2.

node_pair_distinction.2.2.2.1.1.1.2:

{-1}	path?(g')(cdr(p2t'))
{-2}	path?(g')(cdr(p1t'))
{-3}	length(q') ≥ 1
{-4}	path?(g')(q')
{-5}	path?(g')(p1t')
{-6}	path?(g')(p2t')
{-7}	car(rp1) = car(rp2)
{-8}	concatable?(q', p1t')
{-9}	concatable?(q', p2t')
{-10}	length(p1t') ≥ 2
{-11}	length(p2t') ≥ 2
{-12}	rp1 = append(q', cdr(p1t'))
{-13}	rp2 = append(q', cdr(p2t'))
{-14}	car(p1t') = car(p2t')
{-15}	path_to_root(g', n'_2) = rp2
{-16}	path_to_root(g', n'_1) = rp1
{-17}	g' nodes(n'_1)
{-18}	g' nodes(n'_2)
{-19}	finite_tree?(g')
{-20}	n'_2 = car(p1t')
{1}	car(cdr(p1t')) = car(cdr(p2t'))
{2}	∃ (i: below(length(rp2))): nth(rp2, i) = n'_1
{3}	∃ (i: below(length(rp1))): nth(rp1, i) = n'_2
{4}	n'_1 = n'_2
{5}	P'(n'_1, n'_2)

Keeping (-4 -5 -8 -10 -12 -20 3) and hiding *,

Expanding the definition of concatable?,

Replacing using formula -5,

Instantiating the top quantifier in 1 with the terms: length(q') - 1,

we get 2 subgoals:

C Proof scripts

node_pair_distinction.2.2.2.1.1.1.2.1:

{-1}	path?(g')(q')
{-2}	path?(g')(p1t')
{-3}	path_end(q') = path_start(p1t')
{-4}	length(p1t') ≥ 2
{-5}	rp1 = append(q', cdr(p1t'))
{-6}	n'_2 = car(p1t')
{1}	nth(append(q', cdr(p1t')), length(q') - 1) = n'_2

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of node_pair_distinction.2.2.2.1.1.1.2.1.

node_pair_distinction.2.2.2.1.1.1.2.2:

{-1}	path?(g')(q')
{-2}	path?(g')(p1t')
{-3}	path_end(q') = path_start(p1t')
{-4}	length(p1t') ≥ 2
{-5}	rp1 = append(q', cdr(p1t'))
{-6}	n'_2 = car(p1t')
{1}	length[T](q') - 1 < length[T](rp1)

Replacing using formula -5,

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of node_pair_distinction.2.2.2.1.1.1.2.2.

node_pair_distinction.2.2.2.1.1.1.3:

{-1}	path?(g')(cdr(p2t'))
{-2}	path?(g')(cdr(p1t'))
{-3}	length(q') ≥ 1
{-4}	path?(g')(q')
{-5}	path?(g')(p1t')
{-6}	path?(g')(p2t')
{-7}	car(rp1) = car(rp2)
{-8}	concatable?(q', p1t')
{-9}	concatable?(q', p2t')
{-10}	length(p1t') ≥ 2
{-11}	length(p2t') ≥ 2
{-12}	rp1 = append(q', cdr(p1t'))
{-13}	rp2 = append(q', cdr(p2t'))
{-14}	car(p1t') = car(p2t')
{-15}	path_to_root(g', n'_2) = rp2
{-16}	path_to_root(g', n'_1) = rp1
{-17}	g' nodes(n'_1)
{-18}	g' nodes(n'_2)
{-19}	finite_tree?(g')
{1}	car(cdr(p1t')) = car(cdr(p2t'))
{2}	∃ (i: below(length(rp2))): nth(rp2, i) = n'_1
{3}	∃ (i: below(length(rp1))): nth(rp1, i) = n'_2
{4}	n'_1 = n'_2
{5}	nth(cdr(p2t'), length[T](p2t') - 2) = n'_2
{6}	P'(n'_1, n'_2)

Keeping (-4 -6 -9 -13 -15 5) and hiding *,

Adding type constraints for rp2,

Expanding the definition of `root_path`,

Applying disjunctive simplification to flatten sequent,

Hiding formulas: -1, -2, -3, -9,

Replacing using formula -5,

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `node_pair_distinction.2.2.2.1.1.1.3`.

`node_pair_distinction.2.2.2.1.1.1.4`:

{-1}	<code>path?(g')(cdr(p2t'))</code>
{-2}	<code>path?(g')(cdr(p1t'))</code>
{-3}	<code>length(q') ≥ 1</code>
{-4}	<code>path?(g')(q')</code>
{-5}	<code>path?(g')(p1t')</code>
{-6}	<code>path?(g')(p2t')</code>
{-7}	<code>car(rp1) = car(rp2)</code>
{-8}	<code>concatable?(q', p1t')</code>
{-9}	<code>concatable?(q', p2t')</code>
{-10}	<code>length(p1t') ≥ 2</code>
{-11}	<code>length(p2t') ≥ 2</code>
{-12}	<code>rp1 = append(q', cdr(p1t'))</code>
{-13}	<code>rp2 = append(q', cdr(p2t'))</code>
{-14}	<code>car(p1t') = car(p2t')</code>
{-15}	<code>path_to_root(g', n₂') = rp2</code>
{-16}	<code>path_to_root(g', n₁') = rp1</code>
{-17}	<code>g' nodes(n₁')</code>
{-18}	<code>g' nodes(n₂')</code>
{-19}	<code>finite_tree?(g')</code>
{1}	<code>car(cdr(p1t')) = car(cdr(p2t'))</code>
{2}	<code>∃ (i: below(length(rp2))): nth(rp2, i) = n₁'</code>
{3}	<code>∃ (i: below(length(rp1))): nth(rp1, i) = n₂'</code>
{4}	<code>n₁' = n₂'</code>
{5}	<code>nth(cdr(p1t'), length[T](p1t') - 2) = n₁'</code>
{6}	<code>P'(n₁', n₂')</code>

Keeping (-4 -5 -8 -12 -16 5) and hiding *,

Adding type constraints for `rp1`,

Expanding the definition of `root_path`,

Applying disjunctive simplification to flatten sequent,

Hiding formulas: -1, -2, -3, -9,

Replacing using formula -5,

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `node_pair_distinction.2.2.2.1.1.1.4`.

C Proof scripts

node_pair_distinction.2.2.2.1.1.1.5:

{-1}	path?(g')(cdr(p2t'))
{-2}	path?(g')(cdr(p1t'))
{-3}	length(q') ≥ 1
{-4}	path?(g')(q')
{-5}	path?(g')(p1t')
{-6}	path?(g')(p2t')
{-7}	car(rp1) = car(rp2)
{-8}	concatable?(q', p1t')
{-9}	concatable?(q', p2t')
{-10}	length(p1t') ≥ 2
{-11}	length(p2t') ≥ 2
{-12}	rp1 = append(q', cdr(p1t'))
{-13}	rp2 = append(q', cdr(p2t'))
{-14}	car(p1t') = car(p2t')
{-15}	path_to_root(g', n ₂ ') = rp2
{-16}	path_to_root(g', n ₁ ') = rp1
{-17}	g' ' nodes(n ₁ ')
{-18}	g' ' nodes(n ₂ ')
{-19}	finite_tree?(g')
{1}	car(cdr(p1t')) = car(cdr(p2t'))
{2}	∃ (i: below(length(rp2))): nth(rp2, i) = n ₁ '
{3}	∃ (i: below(length(rp1))): nth(rp1, i) = n ₂ '
{4}	n ₁ ' = n ₂ '
{5}	g' ' edges(car(p1t'), car(cdr(p2t')))
{6}	P'(n ₁ ', n ₂ ')

Keeping (-6 -14 5) and hiding *,

Expanding the definition of path?,

Applying disjunctive simplification to flatten sequent,

Instantiating the top quantifier in -2 with the terms: 0,

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of node_pair_distinction.2.2.2.1.1.1.5.

node_pair_distinction.2.2.2.1.1.1.6:

{-1}	path?(g')(cdr(p2t'))
{-2}	path?(g')(cdr(p1t'))
{-3}	length(q') ≥ 1
{-4}	path?(g')(q')
{-5}	path?(g')(p1t')
{-6}	path?(g')(p2t')
{-7}	car(rp1) = car(rp2)
{-8}	concatable?(q', p1t')
{-9}	concatable?(q', p2t')
{-10}	length(p1t') ≥ 2
{-11}	length(p2t') ≥ 2
{-12}	rp1 = append(q', cdr(p1t'))
{-13}	rp2 = append(q', cdr(p2t'))
{-14}	car(p1t') = car(p2t')
{-15}	path_to_root(g', n ₂ ') = rp2
{-16}	path_to_root(g', n ₁ ') = rp1
{-17}	g' ^ nodes(n ₁ ')
{-18}	g' ^ nodes(n ₂ ')
{-19}	finite_tree?(g')
{1}	car(cdr(p1t')) = car(cdr(p2t'))
{2}	∃ (i: below(length(rp2))): nth(rp2, i) = n ₁ '
{3}	∃ (i: below(length(rp1))): nth(rp1, i) = n ₂ '
{4}	n ₁ ' = n ₂ '
{5}	g' ^ edges(car(p1t'), car(cdr(p1t')))
{6}	P'(n ₁ ', n ₂ ')

Keeping (-5 5) and hiding *,

Expanding the definition of path?,

Applying disjunctive simplification to flatten sequent,

Instantiating the top quantifier in -2 with the terms: 0,

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of node_pair_distinction.2.2.2.1.1.1.6.

C Proof scripts

node_pair_distinction.2.2.2.1.1.2:

{-1}	path?(g')(cdr(p2t'))
{-2}	path?(g')(cdr(p1t'))
{-3}	length(q') ≥ 1
{-4}	path?(g')(q')
{-5}	path?(g')(p1t')
{-6}	path?(g')(p2t')
{-7}	car(rp1) = car(rp2)
{-8}	concatable?(q', p1t')
{-9}	concatable?(q', p2t')
{-10}	length(p1t') ≥ 2
{-11}	length(p2t') ≥ 2
{-12}	rp1 = append(q', cdr(p1t'))
{-13}	rp2 = append(q', cdr(p2t'))
{-14}	car(p1t') = car(p2t')
{-15}	path_to_root(g', n ₂ ') = rp2
{-16}	path_to_root(g', n ₁ ') = rp1
{-17}	g' ' nodes(n ₁ ')
{-18}	g' ' nodes(n ₂ ')
{-19}	finite_tree?(g')
{1}	g' ' nodes(car [T](p1t'))
{2}	car(cdr(p1t')) = car(cdr(p2t'))
{3}	∃ (i: below(length(rp2))): nth(rp2, i) = n ₁ '
{4}	∃ (i: below(length(rp1))): nth(rp1, i) = n ₂ '
{5}	n ₁ ' = n ₂ '
{6}	P'(n ₁ ', n ₂ ')

Keeping (-5 1) and hiding *,

Expanding the definition of path?,

Applying disjunctive simplification to flatten sequent,

Instantiating the top quantifier in -1 with the terms: 0,

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of node_pair_distinction.2.2.2.1.1.2.

node_pair_distinction.2.2.2.1.2:

{-1}	$\text{car}(\text{rp1}) = \text{car}(\text{rp2})$
{-2}	$\text{path_to_root}(g', n'_2) = \text{rp2}$
{-3}	$\text{path_to_root}(g', n'_1) = \text{rp1}$
{-4}	$g' \text{'nodes}(n'_1)$
{-5}	$g' \text{'nodes}(n'_2)$
{-6}	$\text{finite_tree?}(g')$
{-7}	$\forall (n_1, n_2, n_3: (g' \text{'nodes}), p_1, p_2: (\text{path?}(g'))):$ $\quad \neg n_1 = n_2 \wedge$ $\quad \neg n_1 = n_3 \wedge$ $\quad \neg n_2 = n_3 \wedge$ $\quad g' \text{'edges}(n_3, \text{car}(p_1)) \wedge$ $\quad g' \text{'edges}(n_3, \text{car}(p_2)) \wedge$ $\quad \text{nth}(p_1, \text{length}(p_1) - 1) = n_1 \wedge$ $\quad \text{nth}(p_2, \text{length}(p_2) - 1) = n_2 \wedge \neg \text{car}(p_1) = \text{car}(p_2)$ $\quad \supset P'(n_1, n_2)$
{1}	$\exists (n: \text{nat}): n < \text{length}(\text{rp1}) \wedge n < \text{length}(\text{rp2}) \wedge \neg \text{nth}(\text{rp1}, n) = \text{nth}(\text{rp2}, n)$
{2}	$\exists (i: \text{below}(\text{length}(\text{rp2}))) : \text{nth}(\text{rp2}, i) = n'_1$
{3}	$\exists (i: \text{below}(\text{length}(\text{rp1}))) : \text{nth}(\text{rp1}, i) = n'_2$
{4}	$n'_1 = n'_2$
{5}	$P'(n'_1, n'_2)$

Hiding formulas: -1, -4, -5, -6, -7, 5,

Case splitting on $\text{length}(\text{rp1}) < \text{length}(\text{rp2})$,

we get 2 subgoals:

node_pair_distinction.2.2.2.1.2.1:

{-1}	$\text{length}(\text{rp1}) < \text{length}(\text{rp2})$
{-2}	$\text{path_to_root}(g', n'_2) = \text{rp2}$
{-3}	$\text{path_to_root}(g', n'_1) = \text{rp1}$
{1}	$\exists (n: \text{nat}): n < \text{length}(\text{rp1}) \wedge n < \text{length}(\text{rp2}) \wedge \neg \text{nth}(\text{rp1}, n) = \text{nth}(\text{rp2}, n)$
{2}	$\exists (i: \text{below}(\text{length}(\text{rp2}))) : \text{nth}(\text{rp2}, i) = n'_1$
{3}	$\exists (i: \text{below}(\text{length}(\text{rp1}))) : \text{nth}(\text{rp1}, i) = n'_2$
{4}	$n'_1 = n'_2$

Adding type constraints for rp1,

Expanding the definition of root_path,

Applying disjunctive simplification to flatten sequent,

Hiding formulas: -2, -3,

Simplifying, rewriting, and recording with decision procedures,

Instantiating the top quantifier in 1 with the terms: $\text{length}(\text{rp1}) - 1$,

Simplifying, rewriting, and recording with decision procedures,

Instantiating the top quantifier in 1 with the terms: $\text{length}(\text{rp1}) - 1$,

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of node_pair_distinction.2.2.2.1.2.1.

node_pair_distinction.2.2.2.1.2.2:

{-1}	$\text{path_to_root}(g', n'_2) = \text{rp2}$
{-2}	$\text{path_to_root}(g', n'_1) = \text{rp1}$
{1}	$\text{length}(\text{rp1}) < \text{length}(\text{rp2})$
{2}	$\exists (n: \text{nat}): n < \text{length}(\text{rp1}) \wedge n < \text{length}(\text{rp2}) \wedge \neg \text{nth}(\text{rp1}, n) = \text{nth}(\text{rp2}, n)$
{3}	$\exists (i: \text{below}(\text{length}(\text{rp2}))) : \text{nth}(\text{rp2}, i) = n'_1$
{4}	$\exists (i: \text{below}(\text{length}(\text{rp1}))) : \text{nth}(\text{rp1}, i) = n'_2$
{5}	$n'_1 = n'_2$

C Proof scripts

Adding type constraints for rp2,
 Expanding the definition of root_path,
 Applying disjunctive simplification to flatten sequent,
 Hiding formulas: -2, -3,
 Simplifying, rewriting, and recording with decision procedures,
 Instantiating the top quantifier in 2 with the terms: length(rp2) - 1,
 Simplifying, rewriting, and recording with decision procedures,
 Instantiating the top quantifier in 3 with the terms: length(rp2) - 1,
 Simplifying, rewriting, and recording with decision procedures,
 This completes the proof of node_pair_distinction.2.2.2.1.2.2.
 node_pair_distinction.2.2.2.2:

{-1} path_to_root(g' , n'_2) = rp2 {-2} path_to_root(g' , n'_1) = rp1 {-3} g' 'nodes(n'_1) {-4} g' 'nodes(n'_2) {-5} finite_tree?(g') {-6} $\forall (n_1, n_2: (g'$ 'nodes), $p_1, p_2: (\text{path?}(g'))$): $\neg n_1 = n_2 \wedge$ root_path(g' , n_1)(p_1) \wedge root_path(g' , n_2)(p_2) $\wedge \neg$ path_start(p_1) = path_start(p_2) $\supset P'(n_1, n_2)$ {-7} $\forall (n_1, n_2, n_3: (g'$ 'nodes), $p_1, p_2: (\text{path?}(g'))$): $\neg n_1 = n_2 \wedge$ $\neg n_1 = n_3 \wedge$ $\neg n_2 = n_3 \wedge$ g' 'edges(n_3 , path_start(p_1)) \wedge g' 'edges(n_3 , path_start(p_2)) \wedge path_end(p_1) = $n_1 \wedge$ path_end(p_2) = $n_2 \wedge \neg$ path_start(p_1) = path_start(p_2) $\supset P'(n_1, n_2)$	<hr style="border: 0.5px solid black;"/> {1} path_start(rp1) = path_start(rp2) {2} $\exists (i: \text{below}(\text{length}(\text{rp2})))$: nth(rp2, i) = n'_1 {3} $\exists (i: \text{below}(\text{length}(\text{rp1})))$: nth(rp1, i) = n'_2 {4} $n'_1 = n'_2$ {5} $P'(n'_1, n'_2)$
---	---

Hiding formulas: -7,
 Instantiating the top quantifier in -6 with the terms: n'_1 , n'_2 , rp1, rp2,
 Simplifying, rewriting, and recording with decision procedures,
 This completes the proof of node_pair_distinction.2.2.2.2.
 Q.E.D.

C.97.35 Graph.symmetric_node_pair_distinction_TCC1

Terse proof for symmetric_node_pair_distinction_TCC1.

symmetric_node_pair_distinction_TCC1:

```

{1}  ∀ (g: graph_type, P: PRED[[g'nodes), (g'nodes)]]):
      (∀ (n1, n2: (g'nodes), p1, p2: (path?(g))):
        ¬ n1 = n2 ∧
        root_path(g, n1)(p1) ∧
        root_path(g, n2)(p2) ∧ ¬ path_start(p1) = path_start(p2)
        ⊃ P(n1, n2))
      ∧
      (∀ (n1, n2: (g'nodes), p: (path?(g))):
        ¬ n1 = n2 ∧ path_start(p) = n2 ∧ path_end(p) = n1 ∧ length(p) ≥ 2 ⊃
        P(n1, n2))
      ∧ (∀ (n: (g'nodes)): P(n, n)) ∧ symmetric?(P) ∧ finite_tree?(g)
      ⊃
      (∀ (n11, n21, n3: (g'nodes), p11, p21: (path?(g))):
        g'edges(n3, path_start(p11)) ∧ ¬ n21 = n3 ∧ ¬ n11 = n3 ∧ ¬ n11 = n21 ⊃
        g'nodes(path_start[T](p21)))

```

Repeatedly Skolemizing and flattening,

Rewriting using nodes_path_start, matching in *,

This completes the proof of symmetric_node_pair_distinction_TCC1.

Q.E.D.

C.97.36 Graph.symmetric_node_pair_distinction

Terse proof for symmetric_node_pair_distinction.

symmetric_node_pair_distinction:

```

{1}  ∀ (g: graph_type, P: PRED[[g'nodes), (g'nodes)]]):
      (finite_tree?(g) ∧
      symmetric?(P) ∧
      (∀ (n: (g'nodes)): P(n, n)) ∧
      (∀ (n1, n2: (g'nodes), p: (path?(g))):
        ¬ n1 = n2 ∧ path_start(p) = n2 ∧ path_end(p) = n1 ∧ length(p) ≥ 2 ⊃
        P(n1, n2))
      ∧
      (∀ (n1, n2: (g'nodes), p1, p2: (path?(g))):
        ¬ n1 = n2 ∧
        root_path(g, n1)(p1) ∧
        root_path(g, n2)(p2) ∧ ¬ path_start(p1) = path_start(p2)
        ⊃ P(n1, n2))
      ∧
      (∀ (n1, n2, n3: (g'nodes), p1, p2: (path?(g))):
        ¬ n1 = n2 ∧
        ¬ n1 = n3 ∧
        ¬ n2 = n3 ∧
        g'edges(n3, path_start(p1)) ∧
        g'edges(n3, path_start(p2)) ∧
        path_end(p1) = n1 ∧
        path_end(p2) = n2 ∧ ¬ path_start(p1) = path_start(p2)
        ⊃ P(n1, n2)))
      ⊃ (∀ (n1, n2: (g'nodes)): P(n1, n2))

```

Skolemizing and flattening,
 Using lemma `node_pair_distinction`,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 Hiding formulas: -2, -4, -5, 2,
 Repeatedly Skolemizing and flattening,
 Instantiating the top quantifier in -6 with the terms: n'_2, n'_1, p' ,
 Simplifying, rewriting, and recording with decision procedures,
 Expanding the definition of `symmetric?`,
 Instantiating the top quantifier in -10 with the terms: n'_2, n'_1 ,
 Simplifying, rewriting, and recording with decision procedures,
 This completes the proof of `symmetric_node_pair_distinction`.
 Q.E.D.

C.98 Proofs for Graph_Relations (graph.pvs)

C.98.1 Graph_Relations.trans_closure_char_TCC1

Terse proof for `trans_closure_char_TCC1`.

`trans_closure_char_TCC1`:

$$\{1\} \quad \forall (x, y: T, l: \text{list}[T]): \text{length}(l) \geq 2 \supset \text{path_list?}[T](l)$$

Repeatedly Skolemizing and flattening,
 Installing automatic rewrites from: `path_list? length`
 Simplifying, rewriting, and recording with decision procedures,
 This completes the proof of `trans_closure_char_TCC1`.
 Q.E.D.

C.98.2 Graph_Relations.trans_closure_char_TCC2

Terse proof for `trans_closure_char_TCC2`.

`trans_closure_char_TCC2`:

$$\{1\} \quad \forall (x, y: T, l: \text{list}[T]):$$

$$\text{path_end}(l) = y \wedge \text{path_start}(l) = x \wedge \text{length}(l) \geq 2 \supset$$

$$(\forall (i: \text{below}(\text{length}[T](l) - 1)): i < \text{length}[T](l))$$

Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of `trans_closure_char_TCC2`.
 Q.E.D.

C.98.3 Graph_Relations.trans_closure_char_TCC3

Terse proof for `trans_closure_char_TCC3`.

`trans_closure_char_TCC3`:

$$\{1\} \quad \forall (x, y: T, l: \text{list}[T]):$$

$$\text{path_end}(l) = y \wedge \text{path_start}(l) = x \wedge \text{length}(l) \geq 2 \supset$$

$$(\forall (i: \text{below}(\text{length}[T](l) - 1)): i + 1 < \text{length}[T](l))$$

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `trans_closure_char_TCC3`.

Q.E.D.

C.98.4 Graph_Relations.trans_closure_char

Terse proof for `trans_closure_char`.

`trans_closure_char`:

{1}	$\forall (R: \text{PRED}[[T, T]], x, y: T):$ $\text{transitive_closure}(R)(x, y) \equiv$ $(\exists (l: \text{list}[T]):$ $\text{length}(l) \geq 2 \wedge$ $\text{path_start}(l) = x \wedge$ $\text{path_end}(l) = y \wedge$ $(\forall (i: \text{below}(\text{length}(l) - 1)):$ $R(\text{nth}(l, i), \text{nth}(l, i + 1))))$
-----	--

Installing automatic rewrites from: `path_list?` `length` `length_is_cons` `length_cdr`

Repeatedly Skolemizing and flattening,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 2 subgoals:

`trans_closure_char.1`:

{-1}	$\text{transitive_closure}(R')(x', y')$
{1}	$\exists (l: \text{list}[T]):$ $\text{length}(l) \geq 2 \wedge$ $\text{path_start}(l) = x' \wedge$ $\text{path_end}(l) = y' \wedge$ $(\forall (i: \text{below}(\text{length}(l) - 1)): R'(\text{nth}(l, i), \text{nth}(l, 1 + i)))$

Expanding the definition of `transitive_closure`,

Instantiating the top quantifier in -1 with the terms: $\lambda (x, y: T): \exists (l: \text{list}[T]): \text{length}(l) \geq 2 \wedge \text{path_start}(l) = x \wedge \text{path_end}(l) = y \wedge (\forall (i: \text{below}(\text{length}(l) - 1)): R'(\text{nth}(l, i), \text{nth}(l, 1 + i)))$,

we get 3 subgoals:

`trans_closure_char.1.1:`

$$\begin{array}{l}
 \{ -1 \} \quad (R' \subseteq \lambda (x, y : T) : \exists (l : \text{list}[T]) : \text{length}(l) \geq 2 \wedge \text{path_start}(l) = x \wedge \text{path_end}(l) = \\
 \quad \wedge \\
 \quad \text{transitive?}(\lambda (x, y : T) : \\
 \quad \quad \exists (l : \text{list}[T]) : \\
 \quad \quad \quad \text{length}(l) \geq 2 \wedge \\
 \quad \quad \quad \text{path_start}(l) = x \wedge \\
 \quad \quad \quad \text{path_end}(l) = y \wedge \\
 \quad \quad \quad (\forall (i : \text{below}(\text{length}(l) - 1)) : \\
 \quad \quad \quad \quad R'(\text{nth}(l, i), \text{nth}(l, 1 + i)))) \\
 \quad \supset \\
 \quad (\exists (l : \text{list}[T]) : \\
 \quad \quad \text{length}(l) \geq 2 \wedge \\
 \quad \quad \text{path_start}(l) = x' \wedge \\
 \quad \quad \text{path_end}(l) = y' \wedge \\
 \quad \quad (\forall (i : \text{below}(\text{length}(l) - 1)) : \\
 \quad \quad \quad R'(\text{nth}(l, i), \text{nth}(l, 1 + i)))) \\
 \hline
 \{ 1 \} \quad \exists (l : \text{list}[T]) : \\
 \quad \text{length}(l) \geq 2 \wedge \\
 \quad \text{path_start}(l) = x' \wedge \\
 \quad \text{path_end}(l) = y' \wedge \\
 \quad (\forall (i : \text{below}(\text{length}(l) - 1)) : R'(\text{nth}(l, i), \text{nth}(l, 1 + i)))
 \end{array}$$

Applying propositional simplification,

we get 2 subgoals:

`trans_closure_char.1.1.1:`

$$\begin{array}{l}
 \{ 1 \} \quad (R' \subseteq \lambda (x, y : T) : \exists (l : \text{list}[T]) : \text{length}(l) \geq 2 \wedge \text{path_start}(l) = x \wedge \text{path_end}(l) = \\
 \{ 2 \} \quad \exists (l : \text{list}[T]) : \\
 \quad \text{length}(l) \geq 2 \wedge \\
 \quad \text{path_start}(l) = x' \wedge \\
 \quad \text{path_end}(l) = y' \wedge \\
 \quad (\forall (i : \text{below}(\text{length}(l) - 1)) : R'(\text{nth}(l, i), \text{nth}(l, 1 + i)))
 \end{array}$$

Hiding formulas: 2,

Expanding the definition(s) of (subset? member),

Repeatedly Skolemizing and flattening,

Instantiating the top quantifier in 1 with the terms: (x''^1 , x''^2),

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `trans_closure_char.1.1.1`.

trans_closure_char.1.1.2:

{1}	$\text{transitive?}(\lambda (x, y: T):$ $\quad \exists (l: \text{list}[T]):$ $\quad \text{length}(l) \geq 2 \wedge$ $\quad \text{path_start}(l) = x \wedge$ $\quad \text{path_end}(l) = y \wedge$ $\quad (\forall (i: \text{below}(\text{length}(l) - 1)): \\ \quad \quad R'(\text{nth}(l, i), \text{nth}(l, 1 + i))))$
{2}	$\exists (l: \text{list}[T]):$ $\text{length}(l) \geq 2 \wedge$ $\text{path_start}(l) = x' \wedge$ $\text{path_end}(l) = y' \wedge$ $(\forall (i: \text{below}(\text{length}(l) - 1)): R'(\text{nth}(l, i), \text{nth}(l, 1 + i)))$

Hiding formulas: 2,

Expanding the definition of transitive?,

Repeatedly Skolemizing and flattening,

Installing automatic rewrites from: length_append nth_append_right nth_append_left

Simplifying, rewriting, and recording with decision procedures,

Instantiating the top quantifier in 1 with the terms: append(l' , cdr(l'')),

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 3 subgoals:

trans_closure_char.1.1.2.1:

{-1}	$\text{length}(l') \geq 2$
{-2}	$\text{path_start}(l') = x''$
{-3}	$\text{path_end}(l') = y''$
{-4}	$\forall (i: \text{below}(\text{length}(l') - 1)): R'(\text{nth}(l', i), \text{nth}(l', 1 + i))$
{-5}	$\text{length}(l'') \geq 2$
{-6}	$\text{path_start}(l'') = y''$
{-7}	$\text{path_end}(l'') = z'$
{-8}	$\forall (i: \text{below}(\text{length}(l'') - 1)): R'(\text{nth}(l'', i), \text{nth}(l'', 1 + i))$
{1}	$\forall (i: \text{below}(\text{length}(\text{append}(l', \text{cdr}(l''))) - 1)): \\ R'(\text{nth}(\text{append}(l', \text{cdr}(l'')), i), \text{nth}(\text{append}(l', \text{cdr}(l'')), 1 + i))$

Repeatedly Skolemizing and flattening,

Simplifying, rewriting, and recording with decision procedures,

Case splitting on $i!1 < \text{length}(l!1) - 1$,

we get 2 subgoals:

trans_closure_char.1.1.2.1.1:

{-1}	$i' < \text{length}(l') - 1$
{-2}	$i' < \text{length}(l') + \text{length}(l'') - 2$
{-3}	$\text{length}(l') \geq 2$
{-4}	$\text{path_start}(l') = x''$
{-5}	$\text{path_end}(l') = y''$
{-6}	$\forall (i: \text{below}(\text{length}(l') - 1)): R'(\text{nth}(l', i), \text{nth}(l', 1 + i))$
{-7}	$\text{length}(l'') \geq 2$
{-8}	$\text{path_start}(l'') = y''$
{-9}	$\text{path_end}(l'') = z'$
{-10}	$\forall (i: \text{below}(\text{length}(l'') - 1)): R'(\text{nth}(l'', i), \text{nth}(l'', 1 + i))$
{1}	$R'(\text{nth}(\text{append}(l', \text{cdr}(l'')), i'), \text{nth}(\text{append}(l', \text{cdr}(l'')), 1 + i'))$

Simplifying, rewriting, and recording with decision procedures,

C Proof scripts

Instantiating quantified variables,

This completes the proof of `trans_closure_char.1.1.2.1.1`.

`trans_closure_char.1.1.2.1.2`:

{-1}	$i' < \text{length}(l') + \text{length}(l'') - 2$
{-2}	$\text{length}(l') \geq 2$
{-3}	$\text{path_start}(l') = x''$
{-4}	$\text{path_end}(l') = y''$
{-5}	$\forall (i: \text{below}(\text{length}(l') - 1)): R'(\text{nth}(l', i), \text{nth}(l', 1 + i))$
{-6}	$\text{length}(l'') \geq 2$
{-7}	$\text{path_start}(l'') = y''$
{-8}	$\text{path_end}(l'') = z'$
{-9}	$\forall (i: \text{below}(\text{length}(l'') - 1)): R'(\text{nth}(l'', i), \text{nth}(l'', 1 + i))$
{1}	$i' < \text{length}(l') - 1$
{2}	$R'(\text{nth}(\text{append}(l', \text{cdr}(l'')), i'), \text{nth}(\text{append}(l', \text{cdr}(l'')), 1 + i'))$

Case splitting on $i!1 = \text{length}(l!1) - 1$,

we get 2 subgoals:

`trans_closure_char.1.1.2.1.2.1`:

{-1}	$i' = \text{length}(l') - 1$
{-2}	$i' < \text{length}(l') + \text{length}(l'') - 2$
{-3}	$\text{length}(l') \geq 2$
{-4}	$\text{path_start}(l') = x''$
{-5}	$\text{path_end}(l') = y''$
{-6}	$\forall (i: \text{below}(\text{length}(l') - 1)): R'(\text{nth}(l', i), \text{nth}(l', 1 + i))$
{-7}	$\text{length}(l'') \geq 2$
{-8}	$\text{path_start}(l'') = y''$
{-9}	$\text{path_end}(l'') = z'$
{-10}	$\forall (i: \text{below}(\text{length}(l'') - 1)): R'(\text{nth}(l'', i), \text{nth}(l'', 1 + i))$
{1}	$i' < \text{length}(l') - 1$
{2}	$R'(\text{nth}(\text{append}(l', \text{cdr}(l'')), i'), \text{nth}(\text{append}(l', \text{cdr}(l'')), 1 + i'))$

Simplifying, rewriting, and recording with decision procedures,

Expanding the definition of `path_end`,

Replacing using formula -1,

Replacing using formula -5,

Instantiating the top quantifier in -10 with the terms: 0,

Expanding the definition of `nth`,

Expanding the definition of `path_start`,

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `trans_closure_char.1.1.2.1.2.1`.

trans_closure_char.1.1.2.1.2.2:

{-1}	$i' < \text{length}(l') + \text{length}(l'') - 2$
{-2}	$\text{length}(l') \geq 2$
{-3}	$\text{path_start}(l') = x''$
{-4}	$\text{path_end}(l') = y''$
{-5}	$\forall (i: \text{below}(\text{length}(l') - 1)): R'(\text{nth}(l', i), \text{nth}(l', 1 + i))$
{-6}	$\text{length}(l'') \geq 2$
{-7}	$\text{path_start}(l'') = y''$
{-8}	$\text{path_end}(l'') = z'$
{-9}	$\forall (i: \text{below}(\text{length}(l'') - 1)): R'(\text{nth}(l'', i), \text{nth}(l'', 1 + i))$
{1}	$i' = \text{length}(l') - 1$
{2}	$i' < \text{length}(l') - 1$
{3}	$R'(\text{nth}(\text{append}(l', \text{cdr}(l'')), i'), \text{nth}(\text{append}(l', \text{cdr}(l'')), 1 + i'))$

Simplifying, rewriting, and recording with decision procedures,

Instantiating the top quantifier in -9 with the terms: $i' - \text{length}(l') + 1$,

Expanding the definition of nth,

which is trivially true.

This completes the proof of trans_closure_char.1.1.2.1.2.2.

trans_closure_char.1.1.2.2:

{-1}	$\text{length}(l') \geq 2$
{-2}	$\text{path_start}(l') = x''$
{-3}	$\text{path_end}(l') = y''$
{-4}	$\forall (i: \text{below}(\text{length}(l') - 1)): R'(\text{nth}(l', i), \text{nth}(l', 1 + i))$
{-5}	$\text{length}(l'') \geq 2$
{-6}	$\text{path_start}(l'') = y''$
{-7}	$\text{path_end}(l'') = z'$
{-8}	$\forall (i: \text{below}(\text{length}(l'') - 1)): R'(\text{nth}(l'', i), \text{nth}(l'', 1 + i))$
{1}	$\text{path_end}(\text{append}(l', \text{cdr}(l''))) = z'$

Expanding the definition of path_end,

Simplifying, rewriting, and recording with decision procedures,

Expanding the definition of nth,

which is trivially true.

This completes the proof of trans_closure_char.1.1.2.2.

trans_closure_char.1.1.2.3:

{-1}	$\text{length}(l') \geq 2$
{-2}	$\text{path_start}(l') = x''$
{-3}	$\text{path_end}(l') = y''$
{-4}	$\forall (i: \text{below}(\text{length}(l') - 1)): R'(\text{nth}(l', i), \text{nth}(l', 1 + i))$
{-5}	$\text{length}(l'') \geq 2$
{-6}	$\text{path_start}(l'') = y''$
{-7}	$\text{path_end}(l'') = z'$
{-8}	$\forall (i: \text{below}(\text{length}(l'') - 1)): R'(\text{nth}(l'', i), \text{nth}(l'', 1 + i))$
{1}	$\text{path_start}(\text{append}(l', \text{cdr}(l''))) = x''$

Expanding the definition of path_start,

Expanding the definition of append,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of trans_closure_char.1.1.2.3.

C Proof scripts

`trans_closure_char.1.2:`

$\begin{array}{l} \{1\} \quad \forall (x, y: T, l: \text{list}[T]): \\ \quad \text{path_start}(l) = x \wedge \text{length}(l) \geq 2 \supset \text{length}(l) \geq 1 \\ \{2\} \quad \exists (l: \text{list}[T]): \\ \quad \text{length}(l) \geq 2 \wedge \\ \quad \text{path_start}(l) = x' \wedge \\ \quad \text{path_end}(l) = y' \wedge \\ \quad (\forall (i: \text{below}(\text{length}(l) - 1)): R'(\text{nth}(l, i), \text{nth}(l, 1 + i))) \end{array}$

Hiding formulas: 2,

Repeatedly simplifying with decision procedures, rewriting, propositional reasoning, quantifier instantiation, skolemization, if-lifting and equality replacement,

This completes the proof of `trans_closure_char.1.2`.

`trans_closure_char.1.3:`

$\begin{array}{l} \{1\} \quad \forall (x, y: T, l: \text{list}[T]): \text{length}(l) \geq 2 \supset \text{length}(l) \geq 1 \\ \{2\} \quad \exists (l: \text{list}[T]): \\ \quad \text{length}(l) \geq 2 \wedge \\ \quad \text{path_start}(l) = x' \wedge \\ \quad \text{path_end}(l) = y' \wedge \\ \quad (\forall (i: \text{below}(\text{length}(l) - 1)): R'(\text{nth}(l, i), \text{nth}(l, 1 + i))) \end{array}$

Hiding formulas: 2,

Repeatedly simplifying with decision procedures, rewriting, propositional reasoning, quantifier instantiation, skolemization, if-lifting and equality replacement,

This completes the proof of `trans_closure_char.1.3`.

`trans_closure_char.2:`

$\begin{array}{l} \{-1\} \quad \exists (l: \text{list}[T]): \\ \quad \text{length}(l) \geq 2 \wedge \\ \quad \text{path_start}(l) = x' \wedge \\ \quad \text{path_end}(l) = y' \wedge \\ \quad (\forall (i: \text{below}(\text{length}(l) - 1)): R'(\text{nth}(l, i), \text{nth}(l, 1 + i))) \end{array}$
<hr/> $\{1\} \quad \text{transitive_closure}(R')(x', y')$

Case splitting on `FORALL (x, y: T, l: (path_list?[T]): length(l) >= 2 AND path_start(l) = x AND path_end(l) = y AND (FORALL (i: below(length(l) - 1)): R!1(nth(l, i), nth(l, 1 + i))) IMPLIES transitive_closure(R!1)(x, y)`,

we get 2 subgoals:

trans_closure_char.2.1:

$\{-1\} \quad \forall (x, y: T, l: (\text{path_list?}[T])):$ $\quad \text{length}(l) \geq 2 \wedge$ $\quad \text{path_start}(l) = x \wedge$ $\quad \text{path_end}(l) = y \wedge$ $\quad (\forall (i: \text{below}(\text{length}(l) - 1)):$ $\quad \quad R'(\text{nth}(l, i), \text{nth}(l, 1 + i)))$ $\quad \supset \text{transitive_closure}(R')(x, y)$	$\{-2\} \quad \exists (l: \text{list}[T]):$ $\quad \text{length}(l) \geq 2 \wedge$ $\quad \text{path_start}(l) = x' \wedge$ $\quad \text{path_end}(l) = y' \wedge$ $\quad (\forall (i: \text{below}(\text{length}(l) - 1))): R'(\text{nth}(l, i), \text{nth}(l, 1 + i)))$
$\{1\} \quad \text{transitive_closure}(R')(x', y')$	

Repeatedly Skolemizing and flattening,
Instantiating quantified variables,
Instantiating quantified variables,
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of trans_closure_char.2.1.

trans_closure_char.2.2:

$\{-1\} \quad \exists (l: \text{list}[T]):$ $\quad \text{length}(l) \geq 2 \wedge$ $\quad \text{path_start}(l) = x' \wedge$ $\quad \text{path_end}(l) = y' \wedge$ $\quad (\forall (i: \text{below}(\text{length}(l) - 1))): R'(\text{nth}(l, i), \text{nth}(l, 1 + i)))$	
$\{1\} \quad \forall (x, y: T, l: (\text{path_list?}[T])):$ $\quad \text{length}(l) \geq 2 \wedge$ $\quad \text{path_start}(l) = x \wedge$ $\quad \text{path_end}(l) = y \wedge$ $\quad (\forall (i: \text{below}(\text{length}(l) - 1)):$ $\quad \quad R'(\text{nth}(l, i), \text{nth}(l, 1 + i)))$ $\quad \supset \text{transitive_closure}(R')(x, y)$	$\{2\} \quad \text{transitive_closure}(R')(x', y')$

Hiding formulas: -1, 2,
Inducting on l on formula 1 using induction scheme path_list_induction[T],
we get 2 subgoals:

trans_closure_char.2.2.1:

$\{1\} \quad \forall (t: T):$ $\quad \forall (x, y: T):$ $\quad (\text{length}(\text{cons}(t, \text{null})) \geq 2 \wedge$ $\quad \text{path_start}(\text{cons}(t, \text{null})) = x \wedge$ $\quad \text{path_end}(\text{cons}(t, \text{null})) = y \wedge$ $\quad (\forall (i: \text{below}(\text{length}(\text{cons}(t, \text{null})) - 1)):$ $\quad \quad R'(\text{nth}(\text{cons}(t, \text{null}), i), \text{nth}(\text{cons}(t, \text{null}), 1 + i))))$ $\quad \supset \text{transitive_closure}(R')(x, y)$	
--	--

Repeatedly Skolemizing and flattening,
Installing automatic rewrites from: nth path_start path_end
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of trans_closure_char.2.2.1.

trans_closure_char.2.2.2:

<pre> {1} ∀ (t: T, tail: (path_list? [T])): (∀ (x, y: T): (length(tail) ≥ 2 ∧ path_start(tail) = x ∧ path_end(tail) = y ∧ (∀ (i: below(length(tail) - 1)): R'(nth(tail, i), nth(tail, 1 + i)))) ⊃ transitive_closure(R')(x, y)) ⊃ (∀ (x, y: T): (length(cons(t, tail)) ≥ 2 ∧ path_start(cons(t, tail)) = x ∧ path_end(cons(t, tail)) = y ∧ (∀ (i: below(length(cons(t, tail)) - 1)): R'(nth(cons(t, tail), i), nth(cons(t, tail), 1 + i)))) ⊃ transitive_closure(R')(x, y)) </pre>

Repeatedly Skolemizing and flattening,
Installing automatic rewrites from: path_start path_end
Simplifying, rewriting, and recording with decision procedures,
Case splitting on length(tail!1) = 1,
we get 2 subgoals:

trans_closure_char.2.2.2.1:

<pre> {-1} length(tail') = 1 {-2} length(tail') ≥ 1 {-3} ∀ (x, y: T): (length(tail') ≥ 2 ∧ car(tail') = x ∧ nth(tail', length(tail') - 1) = y ∧ (∀ (i: below(length(tail') - 1)): R'(nth(tail', i), nth(tail', 1 + i)))) ⊃ transitive_closure(R')(x, y) {-4} 1 + length(tail') ≥ 2 {-5} t' = x'' {-6} nth(cons(t', tail'), length(tail')) = y'' {-7} ∀ (i: below(length(cons(t', tail')) - 1)): R'(nth(cons(t', tail'), i), nth(cons(t', tail'), 1 + i)) {1} transitive_closure(R')(x'', y'') </pre>

Hiding formulas: -3,
Installing automatic rewrites from: nth
Instantiating the top quantifier in -6 with the terms: 0,
Simplifying, rewriting, and recording with decision procedures,
Expanding the definition of transitive_closure,
Repeatedly Skolemizing and flattening,
Expanding the definition(s) of (subset? member),
Instantiating quantified variables,
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of trans_closure_char.2.2.2.1.

`trans_closure_char.2.2.2.2:`

$\{-1\}$ $\text{length}(\text{tail}') \geq 1$ $\{-2\}$ $\forall (x, y: T):$ $\quad (\text{length}(\text{tail}') \geq 2 \wedge$ $\quad \quad \text{car}(\text{tail}') = x \wedge$ $\quad \quad \text{nth}(\text{tail}', \text{length}(\text{tail}') - 1) = y \wedge$ $\quad \quad (\forall (i: \text{below}(\text{length}(\text{tail}') - 1)):$ $\quad \quad \quad R'(\text{nth}(\text{tail}', i), \text{nth}(\text{tail}', 1 + i)))$ $\quad \quad \supset \text{transitive_closure}(R')(x, y)$	$\{-3\}$ $1 + \text{length}(\text{tail}') \geq 2$ $\{-4\}$ $t' = x''$ $\{-5\}$ $\text{nth}(\text{cons}(t', \text{tail}'), \text{length}(\text{tail}')) = y''$ $\{-6\}$ $\forall (i: \text{below}(\text{length}(\text{cons}(t', \text{tail}')) - 1)):$ $\quad R'(\text{nth}(\text{cons}(t', \text{tail}'), i), \text{nth}(\text{cons}(t', \text{tail}'), 1 + i))$
$\{1\}$ $\text{length}(\text{tail}') = 1$ $\{2\}$ $\text{transitive_closure}(R')(x'', y'')$	

Expanding the definition of `nth`,

Simplifying, rewriting, and recording with decision procedures,

Instantiating the top quantifier in -2 with the terms: `car(tail')`, `y''`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 2 subgoals:

`trans_closure_char.2.2.2.2.1:`

$\{-1\}$ $\text{length}(\text{tail}') \geq 1$ $\{-2\}$ $\text{nth}(\text{tail}', \text{length}(\text{tail}') - 1) = y''$ $\{-3\}$ $\text{transitive_closure}(R')(\text{car}(\text{tail}'), y'')$ $\{-4\}$ $1 + \text{length}(\text{tail}') \geq 2$ $\{-5\}$ $t' = x''$ $\{-6\}$ $\forall (i: \text{below}(\text{length}(\text{cons}(t', \text{tail}')) - 1)):$ $\quad R'(\text{nth}(\text{cons}(t', \text{tail}'), i), \text{nth}(\text{cons}(t', \text{tail}'), 1 + i))$	
$\{1\}$ $\text{length}(\text{tail}') = 1$ $\{2\}$ $\text{transitive_closure}(R')(x'', y'')$	

Instantiating the top quantifier in -6 with the terms: 0,

Installing automatic rewrites from: `nth`

Simplifying, rewriting, and recording with decision procedures,

Replacing using formula -5,

Hiding formulas: -1, -2, -4, -5,

Expanding the definition of `transitive_closure`,

Repeatedly Skolemizing and flattening,

Instantiating quantified variables,

Simplifying, rewriting, and recording with decision procedures,

Expanding the definition of `transitive?`,

Instantiating the top quantifier in -4 with the terms: `x''`, `car(tail')`, `y''`,

Simplifying, rewriting, and recording with decision procedures,

Expanding the definition(s) of (`subset?` `member`),

Instantiating the top quantifier in -3 with the terms: (`x''`, `car(tail')`),

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `trans_closure_char.2.2.2.2.1`.

trans_closure_char.2.2.2.2.2:

{-1} length(tail') ≥ 1 {-2} nth(tail', length(tail') - 1) = y'' {-3} 1 + length(tail') ≥ 2 {-4} t' = x'' {-5} ∀ (i: below(length(cons(t', tail')) - 1)): R'(nth(cons(t', tail'), i), nth(cons(t', tail'), 1 + i))	{1} length(tail') = 1 {2} ∀ (i: below(length(tail') - 1)): R'(nth(tail', i), nth(tail', 1 + i)) {3} transitive_closure(R')(x'', y'')
--	---

Keeping (-5 2) and hiding *,
 Repeatedly Skolemizing and flattening,
 Instantiating the top quantifier in -2 with the terms: i' + 1,
 Installing automatic rewrites from: nth
 Simplifying, rewriting, and recording with decision procedures,
 This completes the proof of trans_closure_char.2.2.2.2.2.
 Q.E.D.

C.99 Proofs for Graph_Util (graph.pvs)

C.99.1 Graph_Util.every_nth

Terse proof for every_nth.

every_nth:

{1} ∀ (P: PRED[T], l: list[T]): every(P)(l) ≡ (∀ (i: below(length(l))): P(nth(l, i)))
--

Inducting on l on formula 1,
 we get 2 subgoals:

every_nth.1:

{1} ∀ (P: PRED[T]): every(P)(null) ≡ (∀ (i: below(length(null))): P(nth(null, i)))

Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of every_nth.1.

every_nth.2:

{1} ∀ (cons1_var: T, cons2_var: list[T]): (∀ (P: PRED[T]): every(P)(cons2_var) ≡ (∀ (i: below(length(cons2_var))): P(nth(cons2_var, i)))) ⊃ (∀ (P: PRED[T]): every(P)(cons(cons1_var, cons2_var)) ≡ (∀ (i: below(length(cons(cons1_var, cons2_var)))): P(nth(cons(cons1_var, cons2_var), i)))

Repeatedly Skolemizing and flattening,
 Applying propositional simplification,
 we get 2 subgoals:

every_nth.2.1:

$$\begin{array}{|l}
 \{-1\} \text{ every}(P')(\text{cons}(\text{cons1_var}', \text{cons2_var}')) \\
 \{-2\} \forall (P: \text{PRED}[T]): \\
 \quad \text{every}(P)(\text{cons2_var}') \equiv \\
 \quad (\forall (i: \text{below}(\text{length}(\text{cons2_var}'))): P(\text{nth}(\text{cons2_var}', i))) \\
 \hline
 \{1\} \forall (i: \text{below}(\text{length}(\text{cons}(\text{cons1_var}', \text{cons2_var}')))): \\
 \quad P'(\text{nth}(\text{cons}(\text{cons1_var}', \text{cons2_var}'), i))
 \end{array}$$

Expanding the definition of every,
 Applying disjunctive simplification to flatten sequent,
 Instantiating quantified variables,
 Simplifying, rewriting, and recording with decision procedures,
 Repeatedly Skolemizing and flattening,
 Expanding the definition of length,
 Expanding the definition of nth,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 Instantiating the top quantifier in -4 with the terms: $i' - 1$,
 This completes the proof of **every_nth.2.1**.

every_nth.2.2:

$$\begin{array}{|l}
 \{-1\} \forall (i: \text{below}(\text{length}(\text{cons}(\text{cons1_var}', \text{cons2_var}')))): \\
 \quad P'(\text{nth}(\text{cons}(\text{cons1_var}', \text{cons2_var}'), i)) \\
 \{-2\} \forall (P: \text{PRED}[T]): \\
 \quad \text{every}(P)(\text{cons2_var}') \equiv \\
 \quad (\forall (i: \text{below}(\text{length}(\text{cons2_var}'))): P(\text{nth}(\text{cons2_var}', i))) \\
 \hline
 \{1\} \text{ every}(P')(\text{cons}(\text{cons1_var}', \text{cons2_var}'))
 \end{array}$$

Expanding the definition of every,
 Applying propositional simplification,
 we get 2 subgoals:

every_nth.2.2.1:

$$\begin{array}{|l}
 \{-1\} \forall (i: \text{below}(\text{length}(\text{cons}(\text{cons1_var}', \text{cons2_var}')))): \\
 \quad P'(\text{nth}(\text{cons}(\text{cons1_var}', \text{cons2_var}'), i)) \\
 \{-2\} \forall (P: \text{PRED}[T]): \\
 \quad \text{every}(P)(\text{cons2_var}') \equiv \\
 \quad (\forall (i: \text{below}(\text{length}(\text{cons2_var}'))): P(\text{nth}(\text{cons2_var}', i))) \\
 \hline
 \{1\} P'(\text{cons1_var}')
 \end{array}$$

Instantiating the top quantifier in -1 with the terms: 0,
 we get 2 subgoals:

every_nth.2.2.1.1:

$$\begin{array}{|l}
 \{-1\} P'(\text{nth}(\text{cons}(\text{cons1_var}', \text{cons2_var}'), 0)) \\
 \{-2\} \forall (P: \text{PRED}[T]): \\
 \quad \text{every}(P)(\text{cons2_var}') \equiv \\
 \quad (\forall (i: \text{below}(\text{length}(\text{cons2_var}'))): P(\text{nth}(\text{cons2_var}', i))) \\
 \hline
 \{1\} P'(\text{cons1_var}')
 \end{array}$$

Expanding the definition of nth,
 which is trivially true.

This completes the proof of **every_nth.2.2.1.1**.

every_nth.2.2.1.2:

$$\frac{\begin{array}{l} \{-1\} \quad \forall (P: \text{PRED}[T]): \\ \quad \text{every}(P)(\text{cons2_var}') \equiv \\ \quad (\forall (i: \text{below}(\text{length}(\text{cons2_var}'))): P(\text{nth}(\text{cons2_var}', i))) \end{array}}{\begin{array}{l} \{1\} \quad 0 < \text{length}[T](\text{cons}[T](\text{cons1_var}', \text{cons2_var}')) \\ \{2\} \quad P'(\text{cons1_var}') \end{array}}$$

Expanding the definition of length,
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of every_nth.2.2.1.2.

every_nth.2.2.2:

$$\frac{\begin{array}{l} \{-1\} \quad \forall (i: \text{below}(\text{length}(\text{cons}(\text{cons1_var}', \text{cons2_var}')))): \\ \quad P'(\text{nth}(\text{cons}(\text{cons1_var}', \text{cons2_var}'), i)) \\ \{-2\} \quad \forall (P: \text{PRED}[T]): \\ \quad \text{every}(P)(\text{cons2_var}') \equiv \\ \quad (\forall (i: \text{below}(\text{length}(\text{cons2_var}'))): P(\text{nth}(\text{cons2_var}', i))) \end{array}}{\{1\} \quad \text{every}(P')(\text{cons2_var}')}$$

Instantiating quantified variables,
Simplifying, rewriting, and recording with decision procedures,
Hiding formulas: 1,
Repeatedly Skolemizing and flattening,
Instantiating the top quantifier in -2 with the terms: $i' + 1$,
we get 2 subgoals:

every_nth.2.2.2.1:

$$\frac{\begin{array}{l} \{-1\} \quad i' < \text{length}(\text{cons2_var}') \\ \{-2\} \quad P'(\text{nth}(\text{cons}(\text{cons1_var}', \text{cons2_var}'), i' + 1)) \end{array}}{\{1\} \quad P'(\text{nth}(\text{cons2_var}', i'))}$$

Expanding the definition of nth,
which is trivially true.
This completes the proof of every_nth.2.2.2.1.

every_nth.2.2.2.2:

$$\frac{\{-1\} \quad i' < \text{length}(\text{cons2_var}')}{\begin{array}{l} \{1\} \quad 1 + i' < \text{length}[T](\text{cons}[T](\text{cons1_var}', \text{cons2_var}')) \\ \{2\} \quad P'(\text{nth}(\text{cons2_var}', i')) \end{array}}$$

Expanding the definition of length,
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of every_nth.2.2.2.2.
Q.E.D.

C.99.2 Graph_Util.nth_append_left_TCC1

Terse proof for nth_append_left_TCC1.

nth_append_left_TCC1:

$$\frac{}{\{1\} \quad \forall (l_1, l_2: \text{list}[T], n: \text{nat}): \\ \quad n < \text{length}(l_1) \supset n < \text{length}[T](\text{append}[T](l_1, l_2))}$$

Repeatedly Skolemizing and flattening,
Rewriting using length_append, matching in *,

Simplifying, rewriting, and recording with decision procedures,
 This completes the proof of `nth_append_left_TCC1`.
 Q.E.D.

C.99.3 Graph_Util.nth_append_left

Terse proof for `nth_append_left`.

`nth_append_left`:

$$\frac{}{\{1\} \quad \forall (l_1, l_2: \text{list}[T], n: \text{nat}): \\ n < \text{length}(l_1) \supset \text{nth}(\text{append}(l_1, l_2), n) = \text{nth}(l_1, n)}$$

Inducting on l_1 on formula 1,
 we get 3 subgoals:

`nth_append_left.1`:

$$\frac{}{\{1\} \quad \forall (l_2: \text{list}[T], n: \text{nat}): \\ n < \text{length}(\text{null}) \supset \text{nth}(\text{append}(\text{null}, l_2), n) = \text{nth}(\text{null}, n)}$$

Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of `nth_append_left.1`.

`nth_append_left.2`:

$$\frac{}{\{1\} \quad \forall (\text{cons1_var}: T, \text{cons2_var}: \text{list}[T]): \\ (\forall (l_2: \text{list}[T], n: \text{nat}): \\ n < \text{length}(\text{cons2_var}) \supset \text{nth}(\text{append}(\text{cons2_var}, l_2), n) = \text{nth}(\text{cons2_var}, n)) \\ \supset \\ (\forall (l_2: \text{list}[T], n: \text{nat}): \\ n < \text{length}(\text{cons}(\text{cons1_var}, \text{cons2_var})) \supset \\ \text{nth}(\text{append}(\text{cons}(\text{cons1_var}, \text{cons2_var}), l_2), n) = \\ \text{nth}(\text{cons}(\text{cons1_var}, \text{cons2_var}), n))}$$

Repeatedly Skolemizing and flattening,
 Expanding the definition of `append`,
 Expanding the definition of `nth`,
 Expanding the definition of `length`,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 Rewriting using `-2`, matching in `*`,
 This completes the proof of `nth_append_left.2`.

`nth_append_left.3`:

$$\frac{}{\{1\} \quad \forall (l_1, l_2: \text{list}[T], n: \text{nat}): \\ n < \text{length}(l_1) \supset n < \text{length}[T](\text{append}[T](l_1, l_2)) \\ \{2\} \quad \forall (l_2: \text{list}[T], n: \text{nat}): \\ n < \text{length}(l'_1) \supset \text{nth}(\text{append}(l'_1, l_2), n) = \text{nth}(l'_1, n)}$$

Hiding formulas: 2,
 Repeatedly Skolemizing and flattening,
 Rewriting using `length_append`, matching in `*`,
 Simplifying, rewriting, and recording with decision procedures,
 This completes the proof of `nth_append_left.3`.
 Q.E.D.

C.99.4 Graph_Util.nth_append_right_TCC1

Terse proof for nth_append_right_TCC1.

nth_append_right_TCC1:

$$\{1\} \quad \forall (l_1, l_2: \text{list}[T], n: \text{nat}):$$

$$n \geq \text{length}(l_1) \wedge n < \text{length}(\text{append}(l_1, l_2)) \supset$$

$$n - \text{length}[T](l_1) \geq 0 \wedge n - \text{length}[T](l_1) < \text{length}[T](l_2)$$

Repeatedly Skolemizing and flattening,
 Simplifying, rewriting, and recording with decision procedures,
 Rewriting using length_append, matching in *,
 Simplifying, rewriting, and recording with decision procedures,
 This completes the proof of nth_append_right_TCC1.
 Q.E.D.

C.99.5 Graph_Util.nth_append_right

Terse proof for nth_append_right.

nth_append_right:

$$\{1\} \quad \forall (l_1, l_2: \text{list}[T], n: \text{nat}):$$

$$n \geq \text{length}(l_1) \wedge n < \text{length}(\text{append}(l_1, l_2)) \supset$$

$$\text{nth}(\text{append}(l_1, l_2), n) = \text{nth}(l_2, n - \text{length}(l_1))$$

Inducting on l_1 on formula 1,
 we get 3 subgoals:

nth_append_right.1:

$$\{1\} \quad \forall (l_2: \text{list}[T], n: \text{nat}):$$

$$n \geq \text{length}(\text{null}) \wedge n < \text{length}(\text{append}(\text{null}, l_2)) \supset$$

$$\text{nth}(\text{append}(\text{null}, l_2), n) = \text{nth}(l_2, n - \text{length}(\text{null}))$$

Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of nth_append_right.1.

nth_append_right.2:

$$\{1\} \quad \forall (\text{cons1_var}: T, \text{cons2_var}: \text{list}[T]):$$

$$(\forall (l_2: \text{list}[T], n: \text{nat}):$$

$$n \geq \text{length}(\text{cons2_var}) \wedge n < \text{length}(\text{append}(\text{cons2_var}, l_2)) \supset$$

$$\text{nth}(\text{append}(\text{cons2_var}, l_2), n) = \text{nth}(l_2, n - \text{length}(\text{cons2_var})))$$

$$\supset$$

$$(\forall (l_2: \text{list}[T], n: \text{nat}):$$

$$n \geq \text{length}(\text{cons}(\text{cons1_var}, \text{cons2_var})) \wedge$$

$$n < \text{length}(\text{append}(\text{cons}(\text{cons1_var}, \text{cons2_var}), l_2))$$

$$\supset$$

$$\text{nth}(\text{append}(\text{cons}(\text{cons1_var}, \text{cons2_var}), l_2), n) =$$

$$\text{nth}(l_2, n - \text{length}(\text{cons}(\text{cons1_var}, \text{cons2_var}))))$$

Repeatedly Skolemizing and flattening,
 Rewriting using length_append, matching in *,
 Expanding the definition of length,

Expanding the definition of length,
 Expanding the definition of append,
 Expanding the definition of nth,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 Rewriting using -, matching in *,
 Rewriting using length_append, matching in *,
 Simplifying, rewriting, and recording with decision procedures,
 This completes the proof of nth_append_right.2.
 nth_append_right.3:

$\{1\} \quad \forall (l_1, l_2: \text{list}[T], n: \text{nat}):$ $n \geq \text{length}(l_1) \wedge n < \text{length}(\text{append}(l_1, l_2)) \supset$ $n - \text{length}[T](l_1) \geq 0 \wedge n - \text{length}[T](l_1) < \text{length}[T](l_2)$ $\{2\} \quad \forall (l_2: \text{list}[T], n: \text{nat}):$ $n \geq \text{length}(l'_1) \wedge n < \text{length}(\text{append}(l'_1, l_2)) \supset$ $\text{nth}(\text{append}(l'_1, l_2), n) = \text{nth}(l_2, n - \text{length}(l'_1))$

Hiding formulas: 2,
 Repeatedly Skolemizing and flattening,
 Rewriting using length_append, matching in *,
 Simplifying, rewriting, and recording with decision procedures,
 This completes the proof of nth_append_right.3.
 Q.E.D.

C.99.6 Graph_Util.list_pred_card

Terse proof for list_pred_card.
 list_pred_card:

$\{1\} \quad \forall (l: \text{list}[T], P: \text{PRED}[T]):$ $\text{is_finite}(P) \wedge \text{every}(P)(l) \wedge \text{length}(l) > \text{card}(P) \supset$ $(\exists (n_1, n_2: \text{below}(\text{length}(l))): \neg n_1 = n_2 \wedge \text{nth}(l, n_1) = \text{nth}(l, n_2))$

Repeatedly Skolemizing and flattening,
 Letting f name LAMBDA (i: below(length(!1))): nth(!1, i),
 Case splitting on injective?(f),
 we get 2 subgoals:
 list_pred_card.1:

$\{-1\} \quad \text{injective?}(f)$ $\{-2\} \quad (\lambda (i: \text{below}(\text{length}(l'))): \text{nth}(l', i)) = f$ $\{-3\} \quad \text{is_finite}(P')$ $\{-4\} \quad \text{every}(P')(l')$ $\{-5\} \quad \text{length}(l') > \text{card}(P')$ <hr/> $\{1\} \quad \exists (n_1, n_2: \text{below}(\text{length}(l'))): \neg n_1 = n_2 \wedge \text{nth}(l', n_1) = \text{nth}(l', n_2)$

Using lemma bijective_restrict_to_image,
 Simplifying, rewriting, and recording with decision procedures,
 Case splitting on (EXISTS (d: below(length[T](!1))): TRUE) OR (FORALL (r: (image[below(length(!1)), T] (f, fullset[below(length[T](!1)))])): FALSE),
 we get 2 subgoals:

C Proof scripts

list_pred_card.1.1.1:

{-1}	$(\exists (d: \text{below}(\text{length}[T](l'))): \text{TRUE}) \vee$ $(\forall (r:$ $\quad (\text{image}[\text{below}(\text{length}(l')), T](f, \text{fullset}[\text{below}(\text{length}[T](l'))]))):$ $\quad \text{FALSE})$
{-2}	$\text{bijective?}(\text{restrict_to_image}(f))$
{-3}	$\text{injective?}(f)$
{-4}	$(\lambda (i: \text{below}(\text{length}(l'))): \text{nth}(l', i)) = f$
{-5}	$\text{is_finite}(P')$
{-6}	$\text{every}(P')(l')$
{-7}	$\text{length}(l') > \text{card}(P')$
{1}	$\exists (n_1, n_2: \text{below}(\text{length}(l'))): \neg n_1 = n_2 \wedge \text{nth}(l', n_1) = \text{nth}(l', n_2)$

Using lemma `bijective_inverse_alt_is_bijective`,

Using lemma `card_bij`,

we get 2 subgoals:

list_pred_card.1.1.1.1:

{-1}	$\text{card}(\text{image}(f, \text{fullset}[\text{below}(\text{length}(l'))])) = \text{length}(l') \equiv$ $(\exists (f_1: [(\text{image}(f, \text{fullset}[\text{below}(\text{length}(l'))])) \rightarrow \text{below}[\text{length}(l')]]):$ $\quad \text{bijective?}(f_1))$
{-2}	$\text{bijective?}[(\text{image}(f, \text{fullset}[\text{below}(\text{length}(l'))])), \text{below}(\text{length}(l'))]$ $(\text{inverse_alt}(\text{restrict_to_image}(f)))$
{-3}	$(\exists (d: \text{below}(\text{length}[T](l'))): \text{TRUE}) \vee$ $(\forall (r:$ $\quad (\text{image}[\text{below}(\text{length}(l')), T](f, \text{fullset}[\text{below}(\text{length}[T](l'))]))):$ $\quad \text{FALSE})$
{-4}	$\text{bijective?}(\text{restrict_to_image}(f))$
{-5}	$\text{injective?}(f)$
{-6}	$(\lambda (i: \text{below}(\text{length}(l'))): \text{nth}(l', i)) = f$
{-7}	$\text{is_finite}(P')$
{-8}	$\text{every}(P')(l')$
{-9}	$\text{length}(l') > \text{card}(P')$
{1}	$\exists (n_1, n_2: \text{below}(\text{length}(l'))): \neg n_1 = n_2 \wedge \text{nth}(l', n_1) = \text{nth}(l', n_2)$

Applying disjunctive simplification to flatten sequent,

Splitting conjunctions,

we get 2 subgoals:

list_pred_card.1.1.1.1:

{-1}	$\text{card}(\text{image}(f, \text{fullset}[\text{below}(\text{length}(l'))])) = \text{length}(l')$
{-2}	$\text{card}(\text{image}(f, \text{fullset}[\text{below}(\text{length}(l'))])) = \text{length}(l') \supset$ $(\exists (f_1: [(\text{image}(f, \text{fullset}[\text{below}(\text{length}(l'))])) \rightarrow \text{below}[\text{length}(l')]]):$ $\text{bijective?}(f_1)$
{-3}	$\text{bijective?}[(\text{image}(f, \text{fullset}[\text{below}(\text{length}(l'))])), \text{below}(\text{length}(l'))]$ $(\text{inverse_alt}(\text{restrict_to_image}(f)))$
{-4}	$(\exists (d: \text{below}(\text{length}[T](l'))): \text{TRUE}) \vee$ $(\forall (r:$ $\text{FALSE})$ $(\text{image}[\text{below}(\text{length}(l')), T](f, \text{fullset}[\text{below}(\text{length}[T](l'))]))):$
{-5}	$\text{bijective?}(\text{restrict_to_image}(f))$
{-6}	$\text{injective?}(f)$
{-7}	$(\lambda (i: \text{below}(\text{length}(l'))): \text{nth}(l', i)) = f$
{-8}	$\text{is_finite}(P')$
{-9}	$\text{every}(P')(l')$
{-10}	$\text{length}(l') > \text{card}(P')$
{1}	$\exists (n_1, n_2: \text{below}(\text{length}(l'))): \neg n_1 = n_2 \wedge \text{nth}(l', n_1) = \text{nth}(l', n_2)$

Simplifying, rewriting, and recording with decision procedures,

Hiding formulas: -2,

Using lemma card_subset,

Simplifying, rewriting, and recording with decision procedures,

Hiding formulas: -1, -2, -3, -4, -5, -6, 2,

Expanding the definition(s) of (fullset subset? member image f),

Repeatedly Skolemizing and flattening,

Rewriting using every_nth, matching in *,

Instantiating quantified variables,

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of list_pred_card.1.1.1.1.

list_pred_card.1.1.1.2:

{-1}	$\text{card}(\text{image}(f, \text{fullset}[\text{below}(\text{length}(l'))])) = \text{length}(l') \supset$ $(\exists (f_1: [(\text{image}(f, \text{fullset}[\text{below}(\text{length}(l'))])) \rightarrow \text{below}[\text{length}(l')]]):$ $\text{bijective?}(f_1)$
{-2}	$\text{bijective?}[(\text{image}(f, \text{fullset}[\text{below}(\text{length}(l'))])), \text{below}(\text{length}(l'))]$ $(\text{inverse_alt}(\text{restrict_to_image}(f)))$
{-3}	$(\exists (d: \text{below}(\text{length}[T](l'))): \text{TRUE}) \vee$ $(\forall (r:$ $\text{FALSE})$ $(\text{image}[\text{below}(\text{length}(l')), T](f, \text{fullset}[\text{below}(\text{length}[T](l'))]))):$
{-4}	$\text{bijective?}(\text{restrict_to_image}(f))$
{-5}	$\text{injective?}(f)$
{-6}	$(\lambda (i: \text{below}(\text{length}(l'))): \text{nth}(l', i)) = f$
{-7}	$\text{is_finite}(P')$
{-8}	$\text{every}(P')(l')$
{-9}	$\text{length}(l') > \text{card}(P')$
{1}	$\exists (f_1: [(\text{image}(f, \text{fullset}[\text{below}(\text{length}(l'))])) \rightarrow \text{below}[\text{length}(l')]]):$ $\text{bijective?}(f_1)$
{2}	$\exists (n_1, n_2: \text{below}(\text{length}(l'))): \neg n_1 = n_2 \wedge \text{nth}(l', n_1) = \text{nth}(l', n_2)$

Hiding formulas: -1, 2,

Instantiating quantified variables,

This completes the proof of list_pred_card.1.1.1.2.

list_pred_card.1.1.2:

{-1}	bijective?(image(f , fullset[below(length(l'))]), below(length(l')) (inverse_alt(restrict_to_image(f))))
{-2}	$(\exists (d: \text{below}(\text{length}[T](l'))): \text{TRUE}) \vee$ $(\forall (r:$ $(\text{image}[\text{below}(\text{length}(l')), T](f, \text{fullset}[\text{below}(\text{length}[T](l'))]))):$ FALSE)
{-3}	bijective?(restrict_to_image(f))
{-4}	injective?(f)
{-5}	$(\lambda (i: \text{below}(\text{length}(l'))): \text{nth}(l', i)) = f$
{-6}	is_finite(P')
{-7}	every(P')(l')
{-8}	length(l') > card(P')
{1}	is_finite[T] (image[below(length(l')), T](f , fullset[below(length[T](l'))]))
{2}	$\exists (n_1, n_2: \text{below}(\text{length}(l'))): \neg n_1 = n_2 \wedge \text{nth}(l', n_1) = \text{nth}(l', n_2)$

Hiding formulas: -1, -2, 2,

Using lemma finite_image,

Keeping 1 and hiding *,

Expanding the definition of is_finite,

Instantiating the top quantifier in 1 with the terms: length(l'), id,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of list_pred_card.1.1.2.

list_pred_card.1.2:

{-1}	bijective?(restrict_to_image(f))
{-2}	injective?(f)
{-3}	$(\lambda (i: \text{below}(\text{length}(l'))): \text{nth}(l', i)) = f$
{-4}	is_finite(P')
{-5}	every(P')(l')
{-6}	length(l') > card(P')
{1}	$(\exists (d: \text{below}(\text{length}[T](l'))): \text{TRUE}) \vee$ $(\forall (r:$ $(\text{image}[\text{below}(\text{length}(l')), T](f, \text{fullset}[\text{below}(\text{length}[T](l'))]))):$ FALSE)
{2}	$\exists (n_1, n_2: \text{below}(\text{length}(l'))): \neg n_1 = n_2 \wedge \text{nth}(l', n_1) = \text{nth}(l', n_2)$

Applying disjunctive simplification to flatten sequent,

Hiding formulas: 3,

Repeatedly Skolemizing and flattening,

Expanding the definition of image,

Repeatedly Skolemizing and flattening,

Instantiating quantified variables,

This completes the proof of list_pred_card.1.2.

list_pred_card.2:

{-1}	$(\lambda (i: \text{below}(\text{length}(l'))): \text{nth}(l', i)) = f$
{-2}	is_finite(P')
{-3}	every(P')(l')
{-4}	length(l') > card(P')
{1}	injective?(f)
{2}	$\exists (n_1, n_2: \text{below}(\text{length}(l'))): \neg n_1 = n_2 \wedge \text{nth}(l', n_1) = \text{nth}(l', n_2)$

Expanding the definition of injective?,
 Repeatedly Skolemizing and flattening,
 Instantiating the top quantifier in 2 with the terms: x'_1, x'_2 ,
 Simplifying, rewriting, and recording with decision procedures,
 Replacing using formula -4,
 Simplifying, rewriting, and recording with decision procedures,
 This completes the proof of `list_pred_card.2`.
 Q.E.D.

C.99.7 Graph_Util.path_start_TCC1

Terse proof for `path_start_TCC1`.

`path_start_TCC1`:

$$\{1\} \quad \forall (l: (\text{path_list?})): \text{cons?}[T](l)$$

Repeatedly Skolemizing and flattening,
 Expanding the definition of `path_list?`,
 Expanding the definition of `length`,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 This completes the proof of `path_start_TCC1`.
 Q.E.D.

C.99.8 Graph_Util.path_end_TCC1

Terse proof for `path_end_TCC1`.

`path_end_TCC1`:

$$\{1\} \quad \forall (l: (\text{path_list?})): \\ \text{length}[T](l) - 1 \geq 0 \wedge \text{length}[T](l) - 1 < \text{length}[T](l)$$

Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of `path_end_TCC1`.
 Q.E.D.

C.99.9 Graph_Util.path_list_induction_TCC1

Terse proof for `path_list_induction_TCC1`.

`path_list_induction_TCC1`:

$$\{1\} \quad \forall (t: T): \text{path_list?}(\text{cons}[T](t, \text{null}[T]))$$

Repeatedly Skolemizing and flattening,
 Installing automatic rewrites from: `path_list? length`
 Simplifying, rewriting, and recording with decision procedures,
 This completes the proof of `path_list_induction_TCC1`.
 Q.E.D.

C.99.10 Graph_Util.path_list_induction_TCC2

Terse proof for path_list_induction_TCC2.

path_list_induction_TCC2:

$\{1\} \quad \forall (p: [(\text{path_list?}) \rightarrow \text{boolean}]):$ $\quad (\forall (t: T): p(\text{cons}(t, \text{null}))) \supset$ $\quad (\forall (t_1: T, \text{tail}: (\text{path_list?})): p(\text{tail}) \supset \text{path_list?}(\text{cons}[T](t_1, \text{tail})))$

Repeatedly Skolemizing and flattening,

Hiding formulas: -2, -3,

Installing automatic rewrites from: path_list? length

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of path_list_induction_TCC2.

Q.E.D.

C.99.11 Graph_Util.path_list_induction

Terse proof for path_list_induction.

path_list_induction:

$\{1\} \quad \forall (p: [(\text{path_list?}) \rightarrow \text{boolean}]):$ $\quad (\forall (t: T): p(\text{cons}(t, \text{null}))) \wedge$ $\quad (\forall (t: T, \text{tail}: (\text{path_list?})): p(\text{tail}) \supset p(\text{cons}(t, \text{tail})))$ $\quad \supset (\forall (l: (\text{path_list?})): p(l))$
--

Skolemizing and flattening,

Inducting on l on formula 1,

we get 3 subgoals:

path_list_induction.1:

$\{-1\} \quad \forall (t: T): p'(\text{cons}(t, \text{null}))$ $\{-2\} \quad \forall (t: T, \text{tail}: (\text{path_list?})): p'(\text{tail}) \supset p'(\text{cons}(t, \text{tail}))$
$\{1\} \quad \text{path_list?}(l')$ $\{2\} \quad p'(l')$

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of path_list_induction.1.

path_list_induction.2:

$\{-1\} \quad \forall (t: T): p'(\text{cons}(t, \text{null}))$ $\{-2\} \quad \forall (t: T, \text{tail}: (\text{path_list?})): p'(\text{tail}) \supset p'(\text{cons}(t, \text{tail}))$
$\{1\} \quad \text{path_list?}(\text{null}) \supset p'(\text{null})$

Repeatedly Skolemizing and flattening,

Installing automatic rewrites from: path_list? length

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of path_list_induction.2.

path_list_induction.3:

$\{-1\} \quad \forall (t: T): p'(\text{cons}(t, \text{null}))$ $\{-2\} \quad \forall (t: T, \text{tail}: (\text{path_list?})): p'(\text{tail}) \supset p'(\text{cons}(t, \text{tail}))$
$\{1\} \quad \forall (\text{cons1_var}: T, \text{cons2_var}: \text{list}[T]):$ $\quad (\text{path_list?}(\text{cons2_var}) \supset p'(\text{cons2_var})) \supset$ $\quad \text{path_list?}(\text{cons}(\text{cons1_var}, \text{cons2_var})) \supset p'(\text{cons}(\text{cons1_var}, \text{cons2_var}))$

Repeatedly Skolemizing and flattening,
 Installing automatic rewrites from: path_list? length
 Simplifying, rewriting, and recording with decision procedures,
 Case splitting on cons2_var!1 = null,
 we get 2 subgoals:
 path_list_induction.3.1:

{-1}	cons2_var' = null
{-2}	length(cons2_var') ≥ 1 ⊃ p'(cons2_var')
{-3}	1 + length(cons2_var') ≥ 1
{-4}	∀ (t: T): p'(cons(t, null))
{-5}	∀ (t: T, tail: (path_list?)): p'(tail) ⊃ p'(cons(t, tail))
{1}	
p'(cons(cons1_var', cons2_var'))	

Hiding formulas: -2, -3,
 Instantiating quantified variables,
 Simplifying, rewriting, and recording with decision procedures,
 This completes the proof of path_list_induction.3.1.
 path_list_induction.3.2:

{-1}	length(cons2_var') ≥ 1 ⊃ p'(cons2_var')
{-2}	1 + length(cons2_var') ≥ 1
{-3}	∀ (t: T): p'(cons(t, null))
{-4}	∀ (t: T, tail: (path_list?)): p'(tail) ⊃ p'(cons(t, tail))
{1}	
cons2_var' = null	
{2}	p'(cons(cons1_var', cons2_var'))

Simplifying, rewriting, and recording with decision procedures,
 Rewriting using -4, matching in *,
 This completes the proof of path_list_induction.3.2.
 Q.E.D.

C.99.12 Graph_Util.concat_path_TCC1

Terse proof for concat_path_TCC1.
 concat_path_TCC1:

{1}	∀ (lr: (concatable?)): cons?[T](PROJ_2(lr))
-----	---

Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of concat_path_TCC1.
 Q.E.D.

C.99.13 Graph_Util.concat_path_TCC2

Terse proof for concat_path_TCC2.
 concat_path_TCC2:

{1}	∀ (lr: (concatable?)): path_list?(append[T](PROJ_1(lr), cdr[T](PROJ_2(lr))))
-----	--

Repeatedly Skolemizing and flattening,
 Expanding the definition of concatable?,
 Applying disjunctive simplification to flatten sequent,

C Proof scripts

Hiding formulas: -3,
Expanding the definition of `path_list?`,
Rewriting using `length_append`, matching in `*`,
Rewriting using `length_cdr`, matching in `*`,
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `concat_path_TCC2`.
Q.E.D.

C.99.14 Graph_Util.length_concat_path

Terse proof for `length_concat_path`.

`length_concat_path`:

$$\frac{\{1\} \quad \forall (l, r: \text{list}[T]): \quad \text{concatable?}(l, r) \supset \quad \text{length}(\text{concat_path}(l, r)) = \text{length}(l) + \text{length}(r) - 1}{}$$

Repeatedly Skolemizing and flattening,
Expanding the definition of `concat_path`,
Rewriting using `length_append`, matching in `*`,
Rewriting using `length_cdr`, matching in `*`,
we get 2 subgoals:

`length_concat_path.1`:

$$\frac{\{1\} \quad \text{concatable?}(l', r')}{\{1\} \quad \text{length}(r') - 1 + \text{length}(l') = \text{length}(l') + \text{length}(r') - 1}$$

Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `length_concat_path.1`.

`length_concat_path.2`:

$$\frac{\{1\} \quad \text{concatable?}(l', r')}{\begin{array}{l} \{1\} \quad \text{length}(r') > 0 \\ \{2\} \quad \text{length}(\text{cdr}(r')) = \text{length}(r') - 1 \end{array}}$$

Expanding the definition of `concatable?`,
Expanding the definition of `path_list?`,
which is trivially true.
This completes the proof of `length_concat_path.2`.
Q.E.D.

C.100 Proofs for Hoare (hoare.pvs)

C.100.1 Hoare.valid_TCC1

Terse proof for `valid_TCC1`.

valid_TCC1:

$$\{1\} \quad \forall (P: \text{PRED}[\text{State}], c: [\text{State} \rightarrow \text{StmtResult}[\text{State}]], s):$$

$$P(s) \wedge \text{OK?}(c(s)) \supset$$

$$\text{OK?}[\text{State}](c(s)) \vee$$

$$\text{Break?}[\text{State}](c(s)) \vee$$

$$\text{Continue?}[\text{State}](c(s)) \vee$$

$$\text{Return?}[\text{State}](c(s)) \vee$$

$$\text{Switch?}[\text{State}](c(s)) \vee$$

$$\text{Default?}[\text{State}](c(s)) \vee \text{Exception?}[\text{State}](c(s))$$

Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of valid_TCC1.
 Q.E.D.

C.100.2 Hoare.valid_TCC2

Terse proof for valid_TCC2.

valid_TCC2:

$$\{1\} \quad \forall (P: \text{PRED}[\text{State}], C: [\text{StmtResult}[\text{State}] \rightarrow \text{StmtResult}[\text{State}]], s):$$

$$P(s) \wedge \text{OK?}(C(\text{OK}(s))) \supset$$

$$\text{OK?}[\text{State}](C(\text{OK}[\text{State}](s))) \vee$$

$$\text{Break?}[\text{State}](C(\text{OK}[\text{State}](s))) \vee$$

$$\text{Continue?}[\text{State}](C(\text{OK}[\text{State}](s))) \vee$$

$$\text{Return?}[\text{State}](C(\text{OK}[\text{State}](s))) \vee$$

$$\text{Switch?}[\text{State}](C(\text{OK}[\text{State}](s))) \vee$$

$$\text{Default?}[\text{State}](C(\text{OK}[\text{State}](s))) \vee$$

$$\text{Exception?}[\text{State}](C(\text{OK}[\text{State}](s)))$$

Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of valid_TCC2.
 Q.E.D.

C.100.3 Hoare.valid_and

Terse proof for valid_and.

valid_and:

$$\{1\} \quad \forall (P, Q, R: \text{PRED}[\text{State}], c: [\text{State} \rightarrow \text{StmtResult}[\text{State}]]):$$

$$\text{valid}(P, c, Q) \wedge \text{valid}(P, c, R) \supset$$

$$\text{valid}(P, c, \lambda (s: \text{State}): Q(s) \wedge R(s))$$

Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of valid_and.
 Q.E.D.

C.100.4 Hoare.valid_implied

Terse proof for valid_implied.

valid_implied:

$$\frac{\{1\} \quad \forall (P, Q, R: \text{PRED}[\text{State}], c: [\text{State} \rightarrow \text{StmtResult}[\text{State}]]) : \text{valid}(P, c, Q) \wedge \text{valid}(P, c, \lambda (s: \text{State}): Q(s) \supset R(s)) \supset \text{valid}(P, c, \lambda (s: \text{State}): Q(s) \wedge R(s))}{}$$

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `valid_implied`.
Q.E.D.

C.100.5 Hoare.Valid_and

Terse proof for `Valid_and`.

`Valid_and`:

$$\frac{\{1\} \quad \forall (P, Q, R: \text{PRED}[\text{State}], c: [\text{State} \rightarrow \text{StmtResult}[\text{State}]]) : \text{Valid}(P, c, Q) \wedge \text{Valid}(P, c, R) \supset \text{Valid}(P, c, \lambda (s: \text{State}): Q(s) \wedge R(s))}{}$$

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `Valid_and`.
Q.E.D.

C.101 Proofs for IA32 (constants.pvs)

C.101.1 IA32.max_linear_offset_val

The \LaTeX code for this proof is broken.

C.101.2 IA32.mem_addr_4g_TCC1

Terse proof for `mem_addr_4g_TCC1`.

`mem_addr_4g_TCC1`:

$$\frac{\{1\} \quad \forall (a: \text{Memory_Address_4G}): \text{in_memory}(\text{min_linear}, \text{max_linear})(a)}{}$$

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `mem_addr_4g_TCC1`.
Q.E.D.

C.102 Proofs for Import_All (everything.pvs)

This theory contains no provable formal statements.

C.103 Proofs for IntegralConversions (conversions.pvs)

This theory contains no provable formal statements.

C.104 Proofs for IntegralPromotions (conversions.pvs)

C.104.1 IntegralPromotions.int_conversion_TCC1

Terse proof for `int_conversion_TCC1`.

`int_conversion_TCC1:`

<pre> {1} ∃ (x: [typ: Cpp_Subtype(cv(extend[Cpp_Type_, Cpp_Type, bool, FALSE] ({t: Cpp_Type char?(t) ∨ schar?(t) ∨ uchar?(t) ∨ short?(t) ∨ ushort?(t)}))) → [(range(typ)) → (range(int))])]: TRUE </pre>

Instantiating the top quantifier in 1 with the terms: $(\lambda (\text{typ}: \text{Cpp_Subtype}(\text{cv}(\text{extend}[\text{Cpp_Type_}, \text{Cpp_Type}, \text{bool}, \text{FALSE}] (\{t: \text{Cpp_Type} \mid \text{char?}(t) \vee \text{schar?}(t) \vee \text{uchar?}(t) \vee \text{short?}(t) \vee \text{ushort?}(t)\})))) \lambda (r: (\text{range}(\text{typ}))) : 0)$,

we get 2 subgoals:

`int_conversion_TCC1.1:`

<pre> {1} TRUE </pre>

which is trivially true.

This completes the proof of `int_conversion_TCC1.1`.

`int_conversion_TCC1.2:`

<pre> {1} ∃ (typ: Cpp_Subtype(cv(extend[Cpp_Type_, Cpp_Type, bool, FALSE] ({t: Cpp_Type char?(t) ∨ schar?(t) ∨ uchar?(t) ∨ short?(t) ∨ ushort?(t)}))), r: (range(typ))): range(int)(0) </pre>
--

Repeatedly Skolemizing and flattening,

Keeping (1) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

Rewriting using `int_binary`, matching in *,

Using lemma `min_int`,

Using lemma `max_int`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `int_conversion_TCC1.2`.

Q.E.D.

C.104.2 IntegralPromotions.int_conversion_spec_TCC1

Terse proof for `int_conversion_spec_TCC1`.

int_conversion_spec_TCC1:

```
{1}  ∇ (typ:
      Cpp_Subtype(cv(extend[Cpp_Type_, Cpp_Type, bool, FALSE]
                        ({t: Cpp_Type |
                          char?(t) ∨
                          schar?(t) ∨
                          uchar?(t) ∨ short?(t) ∨ ushort?(t)}))),
      n: (range(typ))):
      Cpp_Type?(int) ∧ cv(non_bool_integral_enum?)(int)
```

Repeatedly Skolemizing and flattening,
 Keeping (1) and hiding *,
 Applying propositional simplification,
 we get 2 subgoals:

int_conversion_spec_TCC1.1:

```
{1}  Cpp_Type?(int)
```

Expanding the definition of Cpp_Type?,
 Expanding the definition of subterm,
 Repeatedly Skolemizing and flattening,
 Replacing using formula -1,

Installing automatic rewrites from: no_pointers_to_bitfield? no_pointers_to_references?
 no_cv_references? no_reference_to_reference? no_reference_to_bitfields? no_pointer_to_member_to_reference?
 no_pointer_to_member_to_cv_void? no_cv_void_parameter? no_array_of_references? no_array_of_bitfields?
 no_array_of_cv_void? no_array_of_function? no_array_of_abstract_class? no_pointer_or_ref_to_incomplete_array_par
 no_array_return_type? no_function_return_type? bitfield_underlying_integral_or_enum_type? cv_array?
 enum_underlying_integral? enum_constants? const_volatile? const_stutter? volatile_stutter?
 no_cv_class? no_cv_union? no_cv_function? no_reference_to_void? no_cv_void?

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 This completes the proof of int_conversion_spec_TCC1.1.

int_conversion_spec_TCC1.2:

```
{1}  cv(non_bool_integral_enum?)(int)
```

Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of int_conversion_spec_TCC1.2.
 Q.E.D.

C.104.3 IntegralPromotions.int_conversion_TCC2

Terse proof for int_conversion_TCC2.

int_conversion_TCC2:

```
{1}  ∇ (typ:
      Cpp_Subtype(cv(extend[Cpp_Type_, Cpp_Type, bool, FALSE]
                        ({t: Cpp_Type |
                          char?(t) ∨
                          schar?(t) ∨
                          uchar?(t) ∨ short?(t) ∨ ushort?(t)}))))):
      cv(non_bool_integral_enum?)(typ)
```

Repeatedly Skolemizing and flattening,
Hiding formulas: -1,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `int_conversion_TCC2`.
Q.E.D.

C.104.4 IntegralPromotions.int_conversion_TCC3

Terse proof for `int_conversion_TCC3`.

`int_conversion_TCC3`:

```
{1}  ∃ (typ:
      Cpp_Subtype(cv(extend[Cpp_Type_, Cpp_Type, bool, FALSE]
                        ({t: Cpp_Type |
                          char?(t) ∨
                          schar?(t) ∨
                          uchar?(t) ∨ short?(t) ∨ ushort?(t)})))):
      Cpp_Type?(int) ∧ cv(non_bool_integral_enum?)(int)
```

Repeatedly Skolemizing and flattening,
Keeping (1) and hiding *,
Applying propositional simplification,
we get 2 subgoals:

`int_conversion_TCC3.1`:

```
{1}  Cpp_Type?(int)
```

Expanding the definition of `Cpp_Type?`,
Expanding the definition of subterm,
Repeatedly Skolemizing and flattening,
Replacing using formula -1,

Installing automatic rewrites from: `no_pointers_to_bitfield?` `no_pointers_to_references?`
`no_cv_references?` `no_reference_to_reference?` `no_reference_to_bitfields?` `no_pointer_to_member_to_reference?`
`no_pointer_to_member_to_cv_void?` `no_cv_void_parameter?` `no_array_of_references?` `no_array_of_bitfields?`
`no_array_of_cv_void?` `no_array_of_function?` `no_array_of_abstract_class?` `no_pointer_or_ref_to_incomplete_array_parameter?`
`no_array_return_type?` `no_function_return_type?` `bitfield_underlying_integral_or_enum_type?` `cv_array?`
`enum_underlying_integral?` `enum_constants?` `const_volatile?` `const_stutter?` `volatile_stutter?`
`no_cv_class?` `no_cv_union?` `no_cv_function?` `no_reference_to_void?` `no_cv_void?`

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `int_conversion_TCC3.1`.

`int_conversion_TCC3.2`:

```
{1}  cv(non_bool_integral_enum?)(int)
```

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `int_conversion_TCC3.2`.
Q.E.D.

C.104.5 IntegralPromotions.bool2int_TCC1

Terse proof for `bool2int_TCC1`.

bool2int_TCC1:

$$\{1\} \quad \forall (b: \text{bool}): b \supset \text{range}(\text{int})(1)$$

Repeatedly Skolemizing and flattening,
 Expanding the definition of range,
 Expanding the definition of range_integral,
 Rewriting using int_binary, matching in *,
 Using lemma min_int,
 Using lemma max_int,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 This completes the proof of bool2int_TCC1.
 Q.E.D.

C.104.6 IntegralPromotions.bool2int_TCC2

Terse proof for bool2int_TCC2.

bool2int_TCC2:

$$\{1\} \quad \forall (b: \text{bool}): \neg b \supset \text{range}(\text{int})(0)$$

Repeatedly Skolemizing and flattening,
 Expanding the definition of range,
 Expanding the definition of range_integral,
 Rewriting using int_binary, matching in *,
 Using lemma max_int,
 Using lemma min_int,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 This completes the proof of bool2int_TCC2.
 Q.E.D.

C.105 Proofs for Interpreted_Data (abstract_data.pvs)

C.105.1 Interpreted_Data.interpreted_data_type?_TCC1

Terse proof for interpreted_data_type?_TCC1.

interpreted_data_type?_TCC1:

$$\{1\} \quad \forall (\text{idt}: \text{Interpreted_data_type}):$$

$$\quad (\forall (l: \text{list}[\text{Byte}], a: \text{Address}):$$

$$\quad \quad \text{valid?}(\text{uidt}(\text{idt}))(l, a) \equiv \text{up?}(\text{from_byte}(\text{idt})(l, a)))$$

$$\quad \wedge$$

$$\quad (\forall (d: \text{Data}, a: \text{Address}): \text{valid?}(\text{uidt}(\text{idt}))(\text{to_byte}(\text{idt})(d, a), a)) \wedge$$

$$\quad \quad \text{uninterpreted_data_type?}(\text{idt}' \text{uidt}))$$

$$\quad \supset$$

$$\quad (\forall (d_1: \text{Data}, a_1: \text{Address}): \text{up?}[\text{Data}](\text{from_byte}(\text{idt})(\text{to_byte}(\text{idt})(d_1, a_1), a_1)))$$

Repeatedly Skolemizing and flattening,
 Instantiating quantified variables,
 Instantiating quantified variables,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `interpreted_data_type?_TCC1`.
Q.E.D.

C.105.2 Interpreted_Data.pod_is_interpreted_data

Terse proof for `pod_is_interpreted_data`.
`pod_is_interpreted_data`:

$$\frac{}{\{1\} \quad \forall (x: (\text{pod_data_type?})): \text{interpreted_data_type?}(x)}$$

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `pod_is_interpreted_data`.
Q.E.D.

C.105.3 Interpreted_Data.pod_size

Terse proof for `pod_size`.
`pod_size`:

$$\frac{}{\{1\} \quad \forall (dt: (\text{pod_data_type?})): \text{size}(\text{uidt}(dt)) > 0}$$

Repeatedly Skolemizing and flattening,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Expanding the definition of `pod_data_type?`,
which is trivially true.
This completes the proof of `pod_size`.
Q.E.D.

C.105.4 Interpreted_Data.data_type_length

Terse proof for `data_type_length`.
`data_type_length`:

$$\frac{}{\{1\} \quad \forall (\text{addr}: \text{Address}, \text{idt}: (\text{interpreted_data_type?}), l: \text{list}[\text{Byte}]): \\ \text{valid?}(\text{uidt}(\text{idt}))(l, \text{addr}) \supset \text{length}(l) = \text{size}(\text{uidt}(\text{idt}))}$$

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `data_type_length`.
Q.E.D.

C.105.5 Interpreted_Data.length_to_byte

Terse proof for `length_to_byte`.
`length_to_byte`:

$$\frac{}{\{1\} \quad \forall (\text{addr}: \text{Address}, \text{idt}: (\text{interpreted_data_type?}), \text{data}: \text{Data}): \\ \text{length}(\text{to_byte}(\text{idt})(\text{data}, \text{addr})) = \text{size}(\text{uidt}(\text{idt}))}$$

Repeatedly Skolemizing and flattening,
Expanding the definition of `interpreted_data_type?`,
Expanding the definition of `uninterpreted_data_type?`,

C Proof scripts

Applying disjunctive simplification to flatten sequent,
Instantiating the top quantifier in -1 with the terms: (to_byte(idt!1)(data!1, addr!1) addr!1),
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Instantiating quantified variables,
This completes the proof of length_to_byte.
Q.E.D.

C.105.6 Interpreted_Data.valid_from_byte

Terse proof for valid_from_byte.

valid_from_byte:

$$\{1\} \quad \forall (\text{addr}: \text{Address}, \text{idt}: (\text{interpreted_data_type?}), l: \text{list}[\text{Byte}]):$$

$$\text{idt}'\text{uidt}'\text{valid?}(l, \text{addr}) \supset \text{up?}(\text{from_byte}(\text{idt})(l, \text{addr}))$$

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of valid_from_byte.
Q.E.D.

C.105.7 Interpreted_Data.valid_iff_from_byte

Terse proof for valid_iff_from_byte.

valid_iff_from_byte:

$$\{1\} \quad \forall (\text{addr}: \text{Address}, \text{idt}: (\text{interpreted_data_type?}), l: \text{list}[\text{Byte}]):$$

$$\text{idt}'\text{uidt}'\text{valid?}(l, \text{addr}) \Leftrightarrow \text{up?}(\text{from_byte}(\text{idt})(l, \text{addr}))$$

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of valid_iff_from_byte.
Q.E.D.

C.105.8 Interpreted_Data.valid_to_byte

Terse proof for valid_to_byte.

valid_to_byte:

$$\{1\} \quad \forall (\text{addr}: \text{Address}, \text{idt}: (\text{interpreted_data_type?}), \text{data}: \text{Data}):$$

$$\text{idt}'\text{uidt}'\text{valid?}(\text{to_byte}(\text{idt})(\text{data}, \text{addr}), \text{addr})$$

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of valid_to_byte.
Q.E.D.

C.105.9 Interpreted_Data.up_from_byte_to_byte

Terse proof for up_from_byte_to_byte.

up_from_byte_to_byte:

$$\{1\} \quad \forall (\text{addr}: \text{Address}, \text{idt}: (\text{interpreted_data_type?}), \text{data}: \text{Data}):$$

$$\text{up?}(\text{from_byte}(\text{idt})(\text{to_byte}(\text{idt})(\text{data}, \text{addr}), \text{addr}))$$

Repeatedly Skolemizing and flattening,
 Expanding the definition of interpreted_data_type?,
 Applying disjunctive simplification to flatten sequent,
 Instantiating quantified variables,
 Instantiating quantified variables,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 This completes the proof of up_from_byte_to_byte.
 Q.E.D.

C.105.10 Interpreted_Data.from_byte_to_byte

Terse proof for from_byte_to_byte.

from_byte_to_byte:

$$\{1\} \quad \forall (\text{addr}: \text{Address}, \text{idt}: (\text{interpreted_data_type?}), \text{data}: \text{Data}):$$

$$\text{from_byte}(\text{idt})(\text{to_byte}(\text{idt})(\text{data}, \text{addr}), \text{addr}) = \text{up}(\text{data})$$

Trying repeated skolemization, instantiation, and if-lifting,
 Applying extensionality,
 This completes the proof of from_byte_to_byte.
 Q.E.D.

C.105.11 Interpreted_Data.in_blessed_memory_subset

Terse proof for in_blessed_memory_subset.

in_blessed_memory_subset:

$$\{1\} \quad \forall (\text{addr}: \text{Address}, \text{idt}: (\text{interpreted_data_type?}), \text{address_range}: \text{PRED}[\text{Address}]):$$

$$\text{in_blessed_memory?}(\text{idt}, \text{addr}, \text{address_range}) \supset$$

$$(\text{address_block}(\text{addr}, \text{size}(\text{uidt}(\text{idt}))) \subseteq \text{address_range})$$

Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of in_blessed_memory_subset.
 Q.E.D.

C.106 Proofs for Interpreted_Data_Lift (abstract_data.pvs)

This theory contains no provable formal statements.

C.107 Proofs for IterationStatements1 (statements.pvs)

C.107.1 IterationStatements1.iterate_while_TCC1

Terse proof for iterate_while_TCC1.

iterate_while_TCC1:

$$\{1\} \quad \forall (n: \text{nat}, s: \text{State}): \neg n = 0 \supset n - 1 \geq 0$$

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `iterate_while_TCC1`.
Q.E.D.

C.107.2 IterationStatements1.iterate_while_TCC2

Terse proof for `iterate_while_TCC2`.

`iterate_while_TCC2`:

$$\{1\} \quad \forall (n: \text{nat}, s: \text{State}): \neg n = 0 \supset n - 1 < n$$

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `iterate_while_TCC2`.
Q.E.D.

C.107.3 IterationStatements1.while_termination_point?_TCC1

Terse proof for `while_termination_point?_TCC1`.

`while_termination_point?_TCC1`:

$$\{1\} \quad \forall (b_ex: [\text{State} \rightarrow \text{ExprResult}[\text{State}, \text{bool}]], \text{body}: [\text{State} \rightarrow \text{StmtResult}[\text{State}]], \\ s: \text{State}, n: \text{nat}, \text{res}: \text{StmtResult}[\text{State}]): \\ \text{OK?}[\text{res}] \wedge \text{res} = \text{iterate_while}(n, b_ex, \text{body})(s) \supset \\ \text{OK?}[\text{State}](\text{res}) \vee \\ \text{Break?}[\text{State}](\text{res}) \vee \\ \text{Continue?}[\text{State}](\text{res}) \vee \\ \text{Return?}[\text{State}](\text{res}) \vee \\ \text{Switch?}[\text{State}](\text{res}) \vee \text{Default?}[\text{State}](\text{res}) \vee \text{Exception?}[\text{State}](\text{res})$$

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `while_termination_point?_TCC1`.
Q.E.D.

C.107.4 IterationStatements1.while_TCC1

Terse proof for `while_TCC1`.

`while_TCC1`:

$$\{1\} \quad \forall (b_ex: [\text{State} \rightarrow \text{ExprResult}[\text{State}, \text{bool}]], \text{body}: [\text{State} \rightarrow \text{StmtResult}[\text{State}]], \\ s: \text{State}): \\ (\exists (n: \text{nat}): \text{while_termination_point?}(b_ex, \text{body}, s)(n)) \supset \\ \text{nonempty?}[\text{nat}](\text{while_termination_point?}(b_ex, \text{body}, s))$$

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `while_TCC1`.
Q.E.D.

C.107.5 IterationStatements1.nonzero_min_while_termination_point

Terse proof for `nonzero_min_while_termination_point`.

nonzero_min_while_termination_point:

$$\frac{\{1\} \quad \forall (b_ex: [State \rightarrow ExprResult[State, bool]], body: [State \rightarrow StmtResult[State]]), \quad s: State):}{(\exists (n: nat): while_termination_point?(b_ex, body, s)(n)) \wedge \neg \min[nat](while_termination_point?(b_ex, body, s)) = 0 \supset OK?(b_ex(s)) \wedge data(b_ex(s))}$$

Repeatedly Skolemizing and flattening,

Rewriting using min_def, matching in *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of nonzero_min_while_termination_point.

Q.E.D.

C.107.6 IterationStatements1.while_as_if_while

Terse proof for while_as_if_while.

while_as_if_while:

$$\frac{\{1\} \quad \forall (b_ex: [State \rightarrow ExprResult[State, bool]], body: [State \rightarrow StmtResult[State]]):}{while(b_ex, body) = if_else(b_ex, body \#\# catch_continue \#\# while(b_ex, body) \#\# catch_break, skip)}$$

Expanding the definition of while,

Expanding the definition of if_else,

Repeatedly Skolemizing and flattening,

Applying decompose-equality,

we get 3 subgoals:

while_as_if_while.1:

```

{1} IF ∃ (n: nat): while_termination_point?(b_ex', body', x')(n)
      THEN (iterate_while(min[nat](while_termination_point?(b_ex', body', x')), b_ex',
                          body')
            ## e2s(b_ex')
            ## catch_break)
            (x')
ELSE Hang
ENDIF
=
(b_ex' ##
  (λ (b: bool):
    IF b
      THEN (body' ## catch_continue ##
            (λ (s: State):
              IF ∃ (n: nat): while_termination_point?(b_ex', body', s)(n)
                THEN (iterate_while(min[nat]
                                     (while_termination_point?(b_ex',
                                                                body',
                                                                s)),
                                     b_ex', body')
                      ## e2s(b_ex')
                      ## catch_break)
                      (s)
                ELSE Hang
                ENDIF))
            ## catch_break
      ELSE skip
      ENDIF))
  (x')

```

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 2 subgoals:

while_as_if_while.1.1:

{-1}	$\exists (n: \text{nat}): \text{while_termination_point?}(\text{b_ex}', \text{body}', x')(n)$
{1}	<pre> iterate_while(min[nat](while_termination_point?(b_ex', body', x')), b_ex', body') ## e2s(b_ex') ## catch_break (x') = (b_ex' ## (\lambda (b: bool): IF b THEN (body' ## catch_continue ## (\lambda (s: State): IF \exists (n: nat): while_termination_point?(b_ex', body', s)(n) THEN (iterate_while(min[nat] (while_termination_point?(b_ex', body', s)), b_ex', body') ## e2s(b_ex') ## catch_break (s) ELSE Hang ENDIF)) ## catch_break ELSE skip ENDIF)) (x') </pre>

Case splitting on $\min(\text{while_termination_point?}(\text{b_ex!1}, \text{body!1}, \text{x!1})) = 0$,

we get 2 subgoals:

C Proof scripts

`while_as_if_while.1.1.1:`

```

{-1} min(while_termination_point?(b_ex', body', x')) = 0
{-2} ∃ (n: nat): while_termination_point?(b_ex', body', x')(n)
-----
{1} (iterate_while(min [nat](while_termination_point?(b_ex', body', x')), b_ex', body') ##
    e2s(b_ex')
    ## catch_break)
    (x')
=
(b_ex' ##
  (λ (b: bool):
    IF b
      THEN (body' ## catch_continue ##
            (λ (s: State):
              IF ∃ (n: nat): while_termination_point?(b_ex', body', s)(n)
                THEN (iterate_while(min [nat]
                                     (while_termination_point?(b_ex',
                                                                body',
                                                                s)),
                                     b_ex', body')
                        ## e2s(b_ex')
                        ## catch_break)
                (s)
              ELSE Hang
              ENDIF))
            ## catch_break)
    ELSE skip
    ENDIF))
  (x')

```

Replacing using formula -1,

Rewriting using `min_def`, matching in `*`,

Expanding the definition of `minimum?`,

Expanding the definition of `while_termination_point?`,

Expanding the definition of `iterate_while`,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `while_as_if_while.1.1.1`.

while_as_if_while.1.1.2:

{-1}	$\exists (n: \text{nat}): \text{while_termination_point?}(\text{b_ex}', \text{body}', x')(n)$
{1}	$\text{min}(\text{while_termination_point?}(\text{b_ex}', \text{body}', x')) = 0$
{2}	<pre> (iterate_while(min[nat](while_termination_point?(b_ex', body', x')), b_ex', body') ## e2s(b_ex') ## catch_break) (x') = (b_ex' ## (\lambda (b: bool): IF b THEN (body' ## catch_continue ## (\lambda (s: State): IF \exists (n: nat): while_termination_point?(b_ex', body', s)(n) THEN (iterate_while(min[nat] (while_termination_point?(b_ex', body', s)), b_ex', body') ## e2s(b_ex') ## catch_break) (s) ELSE Hang ENDIF)) ## catch_break ELSE skip ENDIF)) (x') </pre>

Using lemma nonzero_min_while_termination_point,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Expanding the definition of iterate_while,

Expanding the definition of ##,

Expanding the definition of lift,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 2 subgoals:

while_as_if_while.1.1.2.1:

{-1}	$\exists (n: \text{nat}): \text{while_termination_point?}(\text{b_ex}', \text{body}', x')(n)$
{-2}	$\text{OK?}(\text{b_ex}'(x'))$
{-3}	$\text{data}(\text{b_ex}'(x'))$
{-4}	$\text{OK?}(\text{e2s}(\text{b_ex}')(x'))$
{1}	$\text{min}(\text{while_termination_point?}(\text{b_ex}', \text{body}', x')) = 0$
{2}	$\text{catch_break}(\text{lift}(\text{e2s}(\text{b_ex}'))$ $\quad (\text{lift}(\text{iterate_while}(\text{min} [\text{nat}] (\text{while_termination_point?}(\text{b_ex}', \text{body}', x')) - 1,$ $\quad \quad \quad \text{b_ex}', \text{body}'))$ $\quad (\text{catch_continue}(\text{body}'(\text{state}(\text{e2s}(\text{b_ex}')(x'))))))))$ $=$ $\text{catch_break}(\text{lift}(\lambda (s: \text{State}):$ $\quad \text{IF } \exists (n: \text{nat}): \text{while_termination_point?}(\text{b_ex}', \text{body}', s)(n)$ $\quad \quad \text{THEN } \text{catch_break}(\text{lift}(\text{e2s}(\text{b_ex}'))$ $\quad \quad \quad (\text{iterate_while}(\text{min} [\text{nat}]$ $\quad \quad \quad \quad (\text{while_termination_point?}(\text{b_ex}', \text{body}', s)),$ $\quad \quad \quad \quad \text{b_ex}',$ $\quad \quad \quad \quad \text{body}'))$ $\quad \quad \quad (s)))$ $\quad \quad \quad \text{ELSE Hang}$ $\quad \quad \quad \text{ENDIF})$ $\quad (\text{catch_continue}(\text{body}'(\text{state}(\text{b_ex}'(x'))))))))$

Expanding the definition of e2s,

Expanding the definition of lift,

Case splitting on $\text{OK?}(\text{catch_continue}(\text{body}'!1(\text{state}(\text{b_ex}'!1(x!1))))$,

we get 2 subgoals:

while_as_if_while.1.1.2.1.1:

<pre> {-1} OK?(catch_continue(body'(state(b_ex'(x'))))) {-2} ∃ (n: nat): while_termination_point?(b_ex', body', x')(n) {-3} OK?(b_ex'(x')) {-4} data(b_ex'(x')) {-5} OK?(e2s(b_ex')(x')) </pre>	<hr/> <pre> {1} min(while_termination_point?(b_ex', body', x')) = 0 {2} catch_break(CASES CASES catch_continue(body'(state(b_ex'(x')))) OF OK(state_1): iterate_while(min [nat](while_termination_point?(b_ex', body', x') - 1, b_ex', body') (state_1) ELSE catch_continue(body'(state(b_ex'(x')))) ENDCASES OF OK(state_1): e2s(b_ex')(state_1) ELSE CASES catch_continue(body'(state(b_ex'(x')))) OF OK(state_1): iterate_while(min [nat](while_termination_point?(b_ex', body', x') - 1, b_ex', body') (state_1) ELSE catch_continue(body'(state(b_ex'(x')))) ENDCASES ENDCASES) = catch_break(CASES catch_continue(body'(state(b_ex'(x')))) OF OK(state_1): IF ∃ (n: nat): while_termination_point?(b_ex', body', state_1)(n) THEN catch_break(CASES iterate_while(min [nat] (while_termination_point? (b_ex', body', state_1)), b_ex', body') (state_1) OF OK(state_1): e2s(b_ex')(state_1) ELSE iterate_while(min [nat] (while_termination_point? (b_ex', body', state_1)), b_ex', body') (state_1) ENDCASES) ELSE Hang ENDIF ELSE catch_continue(body'(state(b_ex'(x')))) ENDCASES) </pre>
---	--

Simplifying, rewriting, and recording with decision procedures,

Case splitting on EXISTS (n: nat): while_termination_point?(b_ex!1, body!1, state(catch_continue (body!1(state(b_ex!1(x!1)))))) (n),

we get 2 subgoals:

while_as_if_while.1.1.2.1.1.1:

```

{-1}  ∃ (n: nat):
      while_termination_point?(b_ex', body',
                               state(catch_continue(body'(state(b_ex'(x')))))
                               (n))
{-2}  OK?(catch_continue(body'(state(b_ex'(x')))))
{-3}  ∃ (n: nat): while_termination_point?(b_ex', body', x')(n)
{-4}  OK?(b_ex'(x'))
{-5}  data(b_ex'(x'))
{-6}  OK?(e2s(b_ex')(x'))
-----
{1}   min(while_termination_point?(b_ex', body', x') = 0
{2}   catch_break(CASES iterate_while(min [nat](while_termination_point?(b_ex', body', x') - 1,
                                     b_ex', body')
                                     (state(catch_continue(body'(state(b_ex'(x'))))) OF
                                     OK(state_1): e2s(b_ex')(state_1)
                                     ELSE CASES catch_continue(body'(state(b_ex'(x')))) OF
                                     OK(state_1):
                                     iterate_while(min [nat](while_termination_point?(b_ex', body', x') - 1,
                                     b_ex', body')
                                     (state_1)
                                     ELSE catch_continue(body'(state(b_ex'(x'))))
                                     ENDCASES
                                     ENDCASES)
=
IF ∃ (n: nat):
      while_termination_point?(b_ex', body',
                               state(catch_continue(body'(state(b_ex'(x')))))
                               (n))
THEN  catch_break(catch_break(CASES iterate_while(min [nat]
                                                    (while_termination_point?
                                                     (b_ex',
                                                      body',
                                                       state
                                                       (catch_continue
                                                        (body'
                                                         (state
                                                          (b_ex'(x'))))))),
                                                     b_ex', body')
                                                    (state(catch_continue
                                                         (body'
                                                          (state(b_ex'(x'))))) OF
                                                         OK(state_1): e2s(b_ex')(state_1)
                                                         ELSE iterate_while(min [nat]
                                                                           (while_termination_point?
                                                                            (b_ex',
                                                                             body',
                                                                              state
                                                                              (catch_continue
                                                                               (body'
                                                                                (state
                                                                                 (b_ex'(x'))))))))
                                                                           b_ex', body')
                                                                           (state(catch_continue
                                                                                (body'
                                                                                 (state(b_ex'(x'))))) OF
                                                                                OK(state_1): e2s(b_ex')(state_1)
                                                                                ELSE iterate_while(min [nat]
                                                                           (while_termination_point?
                                                                            (b_ex',
                                                                             body',
                                                                              state
                                                                              (catch_continue
                                                                               (body'
                                                                                (state
                                                                                 (b_ex'(x'))))))))
                                                                           b_ex', body')
                                                                           (state(catch_continue
                                                                                (body'
                                                                                 (state(b_ex'(x')))))
                                                                                ENDCASES)))
                                                    ENDCASES))
ELSE  catch_break(Hang)
ENDIF

```

Replacing using formula -1,

Case splitting on $\text{min}(\text{while_termination_point?}(\text{b_ex!1}, \text{body!1}, \text{x!1})) - 1 = \text{min}(\text{while_termination_point?}(\text{b_ex!1}, \text{body!1}, \text{state}(\text{catch_continue}(\text{body!1}(\text{state}(\text{b_ex!1}(\text{x!1}))))))$,

we get 2 subgoals:

while_as_if_while.1.1.2.1.1.1.1:

```

{-1} min(while_termination_point?(b_ex', body', x')) - 1 =
      min(while_termination_point?(b_ex', body',
                                     state(catch_continue(body'(state(b_ex'(x'))))))
{-2} ∃ (n: nat):
      while_termination_point?(b_ex', body',
                                state(catch_continue(body'(state(b_ex'(x'))))))
                                (n)
{-3} OK?(catch_continue(body'(state(b_ex'(x')))))
{-4} ∃ (n: nat): while_termination_point?(b_ex', body', x')(n)
{-5} OK?(b_ex'(x'))
{-6} data(b_ex'(x'))
{-7} OK?(e2s(b_ex')(x'))
-----
{1} min(while_termination_point?(b_ex', body', x')) = 0
{2} catch_break(CASES iterate_while(min [nat](while_termination_point?(b_ex', body', x')) - 1,
                                     b_ex', body')
                                     (state(catch_continue(body'(state(b_ex'(x')))))) OF
      OK(state_1): e2s(b_ex')(state_1)
      ELSE CASES catch_continue(body'(state(b_ex'(x')))) OF
        OK(state_1):
          iterate_while(min [nat](while_termination_point?(b_ex', body', x')) - 1,
                        b_ex', body')
                        (state_1)
          ELSE catch_continue(body'(state(b_ex'(x'))))
        ENDCASES
      ENDCASES)
=
catch_break(catch_break(CASES iterate_while(min [nat]
                                             (while_termination_point?(b_ex',
                                             b_ex', body')
                                             (state(catch_continue(body'(state
                                             (state
                                             (b_ex'(x'))))))
                                             OK(state_1): e2s(b_ex')(state_1)
                                             ELSE iterate_while(min [nat]
                                             (while_termination_point?(b_ex',
                                             b_ex', body')
                                             (state(catch_continue(body'(state
                                             (state
                                             (b_ex'(x'))))))
                                             ENDCASES))

```

Replacing using formula -1,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `while_as_if_while.1.1.2.1.1.1.1.`

while_as_if_while.1.1.2.1.1.1.2:

```

{-1}  ∃ (n: nat):
      while_termination_point?(b_ex', body',
                               state(catch_continue(body'(state(b_ex'(x')))))
                               (n)))
{-2}  OK?(catch_continue(body'(state(b_ex'(x')))))
{-3}  ∃ (n: nat): while_termination_point?(b_ex', body', x')(n)
{-4}  OK?(b_ex'(x'))
{-5}  data(b_ex'(x'))
{-6}  OK?(e2s(b_ex')(x'))
-----
{1}   min(while_termination_point?(b_ex', body', x') - 1 =
      min(while_termination_point?(b_ex', body',
                                     state(catch_continue(body'(state(b_ex'(x')))))
                                     (n)))
      min(while_termination_point?(b_ex', body', x') = 0
{2}   min(while_termination_point?(b_ex', body', x') = 0
{3}   catch_break(CASES iterate_while(min [nat](while_termination_point?(b_ex', body', x') - 1,
                                     b_ex', body')
                                     (state(catch_continue(body'(state(b_ex'(x'))))) OF
                                     OK(state_1): e2s(b_ex')(state_1)
                                     ELSE CASES catch_continue(body'(state(b_ex'(x')))) OF
                                     OK(state_1):
                                     iterate_while(min [nat](while_termination_point?(b_ex', body', x') - 1,
                                     b_ex', body')
                                     (state_1)
                                     ELSE catch_continue(body'(state(b_ex'(x'))))
                                     ENDCASES
                                     ENDCASES)
      =
      catch_break(catch_break(CASES iterate_while(min [nat]
                                                    (while_termination_point?(b_ex',
                                                    b_ex', body')
                                                    (state(catch_continue(body'
                                                    (state
                                                    (b_ex'
                                                    (x'))))))
                                                    OK(state_1): e2s(b_ex')(state_1)
                                                    ELSE iterate_while(min [nat]
                                                    (while_termination_point?(b_ex',
                                                    b_ex', body')
                                                    (state(catch_continue(body'
                                                    (state
                                                    (b_ex'
                                                    (x'))))))
                                                    OK(state_1): e2s(b_ex')(state_1)
                                                    ELSE iterate_while(min [nat]
                                                    (while_termination_point?(b_ex',
                                                    b_ex', body')
                                                    (state(catch_continue(body'
                                                    (state
                                                    (b_ex'
                                                    (x'))))))
                                                    ENDCASES))
      ENDCASES))

```

Hiding formulas: 3,

Rewriting using min_plus1, matching in *,

we get 2 subgoals:

`while_as_if_while.1.1.2.1.1.1.2.1:`

<div style="display: flex; flex-direction: column; gap: 5px;"> <div style="display: flex; align-items: flex-start;"> {-1} $\exists (n: \text{nat}):$ $\text{while_termination_point?}(\text{b_ex}', \text{body}',$ $\quad \text{state}(\text{catch_continue}(\text{body}'(\text{state}(\text{b_ex}'(x'))))))$ $\quad \quad \quad (n)$ </div> <div style="display: flex; align-items: flex-start;"> {-2} $\text{OK?}(\text{catch_continue}(\text{body}'(\text{state}(\text{b_ex}'(x'))))))$ </div> <div style="display: flex; align-items: flex-start;"> {-3} $\exists (n: \text{nat}): \text{while_termination_point?}(\text{b_ex}', \text{body}', x')(n)$ </div> <div style="display: flex; align-items: flex-start;"> {-4} $\text{OK?}(\text{b_ex}'(x'))$ </div> <div style="display: flex; align-items: flex-start;"> {-5} $\text{data}(\text{b_ex}'(x'))$ </div> <div style="display: flex; align-items: flex-start;"> {-6} $\text{OK?}(\text{e2s}(\text{b_ex}'(x')))$ </div> </div>	<hr style="border: 0.5px solid black;"/> <div style="display: flex; flex-direction: column; gap: 5px;"> <div style="display: flex; align-items: flex-start;"> {1} $1 + \min(\lambda (n: \text{nat}): \text{while_termination_point?}(\text{b_ex}', \text{body}', x')(1 + n)) - 1$ $=$ $\min(\text{while_termination_point?}(\text{b_ex}', \text{body}',$ $\quad \text{state}(\text{catch_continue}(\text{body}'(\text{state}(\text{b_ex}'(x'))))))$ </div> <div style="display: flex; align-items: flex-start;"> {2} $1 + \min(\lambda (n: \text{nat}): \text{while_termination_point?}(\text{b_ex}', \text{body}', x')(1 + n)) = 0$ </div> </div>
---	---

Case splitting on $(\text{LAMBDA } (n: \text{nat}): \text{while_termination_point?}(\text{b_ex!1}, \text{body!1}, x!1)(1 + n)) = \text{while_termination_point?}(\text{b_ex!1}, \text{body!1}, \text{state}(\text{catch_continue}(\text{body!1}(\text{state}(\text{b_ex!1}(x!1)))))$,

we get 2 subgoals:

`while_as_if_while.1.1.2.1.1.1.2.1.1:`

<div style="display: flex; flex-direction: column; gap: 5px;"> <div style="display: flex; align-items: flex-start;"> {-1} $(\lambda (n: \text{nat}): \text{while_termination_point?}(\text{b_ex}', \text{body}', x')(1 + n)) =$ $\text{while_termination_point?}(\text{b_ex}', \text{body}',$ $\quad \text{state}(\text{catch_continue}(\text{body}'(\text{state}(\text{b_ex}'(x'))))))$ </div> <div style="display: flex; align-items: flex-start;"> {-2} $\exists (n: \text{nat}):$ $\text{while_termination_point?}(\text{b_ex}', \text{body}',$ $\quad \text{state}(\text{catch_continue}(\text{body}'(\text{state}(\text{b_ex}'(x'))))))$ $\quad \quad \quad (n)$ </div> <div style="display: flex; align-items: flex-start;"> {-3} $\text{OK?}(\text{catch_continue}(\text{body}'(\text{state}(\text{b_ex}'(x'))))))$ </div> <div style="display: flex; align-items: flex-start;"> {-4} $\exists (n: \text{nat}): \text{while_termination_point?}(\text{b_ex}', \text{body}', x')(n)$ </div> <div style="display: flex; align-items: flex-start;"> {-5} $\text{OK?}(\text{b_ex}'(x'))$ </div> <div style="display: flex; align-items: flex-start;"> {-6} $\text{data}(\text{b_ex}'(x'))$ </div> <div style="display: flex; align-items: flex-start;"> {-7} $\text{OK?}(\text{e2s}(\text{b_ex}'(x')))$ </div> </div>	<hr style="border: 0.5px solid black;"/> <div style="display: flex; flex-direction: column; gap: 5px;"> <div style="display: flex; align-items: flex-start;"> {1} $1 + \min(\lambda (n: \text{nat}): \text{while_termination_point?}(\text{b_ex}', \text{body}', x')(1 + n)) - 1$ $=$ $\min(\text{while_termination_point?}(\text{b_ex}', \text{body}',$ $\quad \text{state}(\text{catch_continue}(\text{body}'(\text{state}(\text{b_ex}'(x'))))))$ </div> <div style="display: flex; align-items: flex-start;"> {2} $1 + \min(\lambda (n: \text{nat}): \text{while_termination_point?}(\text{b_ex}', \text{body}', x')(1 + n)) = 0$ </div> </div>
--	---

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `while_as_if_while.1.1.2.1.1.1.2.1.1`.

C Proof scripts

`while_as_if_while.1.1.2.1.1.1.2.1.2:`

<div style="display: flex; flex-direction: column; gap: 5px;"> <div style="display: flex; align-items: flex-start;"> {-1} $\exists (n: \text{nat}):$ </div> <div style="margin-left: 20px;"> $\text{while_termination_point?}(\text{b_ex}', \text{body}',$ $\text{state}(\text{catch_continue}(\text{body}'(\text{state}(\text{b_ex}'(x'))))))$ (n) </div> <div style="display: flex; align-items: flex-start;"> {-2} $\text{OK?}(\text{catch_continue}(\text{body}'(\text{state}(\text{b_ex}'(x'))))))$ </div> <div style="display: flex; align-items: flex-start;"> {-3} $\exists (n: \text{nat}): \text{while_termination_point?}(\text{b_ex}', \text{body}', x')(n)$ </div> <div style="display: flex; align-items: flex-start;"> {-4} $\text{OK?}(\text{b_ex}'(x'))$ </div> <div style="display: flex; align-items: flex-start;"> {-5} $\text{data}(\text{b_ex}'(x'))$ </div> <div style="display: flex; align-items: flex-start;"> {-6} $\text{OK?}(\text{e2s}(\text{b_ex}')(x'))$ </div> </div>	<hr style="border: 0.5px solid black;"/> <div style="display: flex; flex-direction: column; gap: 5px;"> <div style="display: flex; align-items: flex-start;"> {1} $(\lambda (n: \text{nat}): \text{while_termination_point?}(\text{b_ex}', \text{body}', x')(1+n)) =$ $\text{while_termination_point?}(\text{b_ex}', \text{body}',$ $\text{state}(\text{catch_continue}(\text{body}'(\text{state}(\text{b_ex}'(x'))))))$ </div> <div style="display: flex; align-items: flex-start;"> {2} $1 + \min(\lambda (n: \text{nat}): \text{while_termination_point?}(\text{b_ex}', \text{body}', x')(1+n)) - 1$ $=$ $\min(\text{while_termination_point?}(\text{b_ex}', \text{body}',$ $\text{state}(\text{catch_continue}(\text{body}'(\text{state}(\text{b_ex}'(x'))))))$ </div> <div style="display: flex; align-items: flex-start;"> {3} $1 + \min(\lambda (n: \text{nat}): \text{while_termination_point?}(\text{b_ex}', \text{body}', x')(1+n)) = 0$ </div> </div>
--	---

Hiding formulas: (2 3),

Applying `decompose-equality`,

Expanding the definition of `while_termination_point?`,

Expanding the definition of `iterate_while`,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `while_as_if_while.1.1.2.1.1.1.2.1.2`.

`while_as_if_while.1.1.2.1.1.1.2.2:`

<div style="display: flex; flex-direction: column; gap: 5px;"> <div style="display: flex; align-items: flex-start;"> {-1} $\text{while_termination_point?}(\text{b_ex}', \text{body}', x')(0)$ </div> <div style="display: flex; align-items: flex-start;"> {-2} $\exists (n: \text{nat}):$ </div> <div style="margin-left: 20px;"> $\text{while_termination_point?}(\text{b_ex}', \text{body}',$ $\text{state}(\text{catch_continue}(\text{body}'(\text{state}(\text{b_ex}'(x'))))))$ (n) </div> <div style="display: flex; align-items: flex-start;"> {-3} $\text{OK?}(\text{catch_continue}(\text{body}'(\text{state}(\text{b_ex}'(x'))))))$ </div> <div style="display: flex; align-items: flex-start;"> {-4} $\exists (n: \text{nat}): \text{while_termination_point?}(\text{b_ex}', \text{body}', x')(n)$ </div> <div style="display: flex; align-items: flex-start;"> {-5} $\text{OK?}(\text{b_ex}'(x'))$ </div> <div style="display: flex; align-items: flex-start;"> {-6} $\text{data}(\text{b_ex}'(x'))$ </div> <div style="display: flex; align-items: flex-start;"> {-7} $\text{OK?}(\text{e2s}(\text{b_ex}')(x'))$ </div> </div>	<hr style="border: 0.5px solid black;"/> <div style="display: flex; flex-direction: column; gap: 5px;"> <div style="display: flex; align-items: flex-start;"> {1} $\min(\text{while_termination_point?}(\text{b_ex}', \text{body}', x')) - 1 =$ $\min(\text{while_termination_point?}(\text{b_ex}', \text{body}',$ $\text{state}(\text{catch_continue}(\text{body}'(\text{state}(\text{b_ex}'(x'))))))$ </div> <div style="display: flex; align-items: flex-start;"> {2} $\min(\text{while_termination_point?}(\text{b_ex}', \text{body}', x')) = 0$ </div> </div>
--	---

Hiding formulas: 1,

Rewriting using `min_def`, matching in `*`,

Expanding the definition of `minimum?`,

which is trivially true.

This completes the proof of `while_as_if_while.1.1.2.1.1.1.2.2`.

while_as_if_while.1.1.2.1.1.2:

```

{-1} OK?(catch_continue(body'(state(b_ex'(x')))))
{-2} ∃ (n: nat): while_termination_point?(b_ex', body', x')(n)
{-3} OK?(b_ex'(x'))
{-4} data(b_ex'(x'))
{-5} OK?(e2s(b_ex')(x'))
-----
{1} ∃ (n: nat):
    while_termination_point?(b_ex', body',
        state(catch_continue(body'(state(b_ex'(x')))))
        (n))
{2} min(while_termination_point?(b_ex', body', x')) = 0
{3} catch_break(CASES iterate_while(min [nat](while_termination_point?(b_ex', body', x') - 1,
    b_ex', body')
    (state(catch_continue(body'(state(b_ex'(x'))))) OF
    OK(state_1): e2s(b_ex')(state_1)
    ELSE CASES catch_continue(body'(state(b_ex'(x')))) OF
    OK(state_1):
        iterate_while(min [nat](while_termination_point?(b_ex', body', x') - 1,
            b_ex', body')
            (state_1)
        ELSE catch_continue(body'(state(b_ex'(x'))))
    ENDCASES
    ENDCASES)
=
IF ∃ (n: nat):
    while_termination_point?(b_ex', body',
        state(catch_continue(body'(state(b_ex'(x')))))
        (n))
    THEN catch_break(catch_break(CASES iterate_while(min [nat]
        (while_termination_point?
            (b_ex',
                body',
                state
                (catch_continue
                    (body'
                        (state
                            (b_ex'(x'))))))),
            b_ex', body')
        (state(catch_continue
            (body'
                (state(b_ex'(x'))))) OF
            OK(state_1): e2s(b_ex')(state_1)
            ELSE iterate_while(min [nat]
                (while_termination_point?
                    (b_ex',
                        body',
                        state
                        (catch_continue
                            (body'
                                (state
                                    (b_ex'(x'))))))),
                    b_ex', body')
                (state(catch_continue
                    (body'
                        (state(b_ex'(x')))))
                    ENDCASES))
        ELSE catch_break(Hang)
    ENDIF

```

C Proof scripts

Hiding formulas: 3,
 Repeatedly Skolemizing and flattening,
 Case splitting on $n! = 0$,
 we get 2 subgoals:

`while_as_if_while.1.1.2.1.1.2.1:`

<pre>{-1} n' = 0 {-2} n' ≥ 0 {-3} OK?(catch_continue(body'(state(b_ex'(x'))))) {-4} while_termination_point?(b_ex', body', x')(n') {-5} OK?(b_ex'(x')) {-6} data(b_ex'(x')) {-7} OK?(e2s(b_ex')(x'))</pre>	<hr style="border: 0.5px solid black;"/> <pre>{1} ∃ (n: nat): while_termination_point?(b_ex', body', state(catch_continue(body'(state(b_ex'(x'))))) (n)) {2} min(while_termination_point?(b_ex', body', x')) = 0</pre>
---	--

Replacing using formula -1,
 Rewriting using `min_def`, matching in *,
 Expanding the definition of `minimum?`,
 which is trivially true.
 This completes the proof of `while_as_if_while.1.1.2.1.1.2.1`.
`while_as_if_while.1.1.2.1.1.2.2:`

<pre>{-1} n' ≥ 0 {-2} OK?(catch_continue(body'(state(b_ex'(x'))))) {-3} while_termination_point?(b_ex', body', x')(n') {-4} OK?(b_ex'(x')) {-5} data(b_ex'(x')) {-6} OK?(e2s(b_ex')(x'))</pre>	<hr style="border: 0.5px solid black;"/> <pre>{1} n' = 0 {2} ∃ (n: nat): while_termination_point?(b_ex', body', state(catch_continue(body'(state(b_ex'(x'))))) (n)) {3} min(while_termination_point?(b_ex', body', x')) = 0</pre>
--	--

Instantiating the top quantifier in 2 with the terms: $n' - 1$,
 we get 2 subgoals:

`while_as_if_while.1.1.2.1.1.2.2.1:`

<pre>{-1} n' ≥ 0 {-2} OK?(catch_continue(body'(state(b_ex'(x'))))) {-3} while_termination_point?(b_ex', body', x')(n') {-4} OK?(b_ex'(x')) {-5} data(b_ex'(x')) {-6} OK?(e2s(b_ex')(x'))</pre>	<hr style="border: 0.5px solid black;"/> <pre>{1} n' = 0 {2} while_termination_point?(b_ex', body', state(catch_continue(body'(state(b_ex'(x'))))) (n' - 1)) {3} min(while_termination_point?(b_ex', body', x')) = 0</pre>
--	---

Hiding formulas: 3,

Expanding the definition of while_termination_point?,

Expanding the definition of iterate_while,

Applying disjunctive simplification to flatten sequent,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of while_as_if_while.1.1.2.1.1.2.2.1.

while_as_if_while.1.1.2.1.1.2.2.2:

{-1}	$n' \geq 0$
{-2}	$\text{OK?}(\text{catch_continue}(\text{body}'(\text{state}(\text{b_ex}'(x')))))$
{-3}	$\text{while_termination_point?}(\text{b_ex}', \text{body}', x')(n')$
{-4}	$\text{OK?}(\text{b_ex}'(x'))$
{-5}	$\text{data}(\text{b_ex}'(x'))$
{-6}	$\text{OK?}(\text{e2s}(\text{b_ex}'(x')))$
{1}	$n' - 1 \geq 0$
{2}	$n' = 0$
{3}	$\text{min}(\text{while_termination_point?}(\text{b_ex}', \text{body}', x')) = 0$

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of while_as_if_while.1.1.2.1.1.2.2.2.

while_as_if_while.1.1.2.1.2:

```

{-1}  ∃ (n: nat): while_termination_point?(b_ex', body', x')(n)
{-2}  OK?(b_ex'(x'))
{-3}  data(b_ex'(x'))
{-4}  OK?(e2s(b_ex')(x'))
-----
{1}   OK?(catch_continue(body'(state(b_ex'(x')))))
{2}   min(while_termination_point?(b_ex', body', x') = 0
{3}   catch_break(CASES CASES catch_continue(body'(state(b_ex'(x')))) OF
      OK(state_1):
        iterate_while(min [nat](while_termination_point?(b_ex', body', x') -
          b_ex', body')
          (state_1)
        ELSE catch_continue(body'(state(b_ex'(x'))))
      ENDCASES OF
      OK(state_1): e2s(b_ex')(state_1)
      ELSE CASES catch_continue(body'(state(b_ex'(x')))) OF
        OK(state_1):
          iterate_while(min [nat](while_termination_point?(b_ex', body', x') -
            b_ex', body')
            (state_1)
          ELSE catch_continue(body'(state(b_ex'(x'))))
        ENDCASES
      ENDCASES)
=
catch_break(CASES catch_continue(body'(state(b_ex'(x')))) OF
  OK(state_1):
    IF ∃ (n: nat): while_termination_point?(b_ex', body', state_1)(n)
      THEN catch_break(CASES iterate_while(min [nat]
        (while_termination_point?
          (b_ex', body', state_1)
          b_ex', body')
        (state_1) OF
          OK(state_1): e2s(b_ex')(state_1)
          ELSE iterate_while(min [nat]
            (while_termination_point?
              (b_ex', body', state_1)
              b_ex',
              body')
            (state_1)
          ENDCASES)
        ELSE Hang
      ENDIF
    ELSE catch_continue(body'(state(b_ex'(x'))))
  ENDCASES)

```

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of while_as_if_while.1.1.2.1.2.

while_as_if_while.1.1.2.2:

<pre> {-1} ∃ (n: nat): while_termination_point?(b_ex', body', x')(n) {-2} OK?(b_ex'(x')) {-3} data(b_ex'(x')) </pre>	<hr/> <pre> {1} min(while_termination_point?(b_ex', body', x')) = 0 {2} OK?(e2s(b_ex')(x')) {3} catch_break(lift(e2s(b_ex')) (lift(iterate_while(min [nat](while_termination_point?(b_ex', body', x') - 1, b_ex', body')) (catch_continue(e2s(b_ex')(x')))))) = catch_break(lift(λ (s: State): IF ∃ (n: nat): while_termination_point?(b_ex', body', s)(n) THEN catch_break(lift(e2s(b_ex')) (iterate_while(min [nat] (while_termination_point? (b_ex', body', s), b_ex', body') (s)))) ELSE Hang ENDIF) (catch_continue(body'(state(b_ex'(x')))))) </pre>
---	---

Expanding the definition of e2s,
which is trivially true.

This completes the proof of while_as_if_while.1.1.2.2.

while_as_if_while.1.2:

<pre> {1} ∃ (n: nat): while_termination_point?(b_ex', body', x')(n) {2} Hang?((b_ex' ## (λ (b: bool): IF b THEN (body' ## catch_continue ## (λ (s: State): IF ∃ (n: nat): while_termination_point?(b_ex', body', s)(n) THEN (iterate_while(min [nat] (while_termination_point?(b_ex', body', s)), b_ex', body') ## e2s(b_ex') ## catch_break) (s) ELSE Hang ENDIF)) ## catch_break ELSE skip ENDIF)) (x')) </pre>	<hr/>
---	-------

Copying formula number: 1

Copying formula number: 1

Instantiating the top quantifier in 1 with the terms: 0,
Instantiating the top quantifier in 2 with the terms: 1,
Expanding the definition of while_termination_point?,
Expanding the definition of iterate_while,
Expanding the definition of iterate_while,
Expanding the definition of skip,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Case splitting on EXISTS (n: nat): while_termination_point?(b_ex!1, body!1, state((e2s(b_ex!1) ##
body!1 ## catch_continue ## (LAMBDA (s): OK(s))) (x!1))) (n),
we get 2 subgoals:
while_as_if_while.1.2.1:

<pre> {-1} ∃ (n: nat): while_termination_point?(b_ex', body', state((e2s(b_ex') ## body' ## catch_continue ## (λ (s): OK(s))) (x'))) (n) {-2} OK?(b_ex'(x')) {-3} data(b_ex'(x')) {-4} OK?((e2s(b_ex') ## body' ## catch_continue ## (λ (s): OK(s)))(x')) {-5} OK?(b_ex'(state((e2s(b_ex') ## body' ## catch_continue ## (λ (s): OK(s))) (x')))) {-6} data(b_ex'(state((e2s(b_ex') ## body' ## catch_continue ## (λ (s): OK(s))) (x')))) </pre>	<hr style="border: 0.5px solid black;"/> <pre> {1} ∃ (n: nat): while_termination_point?(b_ex', body', x')(n) {2} Hang?((b_ex' ## (λ (b: bool): IF b THEN (body' ## catch_continue ## (λ (s: State): IF ∃ (n: nat): while_termination_point?(b_ex', body', s)(n) THEN (iterate_while(min[nat] (while_termination_point?(b_ex' b_ex', body') ## e2s(b_ex') ## catch_break) (s) ELSE Hang ENDIF)) ## catch_break ELSE λ (s: State): OK(s) ENDIF)) (x')) </pre>
--	--

Repeatedly Skolemizing and flattening,
Instantiating the top quantifier in 1 with the terms: (n' + 1),
Expanding the definition of while_termination_point?,
Expanding the definition of iterate_while,
Applying disjunctive simplification to flatten sequent,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of **while_as_if_while.1.2.1**.

while_as_if_while.1.2.2:

<pre> {-1} OK?(b_ex'(x')) {-2} data(b_ex'(x')) {-3} OK?((e2s(b_ex') ## body' ## catch_continue ## (λ (s): OK(s)))(x')) {-4} OK?(b_ex'(state((e2s(b_ex') ## body' ## catch_continue ## (λ (s): OK(s)) (x')))) {-5} data(b_ex'(state((e2s(b_ex') ## body' ## catch_continue ## (λ (s): OK(s)) (x')))) </pre>	<pre> {1} ∃ (n: nat): while_termination_point?(b_ex', body', state((e2s(b_ex') ## body' ## catch_continue ## (λ (s): OK(s)) (x'))) (n) {2} ∃ (n: nat): while_termination_point?(b_ex', body', x')(n) {3} Hang?((b_ex' ## (λ (b: bool): IF b THEN (body' ## catch_continue ## (λ (s: State): IF ∃ (n: nat): while_termination_point?(b_ex', body', s)(n) THEN (iterate_while(min[nat] (while_termination_point?(b_ex', body', s)), b_ex', body') ## e2s(b_ex') ## catch_break) (s) ELSE Hang ENDIF)) ## catch_break ELSE λ (s: State): OK(s) ENDIF)) (x')) </pre>
--	--

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of while_as_if_while.1.2.2.

while_as_if_while.2:

<pre> {1} ∃ (b: bool): b ⊃ (∃ (s: State): (∃ (n: nat): while_termination_point?(b_ex', body', s)(n)) ⊃ nonempty?[nat](while_termination_point?(b_ex', body', s))) </pre>	
--	--

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of while_as_if_while.2.

while_as_if_while.3:

$$\{1\} \quad \forall (s: \text{State}):$$

$$\quad (\exists (n: \text{nat}): \text{while_termination_point?}(b_ex', \text{body}', s)(n)) \supset$$

$$\quad \text{nonempty?}[\text{nat}](\text{while_termination_point?}(b_ex', \text{body}', s))$$

Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of `while_as_if_while.3`.
 Q.E.D.

C.107.7 IterationStatements1.iterate_do_as_while

Terse proof for `iterate_do_as_while`.

`iterate_do_as_while`:

$$\{1\} \quad \forall (b_ex: [\text{State} \rightarrow \text{ExprResult}[\text{State}, \text{bool}]], \text{body}: [\text{State} \rightarrow \text{StmtResult}[\text{State}]],$$

$$\quad n: \text{nat}):$$

$$\quad \text{iterate_do}(n, \text{body}, b_ex) = (\text{body} \## \text{catch_continue} \## \text{iterate_while}(n, b_ex, \text{body}))$$

Inducting on n on formula 1,
 we get 2 subgoals:

`iterate_do_as_while.1`:

$$\{1\} \quad \forall (b_ex: [\text{State} \rightarrow \text{ExprResult}[\text{State}, \text{bool}]], \text{body}: [\text{State} \rightarrow \text{StmtResult}[\text{State}]]):$$

$$\quad \text{iterate_do}(0, \text{body}, b_ex) = (\text{body} \## \text{catch_continue} \## \text{iterate_while}(0, b_ex, \text{body}))$$

Repeatedly Skolemizing and flattening,
 Applying decompose-equality,
 Trying repeated skolemization, instantiation, and if-lifting,
 Applying decompose-equality,
 This completes the proof of `iterate_do_as_while.1`.

`iterate_do_as_while.2`:

$$\{1\} \quad \forall j:$$

$$\quad (\forall (b_ex: [\text{State} \rightarrow \text{ExprResult}[\text{State}, \text{bool}]], \text{body}: [\text{State} \rightarrow \text{StmtResult}[\text{State}]]):$$

$$\quad \quad \text{iterate_do}(j, \text{body}, b_ex) =$$

$$\quad \quad (\text{body} \## \text{catch_continue} \## \text{iterate_while}(j, b_ex, \text{body})))$$

$$\quad \supset$$

$$\quad (\forall (b_ex: [\text{State} \rightarrow \text{ExprResult}[\text{State}, \text{bool}]], \text{body}: [\text{State} \rightarrow \text{StmtResult}[\text{State}]]):$$

$$\quad \quad \text{iterate_do}(j + 1, \text{body}, b_ex) =$$

$$\quad \quad (\text{body} \## \text{catch_continue} \## \text{iterate_while}(j + 1, b_ex, \text{body})))$$

Repeatedly Skolemizing and flattening,
 Expanding the definition of `iterate_do`,
 Expanding the definition of `iterate_while`,
 Applying decompose-equality,
 Instantiating quantified variables,
 Replacing using formula -2,

Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of `iterate_do_as_while.2`.
 Q.E.D.

C.107.8 IterationStatements1.do_termination_point?_TCC1

Terse proof for `do_termination_point?_TCC1`.

`do_termination_point?_TCC1`:

$\{1\} \quad \forall (\text{body}: [\text{State} \rightarrow \text{StmtResult}[\text{State}]], \text{b_ex}: [\text{State} \rightarrow \text{ExprResult}[\text{State}, \text{bool}]],$ $s: \text{State}, n: \text{nat}, \text{res}: \text{StmtResult}[\text{State}]):$ $\text{OK?}(\text{res}) \wedge \text{res} = \text{iterate_do}(n, \text{body}, \text{b_ex})(s) \supset$ $\text{OK?}[\text{State}](\text{res}) \vee$ $\text{Break?}[\text{State}](\text{res}) \vee$ $\text{Continue?}[\text{State}](\text{res}) \vee$ $\text{Return?}[\text{State}](\text{res}) \vee$ $\text{Switch?}[\text{State}](\text{res}) \vee \text{Default?}[\text{State}](\text{res}) \vee \text{Exception?}[\text{State}](\text{res})$

Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of `do_termination_point?_TCC1`.
 Q.E.D.

C.107.9 IterationStatements1.termination_point_do_as_while_TCC1

Terse proof for `termination_point_do_as_while_TCC1`.

`termination_point_do_as_while_TCC1`:

$\{1\} \quad \forall (\text{body}: [\text{State} \rightarrow \text{StmtResult}[\text{State}]], s: \text{State}):$ $\text{OK?}((\text{body} \ \#\# \ \text{catch_continue})(s)) \supset$ $\text{OK?}[\text{State}]((\#\# [\text{State}] (\text{body}, \text{catch_continue}[\text{State}])(s)) \vee$ $\text{Break?}[\text{State}]((\#\# [\text{State}] (\text{body}, \text{catch_continue}[\text{State}])(s)) \vee$ $\text{Continue?}[\text{State}]((\#\# [\text{State}] (\text{body}, \text{catch_continue}[\text{State}])(s)) \vee$ $\text{Return?}[\text{State}]((\#\# [\text{State}] (\text{body}, \text{catch_continue}[\text{State}])(s)) \vee$ $\text{Switch?}[\text{State}]((\#\# [\text{State}] (\text{body}, \text{catch_continue}[\text{State}])(s)) \vee$ $\text{Default?}[\text{State}]((\#\# [\text{State}] (\text{body}, \text{catch_continue}[\text{State}])(s)) \vee$ $\text{Exception?}[\text{State}]((\#\# [\text{State}] (\text{body}, \text{catch_continue}[\text{State}])(s))$

Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of `termination_point_do_as_while_TCC1`.
 Q.E.D.

C.107.10 IterationStatements1.termination_point_do_as_while

Terse proof for `termination_point_do_as_while`.

termination_point_do_as_while:

$$\{1\} \quad \forall (b_ex: [State \rightarrow ExprResult [State, bool]], \text{body}: [State \rightarrow StmtResult [State]]), \\ s: State): \\ OK?((\text{body} \ \#\# \ \text{catch_continue})(s)) \supset \\ \text{do_termination_point?}(\text{body}, b_ex, s) = \\ \text{while_termination_point?}(b_ex, \text{body}, \text{state}((\text{body} \ \#\# \ \text{catch_continue})(s)))$$

Repeatedly Skolemizing and flattening,
 Applying decompose-equality,
 Expanding the definition of do_termination_point?,
 Expanding the definition of while_termination_point?,
 Rewriting using iterate_do_as_while, matching in *,
 Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of termination_point_do_as_while.
 Q.E.D.

C.107.11 IterationStatements1.do_while_TCC1

Terse proof for do_while_TCC1.

do_while_TCC1:

$$\{1\} \quad \forall (\text{body}: [State \rightarrow StmtResult [State]], b_ex: [State \rightarrow ExprResult [State, bool]], \\ s: State): \\ (\exists (n: \text{nat}): \text{do_termination_point?}(\text{body}, b_ex, s)(n)) \supset \\ \text{nonempty?}[\text{nat}](\text{do_termination_point?}(\text{body}, b_ex, s))$$

Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of do_while_TCC1.
 Q.E.D.

C.107.12 IterationStatements1.do_as_while

Terse proof for do_as_while.

do_as_while:

$$\{1\} \quad \forall (b_ex: [State \rightarrow ExprResult [State, bool]], \text{body}: [State \rightarrow StmtResult [State]]): \\ \text{do_while}(\text{body}, b_ex) = (\text{body} \ \#\# \ \text{catch_continue} \ \#\# \ \text{while}(b_ex, \text{body}) \ \#\# \ \text{catch_break})$$

Repeatedly Skolemizing and flattening,
 Expanding the definition of do_while,
 Expanding the definition of while,
 Applying decompose-equality,
 we get 3 subgoals:

do_as_while.1:

```

{1} IF  $\exists (n: \text{nat}): \text{do\_termination\_point?}(\text{body}', \text{b\_ex}', x')(n)$ 
    THEN  $(\text{iterate\_do}(\text{min}[\text{nat}](\text{do\_termination\_point?}(\text{body}', \text{b\_ex}', x')), \text{body}', \text{b\_ex}')$ 
         $\text{## e2s}(\text{b\_ex}')$ 
         $\text{## catch\_break}$ 
         $(x')$ 
    ELSE Hang
    ENDIF
    =
     $((\text{body}' \text{## catch\_continue} \text{##}$ 
         $(\lambda (s: \text{State}):$ 
            IF  $\exists (n: \text{nat}): \text{while\_termination\_point?}(\text{b\_ex}', \text{body}', s)(n)$ 
                THEN  $(\text{iterate\_while}(\text{min}[\text{nat}](\text{while\_termination\_point?}(\text{b\_ex}', \text{body}', s)),$ 
                     $\text{b\_ex}', \text{body}')$ 
                     $\text{## e2s}(\text{b\_ex}')$ 
                     $\text{## catch\_break}$ 
                     $(s)$ 
                ELSE Hang
                ENDIF))
         $\text{## catch\_break}$ 
         $(x')$ 

```

Installing automatic rewrites from: (iterate_do_as_while! termination_point_do_as_while!)

Case splitting on $\text{OK?}((\text{body}'1 \text{## catch_continue})(x'1))$,

we get 2 subgoals:

do_as_while.1.1:

```

{-1} OK? $((\text{body}' \text{## catch\_continue})(x'))$ 
{1} IF  $\exists (n: \text{nat}): \text{do\_termination\_point?}(\text{body}', \text{b\_ex}', x')(n)$ 
    THEN  $(\text{iterate\_do}(\text{min}[\text{nat}](\text{do\_termination\_point?}(\text{body}', \text{b\_ex}', x')), \text{body}', \text{b\_ex}')$ 
         $\text{## e2s}(\text{b\_ex}')$ 
         $\text{## catch\_break}$ 
         $(x')$ 
    ELSE Hang
    ENDIF
    =
     $((\text{body}' \text{## catch\_continue} \text{##}$ 
         $(\lambda (s: \text{State}):$ 
            IF  $\exists (n: \text{nat}): \text{while\_termination\_point?}(\text{b\_ex}', \text{body}', s)(n)$ 
                THEN  $(\text{iterate\_while}(\text{min}[\text{nat}](\text{while\_termination\_point?}(\text{b\_ex}', \text{body}', s)),$ 
                     $\text{b\_ex}', \text{body}')$ 
                     $\text{## e2s}(\text{b\_ex}')$ 
                     $\text{## catch\_break}$ 
                     $(s)$ 
                ELSE Hang
                ENDIF))
         $\text{## catch\_break}$ 
         $(x')$ 

```

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of do_as_while.1.1.

do_as_while.1.2:

<pre> {1} OK?((body' ## catch_continue)(x')) {2} IF ∃ (n: nat): do_termination_point?(body', b_ex', x')(n) THEN (iterate_do(min[nat](do_termination_point?(body', b_ex', x')), body', b_ex') ## e2s(b_ex') ## catch_break) (x') ELSE Hang ENDIF = ((body' ## catch_continue ## (λ (s: State): IF ∃ (n: nat): while_termination_point?(b_ex', body', s)(n) THEN (iterate_while(min[nat](while_termination_point?(b_ex', body', s)), b_ex', body') ## e2s(b_ex') ## catch_break) (s) ELSE Hang ENDIF)) ## catch_break) (x') </pre>
--

Trying repeated skolemization, instantiation, and if-lifting,
we get 2 subgoals:

do_as_while.1.2.1:

<pre> {-1} Break?(body'(x')) {1} ∃ (n: nat): do_termination_point?(body', b_ex', x')(n) </pre>
--

Instantiating the top quantifier in 1 with the terms: 0,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of do_as_while.1.2.1.

do_as_while.1.2.2:

<pre> {1} Continue?(body'(x')) {2} OK?(body'(x')) {3} ∃ (n: nat): do_termination_point?(body', b_ex', x')(n) {4} Hang?(body'(x')) </pre>
--

Instantiating the top quantifier in 3 with the terms: 0,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of do_as_while.1.2.2.

do_as_while.2:

<pre> {1} ∀ (s: State): (∃ (n: nat): while_termination_point?(b_ex', body', s)(n)) ⊃ nonempty?[nat](while_termination_point?(b_ex', body', s)) </pre>

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of do_as_while.2.

do_as_while.3:

{1} $\forall (s: \text{State}):$
 $(\exists (n: \text{nat}): \text{do_termination_point?}(\text{body}', \text{b_ex}', s)(n)) \supset$
 $\text{nonempty?}[\text{nat}](\text{do_termination_point?}(\text{body}', \text{b_ex}', s))$

Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of do_as_while.3.
 Q.E.D.

C.107.13 IterationStatements1.do_while_unroll

Terse proof for do_while_unroll.

do_while_unroll:

{1} $\forall (\text{b_ex}: [\text{State} \rightarrow \text{ExprResult}[\text{State}, \text{bool}]], \text{body}: [\text{State} \rightarrow \text{StmtResult}[\text{State}]],$
 $\text{max}: \text{nat}):$
 $\text{do_while}(\text{max}, \text{body}, \text{b_ex}) =$
 $(\text{body} \ \#\# \ \text{catch_continue} \ \#\# \ \text{while}(\text{max}, \text{b_ex}, \text{body}) \ \#\# \ \text{catch_break})$

Repeatedly Skolemizing and flattening,
 Expanding the definition of do_while,
 Expanding the definition of while,
 Rewriting using do_as_while, matching in *,
 This completes the proof of do_while_unroll.
 Q.E.D.

C.108 Proofs for IterationStatements2 (statements.pvs)

C.108.1 IterationStatements2.iterate_for_TCC1

Terse proof for iterate_for_TCC1.

iterate_for_TCC1:

{1} $\forall (n: \text{nat}, s: \text{State}): \neg n = 0 \supset n - 1 \geq 0$

Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of iterate_for_TCC1.
 Q.E.D.

C.108.2 IterationStatements2.iterate_for_TCC2

Terse proof for iterate_for_TCC2.

iterate_for_TCC2:

{1} $\forall (n: \text{nat}, s: \text{State}): \neg n = 0 \supset n - 1 < n$

Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of iterate_for_TCC2.
 Q.E.D.

C.108.3 IterationStatements2.iterate_for_as_while_TCC1

Terse proof for `iterate_for_as_while_TCC1`.

`iterate_for_as_while_TCC1:`

$\{1\} \quad \forall (n: \text{nat}): n > 0 \supset n - 1 \geq 0$

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `iterate_for_as_while_TCC1`.

Q.E.D.

C.108.4 IterationStatements2.iterate_for_as_while

Terse proof for `iterate_for_as_while`.

`iterate_for_as_while:`

$\{1\} \quad \forall (b_ex: [\text{State} \rightarrow \text{ExprResult}[\text{State}, \text{bool}]], \text{body}: [\text{State} \rightarrow \text{StmtResult}[\text{State}]],$ $\quad \text{expr}: [\text{State} \rightarrow \text{ExprResult}[\text{State}, \text{ExprData}]], n: \text{nat}):$ $\quad n > 0 \supset$ $\quad \text{iterate_for}(n, b_ex, \text{expr}, \text{body}) =$ $\quad (\text{e2s}(b_ex) \## \text{body} \## \text{catch_continue} \## \text{iterate_while}(n - 1, \text{expr} \## b_ex, \text{body})$ $\quad \## \text{e2s}(\text{expr}))$
--

Inducting on n on formula 1,

we get 3 subgoals:

`iterate_for_as_while.1:`

$\{1\} \quad \forall (b_ex: [\text{State} \rightarrow \text{ExprResult}[\text{State}, \text{bool}]], \text{body}: [\text{State} \rightarrow \text{StmtResult}[\text{State}]],$ $\quad \text{expr}: [\text{State} \rightarrow \text{ExprResult}[\text{State}, \text{ExprData}]]):$ $\quad 0 > 0 \supset$ $\quad \text{iterate_for}(0, b_ex, \text{expr}, \text{body}) =$ $\quad (\text{e2s}(b_ex) \## \text{body} \## \text{catch_continue} \## \text{iterate_while}(0 - 1, \text{expr} \## b_ex, \text{body}) \##$ $\quad \text{e2s}(\text{expr}))$

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `iterate_for_as_while.1`.

iterate_for_as_while.2:

<pre> {1} ∀ j: (∀ (b_ex: [State → ExprResult[State, bool]], body: [State → StmtRe- sult[State]], expr: [State → ExprResult[State, ExprData]]): j > 0 ⊃ iterate_for(j, b_ex, expr, body) = (e2s(b_ex) ## body ## catch_continue ## iterate_while(j - 1, expr ## b_ex, body) ## e2s(expr))) ⊃ (∀ (b_ex: [State → ExprResult[State, bool]], body: [State → StmtRe- sult[State]], expr: [State → ExprResult[State, ExprData]]): j + 1 > 0 ⊃ iterate_for(j + 1, b_ex, expr, body) = (e2s(b_ex) ## body ## catch_continue ## iterate_while(j + 1 - 1, expr ## b_ex, body) ## e2s(expr))) </pre>
--

Repeatedly Skolemizing and flattening,

Instantiating quantified variables,

Applying decompose-equality,

Case splitting on $j!1 = 0$,

we get 2 subgoals:

iterate_for_as_while.2.1:

<pre> {-1} j' = 0 {-2} j' ≥ 0 {-3} j' > 0 ⊃ iterate_for(j', b_ex', expr', body') = (e2s(b_ex') ## body' ## catch_continue ## iterate_while(j' - 1, expr' ## b_ex', body') ## e2s(expr')) {-4} 1 + j' > 0 </pre>
<hr/> <pre> {1} iterate_for(1 + j', b_ex', expr', body')(x') = (e2s(b_ex') ## body' ## catch_continue ## iterate_while(j', expr' ## b_ex', body') ## e2s(expr')) (x') </pre>

Replacing using formula -1,

Keeping (1) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `iterate_for_as_while.2.1`.

iterate_for_as_while.2.2:

<pre> {-1} j' ≥ 0 {-2} j' > 0 ⊃ iterate_for(j', b_ex', expr', body') = (e2s(b_ex') ## body' ## catch_continue ## iterate_while(j' - 1, expr' ## b_ex', body') ## e2s(expr')) {-3} 1 + j' > 0 </pre>	<pre> {1} j' = 0 {2} iterate_for(1 + j', b_ex', expr', body')(x') = (e2s(b_ex') ## body' ## catch_continue ## iterate_while(j', expr' ## b_ex', body') ## e2s(expr')) (x') </pre>
--	--

Simplifying, rewriting, and recording with decision procedures,

Expanding the definition of iterate_for,

Expanding the definition of iterate_while,

Replacing using formula -2,

Keeping (2) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of iterate_for_as_while.2.2.

iterate_for_as_while.3:

<pre> {1} ∀ (n: nat): n > 0 ⊃ n - 1 ≥ 0 {2} ∀ (b_ex: [State → ExprResult[State, bool]], body: [State → StmtResult[State]], expr: [State → ExprResult[State, ExprData]]): n' > 0 ⊃ iterate_for(n', b_ex, expr, body) = (e2s(b_ex) ## body ## catch_continue ## iterate_while(n' - 1, expr ## b_ex, body) ## e2s(expr)) </pre>	
--	--

Hiding formulas: 2,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of iterate_for_as_while.3.

Q.E.D.

C.108.5 IterationStatements2.for_termination_point?_TCC1

Terse proof for for_termination_point?_TCC1.

for_termination_point?_TCC1:

<pre> {1} ∀ (b_ex: [State → ExprResult[State, bool]], expr: [State → ExprResult[State, ExprData]], body: [State → StmtRe- sult[State]], s: State, n: nat, res: StmtResult[State]): OK?(res) ∧ res = iterate_for(n, b_ex, expr, body)(s) ⊃ OK?[State](res) ∨ Break?[State](res) ∨ Continue?[State](res) ∨ Return?[State](res) ∨ Switch?[State](res) ∨ Default?[State](res) ∨ Exception?[State](res) </pre>	
--	--

Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of `for_termination_point?_TCC1`.
 Q.E.D.

C.108.6 IterationStatements2.termination_point_for_as_while_TCC1

Terse proof for `termination_point_for_as_while_TCC1`.

`termination_point_for_as_while_TCC1`:

```

{1}  ∃ (b_ex: [State → ExprResult[State, bool]], body: [State → StmtResult[State]],
      s: State, n: nat):
  n > 0 ∧ OK?((e2s(b_ex) ## body ## catch_continue)(s)) ⊃
  OK?[State]
    ((##[State]
      (##[State](e2s[State, boolean](b_ex), body), catch_continue[State]))
      (s))
  ∨
  Break?[State]
    ((##[State]
      (##[State](e2s[State, boolean](b_ex), body), catch_continue[State]))
      (s))
  ∨
  Continue?[State]
    ((##[State]
      (##[State](e2s[State, boolean](b_ex), body), catch_continue[State]))
      (s))
  ∨
  Return?[State]
    ((##[State]
      (##[State](e2s[State, boolean](b_ex), body), catch_continue[State]))
      (s))
  ∨
  Switch?[State]
    ((##[State]
      (##[State](e2s[State, boolean](b_ex), body), catch_continue[State]))
      (s))
  ∨
  Default?[State]
    ((##[State]
      (##[State](e2s[State, boolean](b_ex), body),
      catch_continue[State]))
      (s))
  ∨
  Exception?[State]
    ((##[State]
      (##[State](e2s[State, boolean](b_ex), body),
      catch_continue[State]))
      (s))

```

Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of `termination_point_for_as_while_TCC1`.

Q.E.D.

C.108.7 IterationStatements2.termination_point_for_as_while

Terse proof for `termination_point_for_as_while`.

`termination_point_for_as_while`:

$$\{1\} \quad \forall (b_ex: [State \rightarrow ExprResult[State, bool]], body: [State \rightarrow StmtResult[State]], \\ \text{expr: [State} \rightarrow ExprResult[State, ExprData]], s: State, n: nat): \\ n > 0 \wedge OK?((e2s(b_ex) \#\# body \#\# catch_continue)(s)) \supset \\ \text{for_termination_point?}(b_ex, \text{expr}, body, s)(n) = \\ \text{while_termination_point?}(\text{expr} \#\# b_ex, body, \\ \text{state}((e2s(b_ex) \#\# body \#\# catch_continue)(s))) \\ (n - 1)$$

Repeatedly Skolemizing and flattening,

Expanding the definition of `for_termination_point?`,

Expanding the definition of `while_termination_point?`,

Rewriting using `iterate_for_as_while`, matching in `*`,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `termination_point_for_as_while`.

Q.E.D.

C.108.8

IterationStatements2.min_termination_point_for_as_while_TCC1

Terse proof for `min_termination_point_for_as_while_TCC1`.

`min_termination_point_for_as_while_TCC1`:

$$\{1\} \quad \forall (b_ex: [State \rightarrow ExprResult[State, bool]], body: [State \rightarrow StmtResult[State]], \\ \text{expr: [State} \rightarrow ExprResult[State, ExprData]], s: State): \\ (\exists (n: nat): \text{for_termination_point?}(b_ex, \text{expr}, body, s)(n)) \wedge \\ OK?[State]((e2s(b_ex) \#\# body \#\# catch_continue)(s)) \\ \supset \text{nonempty?}[nat](\text{for_termination_point?}(b_ex, \text{expr}, body, s))$$

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `min_termination_point_for_as_while_TCC1`.

Q.E.D.

C.108.9

IterationStatements2.min_termination_point_for_as_while_TCC2

Terse proof for `min_termination_point_for_as_while_TCC2`.

min_termination_point_for_as_while_TCC2:

```

{1}  ∀ (b_ex: [State → ExprResult[State, bool]], body: [State → StmtResult[State]],
      expr: [State → ExprResult[State, ExprData]], s: State):
(¬ min[nat](for_termination_point?(b_ex, expr, body, s)) = 0 ∧
  OK?[State]((e2s(b_ex) ## body ## catch_continue)(s)) ∧
  (∃ (n: nat): for_termination_point?(b_ex, expr, body, s)(n)))
⊃
OK?[State]
  ((##[State]
    (##[State](e2s[State, boolean](b_ex), body), catch_continue[State]))
    (s))
∨
Break?[State]
  ((##[State]
    (##[State](e2s[State, boolean](b_ex), body), catch_continue[State]))
    (s))
∨
Continue?[State]
  ((##[State]
    (##[State](e2s[State, boolean](b_ex), body), catch_continue[State]))
    (s))
∨
Return?[State]
  ((##[State]
    (##[State](e2s[State, boolean](b_ex), body), catch_continue[State]))
    (s))
∨
Switch?[State]
  ((##[State]
    (##[State](e2s[State, boolean](b_ex), body), catch_continue[State]))
    (s))
∨
Default?[State]
  ((##[State]
    (##[State](e2s[State, boolean](b_ex), body),
      catch_continue[State]))
    (s))
∨
Exception?[State]
  ((##[State]
    (##[State](e2s[State, boolean](b_ex), body),
      catch_continue[State]))
    (s))

```

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of min_termination_point_for_as_while_TCC2.

Q.E.D.

C.108.10**IterationStatements2.min_termination_point_for_as_while_TCC3**

Terse proof for min_termination_point_for_as_while_TCC3.

min_termination_point_for_as_while_TCC3:

<pre> {1} ∃ (b_ex: [State → ExprResult[State, bool]], body: [State → StmtResult[State]], expr: [State → ExprResult[State, ExprData]], s: State): (¬ min[nat](for_termination_point?(b_ex, expr, body, s)) = 0 ∧ OK?[State]((e2s(b_ex) ## body ## catch_continue)(s)) ∧ (∃ (n: nat): for_termination_point?(b_ex, expr, body, s)(n))) ⊃ nonempty?[nat] (while_termination_point?[State] (##[State, ExprData, boolean](expr, b_ex), body, state[State] ((##[State] (##[State](e2s[State, boolean](b_ex), body), catch_continue[State]))) (s)))) </pre>

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Repeatedly Skolemizing and flattening,

Case splitting on $n!1 = 0$,

we get 2 subgoals:

min_termination_point_for_as_while_TCC3.1:

<pre> {-1} n' = 0 {-2} n' ≥ 0 {-3} OK?[State]((e2s(b_ex') ## body' ## catch_continue)(s')) {-4} for_termination_point?(b_ex', expr', body', s')(n') </pre>	<pre> {1} min[nat](for_termination_point?(b_ex', expr', body', s')) = 0 {2} nonempty?[nat] (while_termination_point?[State] (##[State, ExprData, boolean](expr', b_ex'), body', state[State] ((##[State] (##[State](e2s[State, boolean](b_ex'), body'), catch_continue[State]))) (s')))) </pre>
--	---

Rewriting using min_def, matching in *,

Expanding the definition of minimum?,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of min_termination_point_for_as_while_TCC3.1.

min_termination_point_for_as_while_TCC3.2:

<pre>{-1} n' ≥ 0 {-2} OK?[State]((e2s(b_ex') ## body' ## catch_continue)(s')) {-3} for_termination_point?(b_ex', expr', body', s')(n')</pre>	<hr style="border: 0.5px solid black;"/> <pre>{1} n' = 0 {2} min[nat](for_termination_point?(b_ex', expr', body', s')) = 0 {3} nonempty?[nat] (while_termination_point?[State] (##[State, ExprData, boolean](expr', b_ex'), body', state[State] ((##[State] (##[State](e2s[State, boolean](b_ex'), body'), catch_continue[State])) (s'))))</pre>
---	--

Rewriting using termination_point_for_as_while, matching in *,

Hiding formulas: 2,

Expanding the definition of nonempty?,

Expanding the definition of empty?,

Expanding the definition of member,

Instantiating quantified variables,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of min_termination_point_for_as_while_TCC3.2.

Q.E.D.

C.108.11 IterationStatements2.min_termination_point_for_as_while

Terse proof for min_termination_point_for_as_while.

min_termination_point_for_as_while:

<pre>{1} ∃ (b_ex: [State → ExprResult[State, bool]], body: [State → StmtResult[State]], expr: [State → ExprResult[State, ExprData]], s: State): (∃ (n: nat): for_termination_point?(b_ex, expr, body, s)(n)) ∧ OK?[State]((e2s(b_ex) ## body ## catch_continue)(s)) ∧ ¬ min[nat](for_termination_point?(b_ex, expr, body, s)) = 0 ⊃ min[nat](for_termination_point?(b_ex, expr, body, s)) = min [nat](while_termination_point?(expr ## b_ex, body, state [State]((e2s(b_ex) ## body ## cat</pre>	
---	--

Skolemizing (with typepred on new Skolem constants),

Applying disjunctive simplification to flatten sequent,

Case splitting on while_termination_point?(expr!1 ## b_ex!1, body!1, state((e2s(b_ex!1) ## body!1 ## catch_continue) (s!1))) = (LAMBDA (n: nat): for_termination_point?(b_ex!1, expr!1, body!1, s!1)(n + 1)),

we get 2 subgoals:

min_termination_point_for_as_while.1:

{-1}	while_termination_point?(expr' ## b_ex', body', state((e2s(b_ex') ## body' ## catch_continue)(s')) = (λ (n: nat): for_termination_point?(b_ex', expr', body', s')(n + 1))
{-2}	∃ (n: nat): for_termination_point?(b_ex', expr', body', s')(n)
{-3}	OK?[State]((e2s(b_ex') ## body' ## catch_continue)(s'))
{1}	min [nat] (for_termination_point?(b_ex', expr', body', s')) = 0
{2}	min [nat] (for_termination_point?(b_ex', expr', body', s')) = min [nat] (while_termination_point?(expr' ## b_ex', body', state [State]((e2s(b_ex') ## bod

Replacing using formula -1,

Keeping (-2 1 2) and hiding *,

Using lemma min_plus1,

Simplifying, rewriting, and recording with decision procedures,

Hiding formulas: 2,

Rewriting using min_def, matching in *,

Expanding the definition of minimum?,

which is trivially true.

This completes the proof of min_termination_point_for_as_while.1.

min_termination_point_for_as_while.2:

{-1}	∃ (n: nat): for_termination_point?(b_ex', expr', body', s')(n)
{-2}	OK?[State]((e2s(b_ex') ## body' ## catch_continue)(s'))
{1}	while_termination_point?(expr' ## b_ex', body', state((e2s(b_ex') ## body' ## catch_continue)(s')) = (λ (n: nat): for_termination_point?(b_ex', expr', body', s')(n + 1))
{2}	min [nat] (for_termination_point?(b_ex', expr', body', s')) = 0
{3}	min [nat] (for_termination_point?(b_ex', expr', body', s')) = min [nat] (while_termination_point?(expr' ## b_ex', body', state [State]((e2s(b_ex') ## bod

Hiding formulas: 3,

Applying decompose-equality,

Rewriting using termination_point_for_as_while, matching in *,

This completes the proof of min_termination_point_for_as_while.2.

Q.E.D.

C.108.12

IterationStatements2.nonzero_min_for_termination_point_TCC1

Terse proof for nonzero_min_for_termination_point_TCC1.

nonzero_min_for_termination_point_TCC1:

{1}	∃ (b_ex: [State → ExprResult[State, bool]], body: [State → StmtResult[State]], expr: [State → ExprResult[State, ExprData]], s: State): (∃ (n: nat): for_termination_point?(b_ex, expr, body, s)(n)) ⊃ nonempty?[nat] (for_termination_point?(b_ex, expr, body, s))
-----	---

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of nonzero_min_for_termination_point_TCC1.

Q.E.D.

C.108.13 IterationStatements2.nonzero_min_for_termination_point

Terse proof for nonzero_min_for_termination_point.

nonzero_min_for_termination_point:

$$\frac{\{1\} \quad \forall (b_ex: [State \rightarrow ExprResult[State, bool]], body: [State \rightarrow StmtResult[State]], \text{expr}: [State \rightarrow ExprResult[State, ExprData]], s: State): (\exists (n: nat): \text{for_termination_point?}(b_ex, \text{expr}, body, s)(n)) \wedge \neg \text{min}[nat](\text{for_termination_point?}(b_ex, \text{expr}, body, s)) = 0 \supset OK?(b_ex(s)) \wedge \text{data}(b_ex(s))}{}$$

Repeatedly Skolemizing and flattening,

Rewriting using min_def, matching in *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of nonzero_min_for_termination_point.

Q.E.D.

C.108.14 IterationStatements2.for_as_while

Terse proof for for_as_while.

for_as_while:

$$\frac{\{1\} \quad \forall (b_ex: [State \rightarrow ExprResult[State, bool]], body: [State \rightarrow StmtResult[State]], \text{expr}: [State \rightarrow ExprResult[State, ExprData]], \text{init}: [State \rightarrow StmtResult[State]]): \text{for}(\text{init}, b_ex, \text{expr}, body) = (\text{init} \## \text{if_else}(b_ex, body \## \text{catch_continue} \## \text{while}(\text{expr} \## b_ex, body) \## \text{catch_break}, \text{skip}))}{}$$

Installing automatic rewrites from: for while comp_eval_if_ok_fstmt_single lift skip

Repeatedly Skolemizing and flattening,

Applying decompose-equality,

Expanding the definition of if_else,

Expanding the definition of while,

Expanding the definition of ##,

Case splitting on OK?(init!1(x!1)),

we get 2 subgoals:

for_as_while.1:

```

|-1} OK?(init'(x'))
|-----
{1} lift(λ (state: State):
      IF ∃ (n: nat): for_termination_point?(b_ex', expr', body', state)(n)
      THEN (iterate_for(min[nat]
                        (for_termination_point?(b_ex', expr', body', state)),
                        b_ex', expr', body')
            ## e2s(b_ex')
            ## catch_break)
            (state)
      ELSE Hang
      ENDIF)
      (init'(x'))
=
lift((b_ex' ##
      (λ (b: bool):
        IF b
          THEN (body' ## catch_continue ##
                (λ (s: State):
                  IF ∃ (n: nat):
                    while_termination_point?(expr' ## b_ex', body', s)(n)
                  THEN (iterate_while(min[nat]
                                      (while_termination_point?(expr'
                                                                expr' ## b_ex', body')
                                      ## e2s(expr' ## b_ex')
                                      ## catch_break)
                                      (s)
                  ELSE Hang
                  ENDIF))
                ## catch_break
          ELSE skip
          ENDIF)))
      (init'(x'))

```

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 2 subgoals:

for_as_while.1.1:

<pre> {-1} OK?(init'(x')) {-2} ∃ (n: nat): for_termination_point?(b_ex', expr', body', state(init'(x')))(n) </pre>	<pre> {1} (iterate_for(min[nat] (for_termination_point?(b_ex', expr', body', state(init'(x')))), b_ex', expr', body') ## e2s(b_ex') ## catch_break) (state(init'(x'))) = (b_ex' ## (λ (b: bool): IF b THEN (body' ## catch_continue ## (λ (s: State): IF ∃ (n: nat): while_termination_point?(expr' ## b_ex', body', s)(n) THEN (iterate_while(min[nat] (while_termination_point?(expr' ## b_ex', body', s)), expr' ## b_ex', body') ## e2s(expr' ## b_ex') ## catch_break) (s) ELSE Hang ENDIF)) ## catch_break ELSE skip ENDIF)) (state(init'(x'))) </pre>
--	---

Case splitting on $\text{min}(\text{for_termination_point?}(b_ex!1, \text{expr}!1, \text{body}!1, \text{state}(\text{init}!1(x!1)))) = 0$,

we get 2 subgoals:

for_as_while.1.1.1:

{-1}	min(for_termination_point?(b_ex', expr', body', state(init'(x')))) = 0
{-2}	OK?(init'(x'))
{-3}	$\exists (n: \text{nat}): \text{for_termination_point?}(b_ex', \text{expr}', \text{body}', \text{state}(\text{init}'(x')))(n)$
{1}	<pre> iterate_for(min[nat] (for_termination_point?(b_ex', expr', body', state(init'(x')))), b_ex', expr', body') ## e2s(b_ex') ## catch_break (state(init'(x'))) = (b_ex' ## (\lambda (b: bool): IF b THEN (body' ## catch_continue ## (\lambda (s: State): IF \exists (n: nat): while_termination_point?(expr' ## b_ex', body', s)(n) THEN (iterate_while(min[nat] (while_termination_point?(expr' ## b_ex', body', s)), expr' ## b_ex', body') ## e2s(expr' ## b_ex') ## catch_break (s) ELSE Hang ENDIF)) ## catch_break ELSE skip ENDIF)) (state(init'(x')))) </pre>

Replacing using formula -1,

Expanding the definition of iterate_for,

Rewriting using min_def, matching in *,

Hiding formulas: -3,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of for_as_while.1.1.1.

for_as_while.1.1.2:

{-1}	OK?(init'(x'))
{-2}	$\exists (n: \text{nat}): \text{for_termination_point?}(\text{b_ex}', \text{expr}', \text{body}', \text{state}(\text{init}'(x')))(n)$
{1}	$\text{min}(\text{for_termination_point?}(\text{b_ex}', \text{expr}', \text{body}', \text{state}(\text{init}'(x')))) = 0$
{2}	<pre> iterate_for(min[nat] (for_termination_point?(b_ex', expr', body', state(init'(x')))), b_ex', expr', body') ## e2s(b_ex') ## catch_break (state(init'(x'))) = (b_ex' ## (\lambda (b: bool): IF b THEN (body' ## catch_continue ## (\lambda (s: State): IF \exists (n: nat): while_termination_point?(expr' ## b_ex', body', s)(n) THEN (iterate_while(min[nat] (while_termination_point?(expr' ## b_ex', body', s)), expr' ## b_ex', body') ## e2s(expr' ## b_ex') ## catch_break (s) ELSE Hang ENDIF)) ## catch_break ELSE skip ENDIF)) (state(init'(x')))) </pre>

Using lemma nonzero_min_for_termination_point,

Rewriting using iterate_for_as_while, matching in *,

Rewriting using min_termination_point_for_as_while, matching in *,

we get 2 subgoals:

for_as_while.1.1.2.1:

```

{-1}  (∃ (n: nat): for_termination_point?(b_ex', expr', body', state(init'(x')))(n))
      ^
      ¬ 1 + min [nat](while_termination_point?(expr' ## b_ex', body', state [State]((e2s(b_ex') ##
      = 0
      ⊃ OK?(b_ex'(state(init'(x')))) ∧ data(b_ex'(state(init'(x'))))
{-2}  OK?(init'(x'))
{-3}  ∃ (n: nat): for_termination_point?(b_ex', expr', body', state(init'(x')))(n)
-----
{1}  1 + min [nat](while_termination_point?(expr' ## b_ex', body', state [State]((e2s(b_ex') ## b
      = 0
{2}  ((e2s(b_ex') ## body' ## catch_continue ##
      iterate_while(1 + min [nat](while_termination_point?(expr' ## b_ex', body', state [State]
      expr' ## b_ex', body')
      ## e2s(expr'))
      ## e2s(b_ex')
      ## catch_break
      (state(init'(x'))))
      =
      (b_ex' ##
      (λ (b: bool):
        IF b
          THEN (body' ## catch_continue ##
                (λ (s: State):
                  IF ∃ (n: nat):
                    while_termination_point?(expr' ## b_ex', body', s)(n)
                    THEN (iterate_while(min[nat]
                                          (while_termination_point?(expr' ## b
                                          body'
                                          s)),
                                          expr' ## b_ex', body')
                                          ## e2s(expr' ## b_ex')
                                          ## catch_break
                                          (s)
                    ELSE Hang
                    ENDIF))
                ## catch_break
          ELSE skip
          ENDIF))
      (state(init'(x'))))

```

Expanding the definition of ##,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Expanding the definition of lift,

Rewriting using comp_simple_stmt_simple_expr, matching in *,

Case splitting on OK?((body!1 ## catch_continue)(state((init!1 ## b_ex!1)(x!1))),

we get 2 subgoals:

for_as_while.1.1.2.1.1:

<pre> {-1} OK?((body' ## catch_continue)(state((init' ## b_ex')(x')))) {-2} ∃ (n: nat): for_termination_point?(b_ex', expr', body', state(init'(x')))(n) {-3} OK?((init' ## b_ex')(x')) {-4} data((init' ## b_ex')(x')) {-5} OK?(init'(x')) </pre>	<pre> {1} 1 + min [nat](while_termination_point?(expr' ## b_ex', body', state [State]((e2s(b_ex') ## body' ## = 0 {2} ((e2s(b_ex') ## body' ## catch_continue ## iterate_while(min[nat] (while_termination_point?(expr' ## b_ex', body', state[State] ((e2s(b_ex') ## body' ## catch_continue) (state(init'(x'))))))), expr' ## b_ex', body') ## e2s(expr') ## e2s(b_ex') ## catch_break (state(init'(x')))) = catch_break(CASES (body' ## catch_continue)(state((init' ## b_ex')(x'))) OF OK(state_1): IF ∃ (n: nat): while_termination_point?(expr' ## b_ex', body', state_1)(n) THEN (iterate_while(min[nat] (while_termination_point?(expr' ## b_ex', body', state_1)), expr' ## b_ex', body') ## e2s(expr' ## b_ex') ## catch_break (state_1) ELSE Hang ENDIF ELSE (body' ## catch_continue)(state((init' ## b_ex')(x'))) ENDCASES) </pre>
--	--

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 2 subgoals:

for_as_while.1.1.2.1.1.1:

$$\begin{array}{l}
 \{-1\} \text{ OK?}((\text{body}' \text{ ## catch_continue})(\text{state}((\text{init}' \text{ ## b_ex}')(\text{x}')))) \\
 \{-2\} \exists (n: \text{nat}): \text{for_termination_point?}(\text{b_ex}', \text{expr}', \text{body}', \text{state}(\text{init}'(\text{x}')))(n) \\
 \{-3\} \text{ OK?}((\text{init}' \text{ ## b_ex}')(\text{x}')) \\
 \{-4\} \text{ data}((\text{init}' \text{ ## b_ex}')(\text{x}')) \\
 \{-5\} \text{ OK?}(\text{init}'(\text{x}')) \\
 \{-6\} \exists (n: \text{nat}): \\
 \quad \text{while_termination_point?}(\text{expr}' \text{ ## b_ex}', \text{body}', \\
 \quad \quad \text{state}((\text{body}' \text{ ## catch_continue} \\
 \quad \quad \quad (\text{state}((\text{init}' \text{ ## b_ex}')(\text{x}'))))) \\
 \quad \quad \quad (n) \\
 \hline
 \{1\} 1 + \min [\text{nat}] (\text{while_termination_point?}(\text{expr}' \text{ ## b_ex}', \text{body}', \text{state} [\text{State}] ((\text{e2s}(\text{b_ex}') \text{ ## b} \\
 \quad = 0 \\
 \{2\} ((\text{e2s}(\text{b_ex}') \text{ ## body}' \text{ ## catch_continue} \text{ ##} \\
 \quad \quad \text{iterate_while}(\min [\text{nat}] \\
 \quad \quad \quad (\text{while_termination_point?}(\text{expr}' \text{ ## b_ex}', \text{body}', \\
 \quad \quad \quad \quad \text{state} [\text{State}] \\
 \quad \quad \quad \quad \quad ((\text{e2s}(\text{b_ex}') \text{ ## body}' \\
 \quad \quad \quad \quad \quad \quad \text{##} \\
 \quad \quad \quad \quad \quad \quad \text{catch_continue}) \\
 \quad \quad \quad \quad \quad \quad (\text{state}(\text{init}'(\text{x}'))))))), \\
 \quad \quad \quad \text{expr}' \text{ ## b_ex}', \text{body}') \\
 \quad \quad \quad \text{## e2s}(\text{expr}')) \\
 \quad \quad \quad \text{## e2s}(\text{b_ex}') \\
 \quad \quad \quad \text{## catch_break} \\
 \quad \quad \quad (\text{state}(\text{init}'(\text{x}')))) \\
 = \\
 \text{catch_break}((\text{iterate_while}(\min [\text{nat}] \\
 \quad \quad \quad \quad (\text{while_termination_point?}(\text{expr}' \text{ ## b_ex}', \\
 \quad \quad \quad \quad \quad \text{body}', \\
 \quad \quad \quad \quad \quad \text{state} \\
 \quad \quad \quad \quad \quad ((\text{body}' \text{ ## catch_com} \\
 \quad \quad \quad \quad \quad \quad (\text{state} \\
 \quad \quad \quad \quad \quad \quad \quad (\text{init}' \text{ ## b_ex}') \\
 \quad \quad \quad \quad \quad \quad \quad (\text{x}'))))))), \\
 \quad \quad \quad \quad \quad \quad \text{expr}' \text{ ## b_ex}', \text{body}') \\
 \quad \quad \quad \quad \text{## e2s}(\text{expr}' \text{ ## b_ex}') \\
 \quad \quad \quad \quad \text{## catch_break} \\
 \quad \quad \quad \quad (\text{state}((\text{body}' \text{ ## catch_continue} \\
 \quad \quad \quad \quad \quad (\text{state}((\text{init}' \text{ ## b_ex}')(\text{x}'))))))))
 \end{array}$$

Keeping (-1 -3 -5 2) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of for_as_while.1.1.2.1.1.1.

for_as_while.1.1.2.1.1.2:

<pre> {-1} OK?((body' ## catch_continue)(state((init' ## b_ex')(x')))) {-2} ∃ (n: nat): for_termination_point?(b_ex', expr', body', state(init'(x')))(n) {-3} OK?((init' ## b_ex')(x')) {-4} data((init' ## b_ex')(x')) {-5} OK?(init'(x')) </pre>	<pre> {1} 1 + min [nat](while_termination_point?(expr' ## b_ex', body', state [State]((e2s(b_ex') ## body' ## = 0 {2} ∃ (n: nat): while_termination_point?(expr' ## b_ex', body', state((body' ## catch_continue) (state((init' ## b_ex')(x'))))) (n) {3} ((e2s(b_ex') ## body' ## catch_continue ## iterate_while(min[nat] (while_termination_point?(expr' ## b_ex', body', state[State] ((e2s(b_ex') ## body' ## catch_continue) (state(init'(x'))))))), expr' ## b_ex', body') ## e2s(expr') ## e2s(b_ex') ## catch_break (state(init'(x')))) = catch_break(Hang) </pre>
--	---

Hiding formulas: 3,

Repeatedly Skolemizing and flattening,

Case splitting on $n! = 0$,

we get 2 subgoals:

for_as_while.1.1.2.1.1.2.1:

<pre> {-1} n' = 0 {-2} n' ≥ 0 {-3} OK?((body' ## catch_continue)(state((init' ## b_ex')(x')))) {-4} for_termination_point?(b_ex', expr', body', state(init'(x')))(n') {-5} OK?((init' ## b_ex')(x')) {-6} data((init' ## b_ex')(x')) {-7} OK?(init'(x')) </pre>	<pre> {1} 1 + min [nat](while_termination_point?(expr' ## b_ex', body', state [State]((e2s(b_ex') ## body' ## = 0 {2} ∃ (n: nat): while_termination_point?(expr' ## b_ex', body', state((body' ## catch_continue) (state((init' ## b_ex')(x'))))) (n) </pre>
---	--

Replacing using formula -1,

Expanding the definition of for_termination_point?,

Expanding the definition of iterate_for,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of for_as_while.1.1.2.1.1.2.1.

for_as_while.1.1.2.1.1.2.2:

<pre> {-1} n' ≥ 0 {-2} OK?((body' ## catch_continue)(state((init' ## b_ex')(x')))) {-3} for_termination_point?(b_ex', expr', body', state(init'(x')))(n') {-4} OK?((init' ## b_ex')(x')) {-5} data((init' ## b_ex')(x')) {-6} OK?(init'(x')) </pre>	<hr/> <pre> {1} n' = 0 {2} 1 + min [nat](while_termination_point?(expr' ## b_ex', body', state [State]((e2s(b_ex') ## b_ex')(x')))) = 0 {3} ∃ (n: nat): while_termination_point?(expr' ## b_ex', body', state((body' ## catch_continue) (state((init' ## b_ex')(x'))))) </pre> <p style="text-align: center;">(n)</p>
---	--

Instantiating the top quantifier in 3 with the terms: $n' - 1$,

we get 2 subgoals:

for_as_while.1.1.2.1.1.2.2.1:

<pre> {-1} n' ≥ 0 {-2} OK?((body' ## catch_continue)(state((init' ## b_ex')(x')))) {-3} for_termination_point?(b_ex', expr', body', state(init'(x')))(n') {-4} OK?((init' ## b_ex')(x')) {-5} data((init' ## b_ex')(x')) {-6} OK?(init'(x')) </pre>	<hr/> <pre> {1} n' = 0 {2} 1 + min [nat](while_termination_point?(expr' ## b_ex', body', state [State]((e2s(b_ex') ## b_ex')(x')))) = 0 {3} while_termination_point?(expr' ## b_ex', body', state((body' ## catch_continue) (state((init' ## b_ex')(x'))))) </pre> <p style="text-align: center;">($n' - 1$)</p>
---	--

Rewriting using `termination_point_for_as_while`, matching in `*`,

we get 2 subgoals:

for_as_while.1.1.2.1.1.2.2.1.1:

<pre> {-1} n' ≥ 0 {-2} OK?((body' ## catch_continue)(state((init' ## b_ex')(x')))) {-3} while_termination_point?(expr' ## b_ex', body', state((e2s(b_ex') ## body' ## catch_continue) (state(init'(x'))))) </pre> <p style="text-align: center;">($n' - 1$)</p> <pre> {-4} OK?((init' ## b_ex')(x')) {-5} data((init' ## b_ex')(x')) {-6} OK?(init'(x')) </pre>	<hr/> <pre> {1} n' = 0 {2} 1 + min [nat](while_termination_point?(expr' ## b_ex', body', state [State]((e2s(b_ex') ## b_ex')(x')))) = 0 {3} while_termination_point?(expr' ## b_ex', body', state((body' ## catch_continue) (state((init' ## b_ex')(x'))))) </pre> <p style="text-align: center;">($n' - 1$)</p>
---	--

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `for_as_while.1.1.2.1.1.2.2.1.1`.

`for_as_while.1.1.2.1.1.2.2.1.2`:

<pre> {-1} n' ≥ 0 {-2} OK?((body' ## catch_continue)(state((init' ## b_ex')(x')))) {-3} for_termination_point?(b_ex', expr', body', state(init'(x')))(n') {-4} OK?((init' ## b_ex')(x')) {-5} data((init' ## b_ex')(x')) {-6} OK?(init'(x')) </pre>	<hr style="border: 0.5px solid black;"/> <pre> {1} OK?((e2s(b_ex') ## body' ## catch_continue)(state(init'(x')))) {2} n' = 0 {3} 1 + min [nat](while_termination_point?(expr' ## b_ex', body', state [State]((e2s(b_ex') ## body' ## = 0 {4} while_termination_point?(expr' ## b_ex', body', state((body' ## catch_continue) (state((init' ## b_ex')(x'))))) (n' - 1) </pre>
---	--

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `for_as_while.1.1.2.1.1.2.2.1.2`.

`for_as_while.1.1.2.1.1.2.2.2`:

<pre> {-1} n' ≥ 0 {-2} OK?((body' ## catch_continue)(state((init' ## b_ex')(x')))) {-3} for_termination_point?(b_ex', expr', body', state(init'(x')))(n') {-4} OK?((init' ## b_ex')(x')) {-5} data((init' ## b_ex')(x')) {-6} OK?(init'(x')) </pre>	<hr style="border: 0.5px solid black;"/> <pre> {1} n' - 1 ≥ 0 {2} n' = 0 {3} 1 + min [nat](while_termination_point?(expr' ## b_ex', body', state [State]((e2s(b_ex') ## body' ## = 0 </pre>
---	---

Keeping (-1 1 2) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `for_as_while.1.1.2.1.1.2.2.2`.

for_as_while.1.1.2.1.2:

```

{-1}  ∃ (n: nat): for_termination_point?(b_ex', expr', body', state(init'(x')))(n)
{-2}  OK?((init' ## b_ex')(x'))
{-3}  data((init' ## b_ex')(x'))
{-4}  OK?(init'(x'))
-----
{1}   OK?((body' ## catch_continue)(state((init' ## b_ex')(x'))))
{2}   1 + min [nat](while_termination_point?(expr' ## b_ex', body', state [State]((e2s(b_ex') ## b_ex',
= 0
{3}   ((e2s(b_ex') ## body' ## catch_continue ##
      iterate_while(min [nat]
                    (while_termination_point?(expr' ## b_ex', body',
                                                state[State]
                                                  ((e2s(b_ex') ## body'
                                                    ##
                                                    catch_continue)
                                                    (state(init'(x'))))))),
                    expr' ## b_ex', body')
      ## e2s(expr')
      ## e2s(b_ex')
      ## catch_break
      (state(init'(x'))))
=
catch_break(CASES (body' ## catch_continue)(state((init' ## b_ex')(x')))) OF
  OK(state_1):
    IF ∃ (n: nat):
      while_termination_point?(expr' ## b_ex', body', state_1)(n)
      THEN (iterate_while(min [nat]
                          (while_termination_point?(expr' ## b_ex',
                                                      body',
                                                      state_1)),
                          expr' ## b_ex', body')
           ## e2s(expr' ## b_ex')
           ## catch_break
           (state_1))
    ELSE Hang
  ENDIF
ELSE (body' ## catch_continue)(state((init' ## b_ex')(x'))))
ENDCASES)

```

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of for_as_while.1.1.2.1.2.

for_as_while.1.1.2.2:

{-1}	(∃ (n: nat): for_termination_point?(b_ex', expr', body', state(init'(x')))(n))	
	⊃ OK?(b_ex'(state(init'(x')))) ∧ data(b_ex'(state(init'(x'))))	
{-2}	OK?(init'(x'))	
{-3}	∃ (n: nat): for_termination_point?(b_ex', expr', body', state(init'(x')))(n)	
{1}	OK?[State]((e2s(b_ex') ## body' ## catch_continue)(state(init'(x'))))	
{2}	min(for_termination_point?(b_ex', expr', body', state(init'(x')))) = 0	
{3}	((e2s(b_ex') ## body' ## catch_continue ## iterate_while(min [nat](for_termination_point?(b_ex', expr', body', state(init'(x')))) - 1, expr' ## b_ex', body') ## e2s(expr') ## e2s(b_ex') ## catch_break) (state(init'(x'))))	
	=	
	(b_ex' ## (λ (b: bool): IF b THEN (body' ## catch_continue ## (λ (s: State): IF ∃ (n: nat): while_termination_point?(expr' ## b_ex', body', s)(n) THEN (iterate_while(min [nat] (while_termination_point?(expr' ## b_ex', body', s)), expr' ## b_ex', body') ## e2s(expr' ## b_ex') ## catch_break) (s) ELSE Hang ENDIF)) ## catch_break ELSE skip ENDIF)) (state(init'(x'))))	

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of for_as_while.1.1.2.2.

for_as_while.1.2:

<pre> {-1} OK?(init'(x')) {1} ∃ (n: nat): for_termination_point?(b_ex', expr', body', state(init'(x')))(n) {2} Hang?((b_ex' ## (λ (b: bool): IF b THEN (body' ## catch_continue ## (λ (s: State): IF ∃ (n: nat): while_termination_point?(expr' ## b_ex', body', s)(n) THEN (iterate_while(min[nat] (while_termination_point?(expr' expr' ## b_ex', body') ## e2s(expr' ## b_ex') ## catch_break) (s) ELSE Hang ENDIF)) ## catch_break ELSE skip ENDIF)) (state(init'(x')))) </pre>	
---	--

Copying formula number: 1

Copying formula number: 1

Instantiating the top quantifier in 1 with the terms: 0,

Instantiating the top quantifier in 2 with the terms: 1,

Expanding the definition of for_termination_point?,

Expanding the definition of iterate_for,

Expanding the definition of iterate_for,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Expanding the definition of ##,

Expanding the definition of catch_break,

Expanding the definition of lift,

Expanding the definition of e2s,

Simplifying, rewriting, and recording with decision procedures,

Expanding the definition of catch_continue,

Simplifying, rewriting, and recording with decision procedures,

Case splitting on OK?(body!1(state(b_ex!1(state(init!1(x!1)))))) OR Continue?(body!1(state(b_ex!1(state(init!1(x!1))))))

we get 2 subgoals:

for_as_while.1.2.1:

```

{-1} OK?(body'(state(b_ex'(state(init'(x'))))) ∨
      Continue?(body'(state(b_ex'(state(init'(x'))))))
{-2} OK?(OK(state(init'(x'))))
{-3} OK?(b_ex'(state(init'(x'))))
{-4} data(b_ex'(state(init'(x'))))
{-5} OK?((((λ (s: State):
          CASES b_ex'(s) OF
            OK(state, data): OK(state),
            Fatal: Fatal,
            Hang: Hang,
            Exception(ex_1, state): Exception(ex_1, state)
          ENDCASES)
        ## body'
        ##
        (λ (res: StmtResult[State]):
          CASES res OF Continue(state): OK(state) ELSE res ENDCASES))
      ##
      (λ (s: State):
        CASES expr'(s) OF
          OK(state, data): OK(state),
          Fatal: Fatal,
          Hang: Hang,
          Exception(ex_1, state): Exception(ex_1, state)
        ENDCASES))
      ## (λ (s): OK(s))
      (state(init'(x'))))
{-6} OK?(b_ex'(state((((λ (s: State):
          CASES b_ex'(s) OF
            OK(state, data): OK(state),
            Fatal: Fatal,
            Hang: Hang,
            Exception(ex_1, state): Exception(ex_1, state)
          ENDCASES)
        ## body'
        ##
        (λ (res: StmtResult[State]):
          CASES res OF Continue(state): OK(state) ELSE res END-
CASES))
      ##
      (λ (s: State):
        CASES expr'(s) OF
          OK(state, data): OK(state),
          Fatal: Fatal,
          Hang: Hang,
          Exception(ex_1, state): Exception(ex_1, state)
        ENDCASES))
      ## (λ (s): OK(s))
      (state(init'(x')))))
{-7} data(b_ex'(state((((λ (s: State):
          CASES b_ex'(s) OF
            OK(state, data): OK(state),
            Fatal: Fatal,
            Hang: Hang,
            Exception(ex_1, state): Exception(ex_1, state)
          ENDCASES)
        ## body'
        ##
        (λ (res: StmtResult[State]):
          CASES res OF Continue(state): OK(state) ELSE res END-
CASES))
      ##
      (λ (s: State):
        CASES expr'(s) OF
          OK(state, data): OK(state),
          Fatal: Fatal,
          Hang: Hang,
          Exception(ex_1, state): Exception(ex_1, state)
        ENDCASES))
      ## (λ (s): OK(s))
      (state(init'(x')))))

```


C Proof scripts

Case splitting on EXISTS (n: nat): while_termination_point?(expr!1 ## b_ex!1, body!1, state(body!1 (state(b_ex!1(state(init!1(x!1))))))) (n),

we get 3 subgoals:

for_as_while.1.2.1.1:

```

{-1}  ∃ (n: nat):
      while_termination_point?(expr' ## b_ex', body',
                               state(body'(state(b_ex'(state(init'(x'))))))))
      (n)
{-2}  OK?(body'(state(b_ex'(state(init'(x')))))) ∨
      Continue?(body'(state(b_ex'(state(init'(x'))))))
{-3}  OK?(OK(state(init'(x'))))
{-4}  OK?(b_ex'(state(init'(x'))))
{-5}  data(b_ex'(state(init'(x'))))
{-6}  OK?((((λ (s: State):
             CASES b_ex'(s) OF
               OK(state, data): OK(state),
               Fatal: Fatal,
               Hang: Hang,
               Exception(ex_1, state): Exception(ex_1, state)
             ENDCASES)
           ## body'
           ##
           (λ (res: StmtResult[State]):
             CASES res OF Continue(state): OK(state) ELSE res ENDCASES))
        ##
        (λ (s: State):
          CASES expr'(s) OF
            OK(state, data): OK(state),
            Fatal: Fatal,
            Hang: Hang,
            Exception(ex_1, state): Exception(ex_1, state)
          ENDCASES))
        ## (λ (s): OK(s))
        (state(init'(x'))))
{-7}  OK?(b_ex'(state((((λ (s: State):
             CASES b_ex'(s) OF
               OK(state, data): OK(state),
               Fatal: Fatal,
               Hang: Hang,
               Exception(ex_1, state): Exception(ex_1, state)
             ENDCASES)
           ## body'
           ##
           (λ (res: StmtResult[State]):
             CASES res OF Continue(state): OK(state) ELSE res END-
CASES))
        ##
        (λ (s: State):
          CASES expr'(s) OF
            OK(state, data): OK(state),
            Fatal: Fatal,
            Hang: Hang,
            Exception(ex_1, state): Exception(ex_1, state)
          ENDCASES))
        ## (λ (s): OK(s))
        (state(init'(x'))))
{-8}  data(b_ex'(state((((λ (s: State):
             CASES b_ex'(s) OF
               OK(state, data): OK(state),
               Fatal: Fatal,
               Hang: Hang,
               Exception(ex_1, state): Exception(ex_1, state)
             ENDCASES)
           ## body'
           ##
           (λ (res: StmtResult[State]):
             CASES res OF Continue(state): OK(state) ELSE res END-
CASES))
        ##
        (λ (s: State):
          CASES expr'(s) OF
            OK(state, data): OK(state),
            Fatal: Fatal,
            Hang: Hang,
            Exception(ex_1, state): Exception(ex_1, state)
          ENDCASES))
        ## (λ (s): OK(s))
        (state(init'(x'))))

```

C Proof scripts

Hiding formulas: 2,

Repeatedly Skolemizing and flattening,

Instantiating the top quantifier in 1 with the terms: $n' + 1$,

Rewriting using `termination_point_for_as_while`, matching in `*`,

we get 2 subgoals:

for_as_while.1.2.1.1.1:

```

{-1}  n' ≥ 0
{-2}  while_termination_point?(expr' ## b_ex', body',
      state(body'(state(b_ex'(state(init'(x'))))))))
      (n')
{-3}  OK?(body'(state(b_ex'(state(init'(x')))))) ∨
      Continue?(body'(state(b_ex'(state(init'(x'))))))
{-4}  OK?(OK(state(init'(x'))))
{-5}  OK?(b_ex'(state(init'(x'))))
{-6}  data(b_ex'(state(init'(x'))))
{-7}  OK?((((λ (s: State):
          CASES b_ex'(s) OF
            OK(state, data): OK(state),
            Fatal: Fatal,
            Hang: Hang,
            Exception(ex_1, state): Exception(ex_1, state)
          ENDCASES)
          ## body'
          ##
          (λ (res: StmtResult[State]):
            CASES res OF Continue(state): OK(state) ELSE res ENDCASES))
          ##
          (λ (s: State):
            CASES expr'(s) OF
              OK(state, data): OK(state),
              Fatal: Fatal,
              Hang: Hang,
              Exception(ex_1, state): Exception(ex_1, state)
            ENDCASES))
          ## (λ (s): OK(s))
          (state(init'(x'))))
{-8}  OK?(b_ex'(state((((λ (s: State):
          CASES b_ex'(s) OF
            OK(state, data): OK(state),
            Fatal: Fatal,
            Hang: Hang,
            Exception(ex_1, state): Exception(ex_1, state)
          ENDCASES)
          ## body'
          ##
          (λ (res: StmtResult[State]):
            CASES res OF Continue(state): OK(state) ELSE res END-
CASES))
          ##
          (λ (s: State):
            CASES expr'(s) OF
              OK(state, data): OK(state),
              Fatal: Fatal,
              Hang: Hang,
              Exception(ex_1, state): Exception(ex_1, state)
            ENDCASES))
          ## (λ (s): OK(s))
          (state(init'(x'))))))
{-9}  data(b_ex'(state((((λ (s: State):
          CASES b_ex'(s) OF
            OK(state, data): OK(state),
            Fatal: Fatal,
            Hang: Hang,
            Exception(ex_1, state): Exception(ex_1, state)
          ENDCASES)
          ## body'
          ##
          (λ (res: StmtResult[State]):
            CASES res OF Continue(state): OK(state) ELSE res END-
CASES))
          ##
          (λ (s: State):
            CASES expr'(s) OF
              OK(state, data): OK(state),
              Fatal: Fatal,
              Hang: Hang,
              Exception(ex_1, state): Exception(ex_1, state)
            ENDCASES))
          ## (λ (s): OK(s))
          (state(init'(x'))))))

```

C Proof scripts

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `for_as_while.1.2.1.1.1`.

for_as_while.1.2.1.1.2:

```

{-1}  n' ≥ 0
{-2}  while_termination_point?(expr' ## b_ex', body',
      state(body'(state(b_ex'(state(init'(x'))))))))
      (n')
{-3}  OK?(body'(state(b_ex'(state(init'(x')))))) ∨
      Continue?(body'(state(b_ex'(state(init'(x'))))))
{-4}  OK?(OK(state(init'(x'))))
{-5}  OK?(b_ex'(state(init'(x'))))
{-6}  data(b_ex'(state(init'(x'))))
{-7}  OK?((((λ (s: State):
          CASES b_ex'(s) OF
            OK(state, data): OK(state),
            Fatal: Fatal,
            Hang: Hang,
            Exception(ex_1, state): Exception(ex_1, state)
          ENDCASES)
          ## body'
          ##
          (λ (res: StmtResult[State]):
            CASES res OF Continue(state): OK(state) ELSE res ENDCASES))
          ##
          (λ (s: State):
            CASES expr'(s) OF
              OK(state, data): OK(state),
              Fatal: Fatal,
              Hang: Hang,
              Exception(ex_1, state): Exception(ex_1, state)
            ENDCASES))
          ## (λ (s): OK(s))
          (state(init'(x'))))
{-8}  OK?(b_ex'(state((((λ (s: State):
          CASES b_ex'(s) OF
            OK(state, data): OK(state),
            Fatal: Fatal,
            Hang: Hang,
            Exception(ex_1, state): Exception(ex_1, state)
          ENDCASES)
          ## body'
          ##
          (λ (res: StmtResult[State]):
            CASES res OF Continue(state): OK(state) ELSE res END-
CASES))
          ##
          (λ (s: State):
            CASES expr'(s) OF
              OK(state, data): OK(state),
              Fatal: Fatal,
              Hang: Hang,
              Exception(ex_1, state): Exception(ex_1, state)
            ENDCASES))
          ## (λ (s): OK(s))
          (state(init'(x'))))))
{-9}  data(b_ex'(state((((λ (s: State):
          CASES b_ex'(s) OF
            OK(state, data): OK(state),
            Fatal: Fatal,
            Hang: Hang,
            Exception(ex_1, state): Exception(ex_1, state)
          ENDCASES)
          ## body'
          ##
          (λ (res: StmtResult[State]):
            CASES res OF Continue(state): OK(state) ELSE res END-
CASES))
          ##
          (λ (s: State):
            CASES expr'(s) OF
              OK(state, data): OK(state),
              Fatal: Fatal,
              Hang: Hang,
              Exception(ex_1, state): Exception(ex_1, state)
            ENDCASES))
          ## (λ (s): OK(s))
          (state(init'(x'))))))

```

C Proof scripts

Hiding formulas: (-2 2),

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `for_as_while.1.2.1.1.2`.

for_as_while.1.2.1.2:

```

{-1} OK?(body'(state(b_ex'(state(init'(x'))))) ∨
      Continue?(body'(state(b_ex'(state(init'(x'))))))
{-2} OK?(OK(state(init'(x'))))
{-3} OK?(b_ex'(state(init'(x'))))
{-4} data(b_ex'(state(init'(x'))))
{-5} OK?((((λ (s: State):
          CASES b_ex'(s) OF
            OK(state, data): OK(state),
            Fatal: Fatal,
            Hang: Hang,
            Exception(ex_1, state): Exception(ex_1, state)
          ENDCASES)
        ## body'
        ##
        (λ (res: StmtResult[State]):
          CASES res OF Continue(state): OK(state) ELSE res ENDCASES))
      ##
      (λ (s: State):
        CASES expr'(s) OF
          OK(state, data): OK(state),
          Fatal: Fatal,
          Hang: Hang,
          Exception(ex_1, state): Exception(ex_1, state)
        ENDCASES))
      ## (λ (s): OK(s))
      (state(init'(x'))))
{-6} OK?(b_ex'(state((((λ (s: State):
          CASES b_ex'(s) OF
            OK(state, data): OK(state),
            Fatal: Fatal,
            Hang: Hang,
            Exception(ex_1, state): Exception(ex_1, state)
          ENDCASES)
        ## body'
        ##
        (λ (res: StmtResult[State]):
          CASES res OF Continue(state): OK(state) ELSE res END-
CASES))
      ##
      (λ (s: State):
        CASES expr'(s) OF
          OK(state, data): OK(state),
          Fatal: Fatal,
          Hang: Hang,
          Exception(ex_1, state): Exception(ex_1, state)
        ENDCASES))
      ## (λ (s): OK(s))
      (state(init'(x')))))
{-7} data(b_ex'(state((((λ (s: State):
          CASES b_ex'(s) OF
            OK(state, data): OK(state),
            Fatal: Fatal,
            Hang: Hang,
            Exception(ex_1, state): Exception(ex_1, state)
          ENDCASES)
        ## body'
        ##
        (λ (res: StmtResult[State]):
          CASES res OF Continue(state): OK(state) ELSE res END-
CASES))
      ##
      (λ (s: State):
        CASES expr'(s) OF

```


C Proof scripts

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `for_as_while.1.2.1.2`.

for_as_while.1.2.1.3:

```

{-1} OK?(body'(state(b_ex'(state(init'(x'))))) ∨
      Continue?(body'(state(b_ex'(state(init'(x'))))))
{-2} OK?(OK(state(init'(x'))))
{-3} OK?(b_ex'(state(init'(x'))))
{-4} data(b_ex'(state(init'(x'))))
{-5} OK?((((λ (s: State):
          CASES b_ex'(s) OF
            OK(state, data): OK(state),
            Fatal: Fatal,
            Hang: Hang,
            Exception(ex_1, state): Exception(ex_1, state)
          ENDCASES)
        ## body'
        ##
        (λ (res: StmtResult[State]):
          CASES res OF Continue(state): OK(state) ELSE res ENDCASES))
      ##
      (λ (s: State):
        CASES expr'(s) OF
          OK(state, data): OK(state),
          Fatal: Fatal,
          Hang: Hang,
          Exception(ex_1, state): Exception(ex_1, state)
        ENDCASES))
      ## (λ (s): OK(s))
      (state(init'(x'))))
{-6} OK?(b_ex'(state((((λ (s: State):
          CASES b_ex'(s) OF
            OK(state, data): OK(state),
            Fatal: Fatal,
            Hang: Hang,
            Exception(ex_1, state): Exception(ex_1, state)
          ENDCASES)
        ## body'
        ##
        (λ (res: StmtResult[State]):
          CASES res OF Continue(state): OK(state) ELSE res END-
CASES))
      ##
      (λ (s: State):
        CASES expr'(s) OF
          OK(state, data): OK(state),
          Fatal: Fatal,
          Hang: Hang,
          Exception(ex_1, state): Exception(ex_1, state)
        ENDCASES))
      ## (λ (s): OK(s))
      (state(init'(x')))))
{-7} data(b_ex'(state((((λ (s: State):
          CASES b_ex'(s) OF
            OK(state, data): OK(state),
            Fatal: Fatal,
            Hang: Hang,
            Exception(ex_1, state): Exception(ex_1, state)
          ENDCASES)
        ## body'
        ##
        (λ (res: StmtResult[State]):
          CASES res OF Continue(state): OK(state) ELSE res END-
CASES))
      ##
      (λ (s: State):
        CASES expr'(s) OF
          OK(state, data): OK(state),
          Fatal: Fatal,
          Hang: Hang,
          Exception(ex_1, state): Exception(ex_1, state)
        ENDCASES))
      ## (λ (s): OK(s))
      (state(init'(x')))))

```

C Proof scripts

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `for_as_while.1.2.1.3`.

for_as_while.1.2.2:

```

{-1} OK?(OK(state(init'(x'))))
{-2} OK?(b_ex'(state(init'(x'))))
{-3} data(b_ex'(state(init'(x'))))
{-4} OK?((((λ (s: State):
      CASES b_ex'(s) OF
        OK(state, data): OK(state),
        Fatal: Fatal,
        Hang: Hang,
        Exception(ex_1, state): Exception(ex_1, state)
      ENDCASES)
    ## body'
    ##
    (λ (res: StmtResult[State]):
      CASES res OF Continue(state): OK(state) ELSE res ENDCASES))
  ##
  (λ (s: State):
    CASES expr'(s) OF
      OK(state, data): OK(state),
      Fatal: Fatal,
      Hang: Hang,
      Exception(ex_1, state): Exception(ex_1, state)
    ENDCASES))
  ## (λ (s): OK(s))
  (state(init'(x'))))
{-5} OK?(b_ex'(state((((λ (s: State):
      CASES b_ex'(s) OF
        OK(state, data): OK(state),
        Fatal: Fatal,
        Hang: Hang,
        Exception(ex_1, state): Exception(ex_1, state)
      ENDCASES)
    ## body'
    ##
    (λ (res: StmtResult[State]):
      CASES res OF Continue(state): OK(state) ELSE res END-
CASES))
    ##
    (λ (s: State):
      CASES expr'(s) OF
        OK(state, data): OK(state),
        Fatal: Fatal,
        Hang: Hang,
        Exception(ex_1, state): Exception(ex_1, state)
      ENDCASES))
    ## (λ (s): OK(s))
    (state(init'(x')))))
{-6} data(b_ex'(state((((λ (s: State):
      CASES b_ex'(s) OF
        OK(state, data): OK(state),
        Fatal: Fatal,
        Hang: Hang,
        Exception(ex_1, state): Exception(ex_1, state)
      ENDCASES)
    ## body'
    ##
    (λ (res: StmtResult[State]):
      CASES res OF Continue(state): OK(state) ELSE res END-
CASES))
    ##
    (λ (s: State):
      CASES expr'(s) OF
        OK(state, data): OK(state),
        Fatal: Fatal,
        Hang: Hang,
        Exception(ex_1, state): Exception(ex_1, state)
      ENDCASES))
    ## (λ (s): OK(s))
    (state(init'(x')))))

```

C Proof scripts

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `for_as_while.1.2.2`.

`for_as_while.2`:

```
{1} OK?(init'(x'))
{2} lift(λ (state: State):
      IF ∃ (n: nat): for_termination_point?(b_ex', expr', body', state)(n)
      THEN (iterate_for(min[nat]
                        (for_termination_point?(b_ex', expr', body', state)),
                        b_ex', expr', body')
            ## e2s(b_ex')
            ## catch_break
            (state)
      ELSE Hang
      ENDIF)
      (init'(x'))
=
lift((b_ex' ##
      (λ (b: bool):
        IF b
          THEN (body' ## catch_continue ##
                (λ (s: State):
                  IF ∃ (n: nat):
                    while_termination_point?(expr' ## b_ex', body', s)(n)
                  THEN (iterate_while(min[nat]
                                     (while_termination_point?(expr'
                                                                expr' ## b_ex', body')
                                     ## e2s(expr' ## b_ex')
                                     ## catch_break
                                     (s)
                  ELSE Hang
                  ENDIF))
                ## catch_break
          ELSE skip
          ENDIF)))
      (init'(x'))
```

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `for_as_while.2`.

Q.E.D.

C.108.15 IterationStatements2.for_unroll

Terse proof for `for_unroll`.

for_unroll:

$\{1\} \quad \forall (b_ex: [State \rightarrow ExprResult[State, bool]], body: [State \rightarrow StmtResult[State]],$ $\quad expr: [State \rightarrow ExprResult[State, ExprData]], init: [State \rightarrow StmtResult[State]],$ $\quad max: nat):$ $\quad for(max, init, b_ex, expr, body) =$ $\quad (init \##$ $\quad \quad if_else(b_ex, body \## catch_continue \## while(max, expr \## b_ex, body) \## catch_break,$ $\quad \quad \quad skip))$

Repeatedly Skolemizing and flattening,
 Expanding the definition of for,
 Expanding the definition of while,
 Rewriting using for_as_while, matching in *,
 This completes the proof of for_unroll.
 Q.E.D.

C.109 Proofs for IterationStatements3 (statements.pvs)

C.109.1 IterationStatements3.do_while_inv_unroll

Terse proof for do_while_inv_unroll.

do_while_inv_unroll:

$\{1\} \quad \forall (<: PRED[[Ord, Ord]], P, Q, R: [State \rightarrow PRED[StmtResult[State]]],$ $\quad b_ex: [State \rightarrow ExprResult[State, bool]], body: [State \rightarrow StmtResult[State]],$ $\quad variant: [State \rightarrow [State \rightarrow Ord]]):$ $\quad do_while(body, b_ex)(P, Q, R, variant, <) =$ $\quad (body \## catch_continue \## while(b_ex, body)(P, Q, R, variant, <) \## catch_break)$

Repeatedly Skolemizing and flattening,
 Expanding the definition of do_while,
 Expanding the definition of while,
 Rewriting using do_as_while, matching in *,
 This completes the proof of do_while_inv_unroll.
 Q.E.D.

C.110 Proofs for IterationStatements4 (statements.pvs)

C.110.1 IterationStatements4.for_inv_unroll

Terse proof for for_inv_unroll.

for_inv_unroll:

```

{1}  ∀ (<: PRED[[Ord, Ord]], P, Q, R: [State → PRED[StmtResult[State]]],
      b_ex: [State → ExprResult[State, bool]], body: [State → StmtRe-
sult[State]],
      expr: [State → ExprResult[State, ExprData]], init: [State → StmtRe-
sult[State]],
      variant: [State → [State → Ord]]):
for(init, b_ex, expr, body)(P, Q, R, variant, <) =
  (init ##
   if_else(b_ex,
            body ## catch_continue ## while(expr ## b_ex, body)(P, Q, R, vari-
ant, <) ##
            catch_break,
            skip))

```

Repeatedly Skolemizing and flattening,
Expanding the definition of for,
Expanding the definition of while,
Rewriting using for_as_while, matching in *,
This completes the proof of for_inv_unroll.
Q.E.D.

C.111 Proofs for JumpStatements1 (statements.pvs)

This theory contains no provable formal statements.

C.112 Proofs for JumpStatements2 (statements.pvs)

This theory contains no provable formal statements.

C.113 Proofs for LabeledStatements (statements.pvs)

This theory contains no provable formal statements.

C.114 Proofs for Linear_Memory (linear_memory.pvs)

C.114.1 Linear_Memory.xlat_idx_TCC1

Terse proof for xlat_idx_TCC1.

xlat_idx_TCC1:

```

{1}  ∀ (lvl: Level, base, addr: Memory_Address_4G): offset(addr) ≥ 0

```

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of xlat_idx_TCC1.
Q.E.D.

C.114.2 Linear_Memory.xlat_idx_TCC2

Terse proof for xlat_idx_TCC2.

xlat_idx_TCC2:

$$\{1\} \quad \forall (lvl: \text{Level}, \text{base}, \text{addr}: \text{Memory_Address_4G}):$$

$$\text{bus_width} - (lvl + 1) \times \text{bits_per_level} \geq 0$$

Trying repeated skolemization, instantiation, and if-lifting,

Using lemma bus_width,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of xlat_idx_TCC2.

Q.E.D.

C.114.3 Linear_Memory.xlat_idx_pe_aligned_TCC1

Terse proof for xlat_idx_pe_aligned_TCC1.

xlat_idx_pe_aligned_TCC1:

$$\{1\} \quad \forall (lvl: \text{Level}, \text{base}, \text{addr}: \text{Memory_Address_4G}): \text{offset}(\text{base}) \geq 0$$

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of xlat_idx_pe_aligned_TCC1.

Q.E.D.

C.114.4 Linear_Memory.xlat_idx_pe_aligned_TCC2

Terse proof for xlat_idx_pe_aligned_TCC2.

xlat_idx_pe_aligned_TCC2:

$$\{1\} \quad \forall (lvl: \text{Level}, \text{base}, \text{addr}: \text{Memory_Address_4G}):$$

$$\text{aligned?}(\text{bits_per_level} + \text{pe_size})(\text{offset}(\text{base})) \supset$$

$$\text{offset}(\text{xlat_idx}(lvl, \text{base}, \text{addr})) \geq 0$$

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of xlat_idx_pe_aligned_TCC2.

Q.E.D.

C.114.5 Linear_Memory.xlat_idx_pe_aligned

Terse proof for xlat_idx_pe_aligned.

xlat_idx_pe_aligned:

$$\{1\} \quad \forall (lvl: \text{Level}, \text{base}, \text{addr}: \text{Memory_Address_4G}):$$

$$\text{aligned?}(\text{bits_per_level} + \text{pe_size})(\text{offset}(\text{base})) \supset$$

$$\text{aligned?}(\text{pe_size})(\text{offset}(\text{xlat_idx}(lvl, \text{base}, \text{addr})))$$

Repeatedly Skolemizing and flattening,

Expanding the definition of xlat_idx,

Installing automatic rewrites from: (bus_width! bits_per_level! max_level!)

Case splitting on $-1 * (\text{bits_per_level} * lvl!1) - \text{bits_per_level} + \text{bus_width} \geq 0$,

we get 2 subgoals:

C Proof scripts

xlat_idx_pe_aligned.1:

{-1}	$-1 \times (\text{bits_per_level} \times \text{lvl}') - \text{bits_per_level} + \text{bus_width} \geq 0$
{-2}	$\text{lvl}' < \text{max_level}$
{-3}	$\text{Mem?}(\text{base}'\text{'type_of})$
{-4}	$0 \leq \text{base}'\text{'offset}$
{-5}	$\text{base}'\text{'offset} < \text{max_linear_offset}$
{-6}	$\text{Mem?}(\text{addr}'\text{'type_of})$
{-7}	$0 \leq \text{addr}'\text{'offset}$
{-8}	$\text{addr}'\text{'offset} < \text{max_linear_offset}$
{-9}	$\text{aligned?}(\text{bits_per_level} + \text{pe_size})(\text{offset}(\text{base}'))$
{1}	$\text{aligned?}(\text{pe_size})$ $(\text{offset}(\text{base}' + \text{shift_bits_left}(\text{cut_bits}(\text{offset}(\text{addr}')), -1 \times \text{bits_per_level} \times \text{lvl}' - \text{bits_per_level}))$

Using lemma aligned_shift_bits_ok,

we get 2 subgoals:

xlat_idx_pe_aligned.1.1:

{-1}	$\text{pe_size} \leq \text{pe_size} \supset$ $\text{aligned?}(\text{pe_size})$ $(\text{shift_bits_left}(\text{cut_bits}(\text{offset}(\text{addr}')),$ $\text{bus_width} - (\text{lvl}' + 1) \times \text{bits_per_level},$ $\text{bits_per_level}),$ $\text{pe_size}))$
{-2}	$-1 \times (\text{bits_per_level} \times \text{lvl}') - \text{bits_per_level} + \text{bus_width} \geq 0$
{-3}	$\text{lvl}' < \text{max_level}$
{-4}	$\text{Mem?}(\text{base}'\text{'type_of})$
{-5}	$0 \leq \text{base}'\text{'offset}$
{-6}	$\text{base}'\text{'offset} < \text{max_linear_offset}$
{-7}	$\text{Mem?}(\text{addr}'\text{'type_of})$
{-8}	$0 \leq \text{addr}'\text{'offset}$
{-9}	$\text{addr}'\text{'offset} < \text{max_linear_offset}$
{-10}	$\text{aligned?}(\text{bits_per_level} + \text{pe_size})(\text{offset}(\text{base}'))$
{1}	$\text{aligned?}(\text{pe_size})$ $(\text{offset}(\text{base}' + \text{shift_bits_left}(\text{cut_bits}(\text{offset}(\text{addr}')), -1 \times \text{bits_per_level} \times \text{lvl}' - \text{bits_per_level}))$

Expanding the definition of +,

Using lemma aligned_plus,

we get 4 subgoals:

xlat_idx_pe_aligned.1.1.1:

{-1}	aligned?(min(bits_per_level + pe_size, pe_size)) (offset(base') + shift_bits_left(cut_bits(offset(addr'), bus_width - (lvl' + 1) × bits_per_level, bits_per_level),
{-2}	pe_size ≤ pe_size ⊃ aligned?(pe_size) (shift_bits_left(cut_bits(offset(addr'), bus_width - (lvl' + 1) × bits_per_level, bits_per_level), pe_size))
{-3}	-1 × (bits_per_level × lvl') - bits_per_level + bus_width ≥ 0
{-4}	lvl' < max_level
{-5}	Mem?(base' type_of)
{-6}	0 ≤ base' offset
{-7}	base' offset < max_linear_offset
{-8}	Mem?(addr' type_of)
{-9}	0 ≤ addr' offset
{-10}	addr' offset < max_linear_offset
{-11}	aligned?(bits_per_level + pe_size)(offset(base'))
{1}	aligned?(pe_size) (base' offset + shift_bits_left(cut_bits(offset(addr'), -1 × bits_per_level × lvl' - bits_per_level + bus_width,

Expanding the definition of min,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of xlat_idx_pe_aligned.1.1.1.

xlat_idx_pe_aligned.1.1.2:

{-1}	pe_size ≤ pe_size ⊃ aligned?(pe_size) (shift_bits_left(cut_bits(offset(addr'), bus_width - (lvl' + 1) × bits_per_level, bits_per_level), pe_size))
{-2}	-1 × (bits_per_level × lvl') - bits_per_level + bus_width ≥ 0
{-3}	lvl' < max_level
{-4}	Mem?(base' type_of)
{-5}	0 ≤ base' offset
{-6}	base' offset < max_linear_offset
{-7}	Mem?(addr' type_of)
{-8}	0 ≤ addr' offset
{-9}	addr' offset < max_linear_offset
{-10}	aligned?(bits_per_level + pe_size)(offset(base'))
{1}	aligned?(pe_size) (shift_bits_left(cut_bits(offset(addr'), 22 - 10 × lvl', 10), pe_size))
{2}	aligned?(pe_size) (base' offset + shift_bits_left(cut_bits(offset(addr'), -1 × bits_per_level × lvl' - bits_per_level + bus_width,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of xlat_idx_pe_aligned.1.1.2.

C Proof scripts

xlat_idx_pe_aligned.1.1.3:

{-1}	$pe_size \leq pe_size \supset$ $aligned?(pe_size)$ $(shift_bits_left(cut_bits(offset(addr'),$ $bus_width - (lvl' + 1) \times bits_per_level,$ $bits_per_level),$ $pe_size))$
{-2}	$-1 \times (bits_per_level \times lvl') - bits_per_level + bus_width \geq 0$
{-3}	$lvl' < max_level$
{-4}	$Mem?(base' \text{ 'type_of})$
{-5}	$0 \leq base' \text{ 'offset}$
{-6}	$base' \text{ 'offset} < max_linear_offset$
{-7}	$Mem?(addr' \text{ 'type_of})$
{-8}	$0 \leq addr' \text{ 'offset}$
{-9}	$addr' \text{ 'offset} < max_linear_offset$
{-10}	$aligned?(bits_per_level + pe_size)(offset(base'))$
{1}	$22 - 10 \times lvl' \geq 0$
{2}	$aligned?(pe_size)$ $(base' \text{ 'offset} + shift_bits_left(cut_bits(offset(addr'), -1 \times bits_per_level \times lvl' - bits_per_level))$

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of xlat_idx_pe_aligned.1.1.3.

xlat_idx_pe_aligned.1.1.4:

{-1}	$pe_size \leq pe_size \supset$ $aligned?(pe_size)$ $(shift_bits_left(cut_bits(offset(addr'),$ $bus_width - (lvl' + 1) \times bits_per_level,$ $bits_per_level),$ $pe_size))$
{-2}	$-1 \times (bits_per_level \times lvl') - bits_per_level + bus_width \geq 0$
{-3}	$lvl' < max_level$
{-4}	$Mem?(base' \text{ 'type_of})$
{-5}	$0 \leq base' \text{ 'offset}$
{-6}	$base' \text{ 'offset} < max_linear_offset$
{-7}	$Mem?(addr' \text{ 'type_of})$
{-8}	$0 \leq addr' \text{ 'offset}$
{-9}	$addr' \text{ 'offset} < max_linear_offset$
{-10}	$aligned?(bits_per_level + pe_size)(offset(base'))$
{1}	$aligned?(10 + pe_size)(offset(base'))$
{2}	$aligned?(pe_size)$ $(base' \text{ 'offset} + shift_bits_left(cut_bits(offset(addr'), -1 \times bits_per_level \times lvl' - bits_per_level))$

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of xlat_idx_pe_aligned.1.1.4.

xlat_idx_pe_aligned.1.2:

<pre> {-1} -1 × (bits_per_level × lvl') - bits_per_level + bus_width ≥ 0 {-2} lvl' < max_level {-3} Mem?(base' type_of) {-4} 0 ≤ base' offset {-5} base' offset < max_linear_offset {-6} Mem?(addr' type_of) {-7} 0 ≤ addr' offset {-8} addr' offset < max_linear_offset {-9} aligned?(bits_per_level + pe_size)(offset(base')) </pre>	<hr/> <pre> {1} 22 - 10 × lvl' ≥ 0 {2} aligned?(pe_size) (offset (base' + shift_bits_left(cut_bits(offset(addr')), -1 × bits_per_level × lvl' - bits_per_level + bus_wi </pre>
--	--

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of xlat_idx_pe_aligned.1.2.

xlat_idx_pe_aligned.2:

<pre> {-1} lvl' < max_level {-2} Mem?(base' type_of) {-3} 0 ≤ base' offset {-4} base' offset < max_linear_offset {-5} Mem?(addr' type_of) {-6} 0 ≤ addr' offset {-7} addr' offset < max_linear_offset {-8} aligned?(bits_per_level + pe_size)(offset(base')) </pre>	<hr/> <pre> {1} -1 × (bits_per_level × lvl') - bits_per_level + bus_width ≥ 0 {2} aligned?(pe_size) (offset (base' + shift_bits_left(cut_bits(offset(addr')), -1 × bits_per_level × lvl' - bits_per_level + bus_wi </pre>
--	---

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of xlat_idx_pe_aligned.2.
Q.E.D.

C.114.6 Linear_Memory.xlat_idx_memory_address

Terse proof for xlat_idx_memory_address.

xlat_idx_memory_address:

<pre> {1} ∀ (lvl: Level, base, addr: Memory_Address_4G): aligned?(bits_per_level + pe_size)(offset(base)) ⊃ xlat_idx(lvl, base, addr) < max_linear </pre>

Repeatedly Skolemizing and flattening,
Installing automatic rewrites from: Mem + < max_linear max_linear_offset bits_per_level pe_size
max_level bus_width

Expanding the definition of xlat_idx,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Using lemma aligned_add_below,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Rewriting using shift_bits_left_less, matching in *,

This completes the proof of `xlat_idx_memory_address`.
 Q.E.D.

C.114.7 Linear_Memory.xlat_idx_memory_address2

Terse proof for `xlat_idx_memory_address2`.

`xlat_idx_memory_address2`:

<pre>{1} ∀ (lvl: Level, base, addr: Memory_Address_4G): aligned?(bits_per_level + pe_size)(offset(base)) ⊃ 0 ≤ offset(xlat_idx(lvl, base, addr))</pre>

Expanding the definition of `xlat_idx`,
 Repeatedly Skolemizing and flattening,
 Adding type constraints for `shift_bits_left(cut_bits(offset(addr!1), -1 * (lvl!1 * bits_per_level) - bits_per_level + bus_width, bits_per_level), pe_size)`,
 we get 2 subgoals:

`xlat_idx_memory_address2.1`:

<pre>{-1} shift_bits_left(cut_bits(offset(addr'), -1 × (lvl' × bits_per_level) - bits_per_level + bus_width, bits_per_level), pe_size) ≥ 0 {-2} lvl' < max_level {-3} Mem?(base', type_of) {-4} 0 ≤ base' offset {-5} base' offset < max_linear_offset {-6} Mem?(addr', type_of) {-7} 0 ≤ addr' offset {-8} addr' offset < max_linear_offset {-9} aligned?(bits_per_level + pe_size)(offset(base'))</pre>
<pre>{1} 0 ≤ offset (base' + shift_bits_left(cut_bits(offset(addr'), -1 × lvl' × bits_per_level - bits_per_level + bus_w</pre>

Expanding the definition of `+`,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 This completes the proof of `xlat_idx_memory_address2.1`.

`xlat_idx_memory_address2.2`:

<pre>{-1} lvl' < max_level {-2} Mem?(base', type_of) {-3} 0 ≤ base' offset {-4} base' offset < max_linear_offset {-5} Mem?(addr', type_of) {-6} 0 ≤ addr' offset {-7} addr' offset < max_linear_offset {-8} aligned?(bits_per_level + pe_size)(offset(base'))</pre>
<pre>{1} -1 × (lvl' × bits_per_level) - bits_per_level + bus_width ≥ 0 {2} 0 ≤ offset (base' + shift_bits_left(cut_bits(offset(addr'), -1 × lvl' × bits_per_level - bits_per_level + bus_w</pre>

Keeping (-1 1) and hiding *,
 Trying repeated skolemization, instantiation, and if-lifting,
 Expanding the definition of bus_width,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 This completes the proof of xlat_idx_memory_address2.2.
 Q.E.D.

C.114.8 Linear_Memory.xlat_ofs_memory_address

Terse proof for xlat_ofs_memory_address.

xlat_ofs_memory_address:

$$\{1\} \quad \forall (lvl: \text{Level}, \text{base}, \text{addr}: \text{Memory_Address_4G}):$$

$$\text{aligned?}(\text{bus_width} - (lvl + 1) \times \text{bits_per_level})(\text{offset}(\text{base})) \supset$$

$$\text{xlat_ofs}(lvl, \text{base}, \text{addr}) < \text{max_linear}$$

Installing automatic rewrites from: Mem + < max_linear max_linear_offset bits_per_level pe_size
 max_level bus_width

Repeatedly Skolemizing and flattening,
 Expanding the definition of xlat_ofs,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 Using lemma aligned_add_below,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 This completes the proof of xlat_ofs_memory_address.
 Q.E.D.

C.114.9 Linear_Memory.xlat_ofs_memory_address2

Terse proof for xlat_ofs_memory_address2.

xlat_ofs_memory_address2:

$$\{1\} \quad \forall (lvl: \text{Level}, \text{base}, \text{addr}: \text{Memory_Address_4G}):$$

$$\text{aligned?}(\text{bus_width} - (lvl + 1) \times \text{bits_per_level})(\text{offset}(\text{base})) \supset$$

$$0 \leq \text{offset}(\text{xlat_ofs}(lvl, \text{base}, \text{addr}))$$

Expanding the definition of xlat_ofs,
 Expanding the definition of +,
 Repeatedly Skolemizing and flattening,
 Using lemma cut_bits_below,
 we get 2 subgoals:

xlat_ofs_memory_address2.1:

<pre> {-1} cut_bits(offset(addr'), 0, -1 × lvl' × bits_per_level - bits_per_level + bus_width) < expt(2, -1 × lvl' × bits_per_level - bits_per_level + bus_width) {-2} lvl' < max_level {-3} Mem?(base' 'type_of) {-4} 0 ≤ base' 'offset {-5} base' 'offset < max_linear_offset {-6} Mem?(addr' 'type_of) {-7} 0 ≤ addr' 'offset {-8} addr' 'offset < max_linear_offset {-9} aligned?(-1 × lvl' × bits_per_level - bits_per_level + bus_width) (offset(base')) </pre>	<hr style="border: 0.5px solid black;"/> <pre> {1} 0 ≤ base' 'offset + cut_bits(offset(addr'), 0, -1 × lvl' × bits_per_level - bits_per_level + bus_width) </pre>
--	--

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of xlat_ofs_memory_address2.1.

xlat_ofs_memory_address2.2:

<pre> {-1} lvl' < max_level {-2} Mem?(base' 'type_of) {-3} 0 ≤ base' 'offset {-4} base' 'offset < max_linear_offset {-5} Mem?(addr' 'type_of) {-6} 0 ≤ addr' 'offset {-7} addr' 'offset < max_linear_offset {-8} aligned?(-1 × lvl' × bits_per_level - bits_per_level + bus_width) (offset(base')) </pre>	<hr style="border: 0.5px solid black;"/> <pre> {1} -1 × lvl' × bits_per_level - bits_per_level + bus_width ≥ 0 {2} 0 ≤ base' 'offset + cut_bits(offset(addr'), 0, -1 × lvl' × bits_per_level - bits_per_level + bus_width) </pre>
--	---

Keeping (-1 1) and hiding *,
Installing automatic rewrites from: bus_width
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of xlat_ofs_memory_address2.2.
Q.E.D.

C.114.10 Linear_Memory.raise_fault_TCC1

Terse proof for raise_fault_TCC1.

raise_fault_TCC1:

<pre> {1} ∀ (access: Memory_access, pfa: Memory_Address_4G): (Read ∈ (singleton[Memory_access](access) ∪ {Read})) </pre>	<hr style="border: 0.5px solid black;"/>
---	--

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of raise_fault_TCC1.
Q.E.D.

C.114.11 Linear_Memory.translate_TCC1

Terse proof for `translate_TCC1`.

`translate_TCC1`:

$\{1\} \quad \forall (lvl: \text{Level}, \text{base}, \text{addr}: \text{Memory_Address_4G}, \text{access}: \text{Memory_access}, \text{priv}: \text{Memory_privilege}, \text{pe_addr}: \text{Address}):$ $\text{pe_addr} = \text{xlat_idx}(lvl, \text{base}, \text{addr}) \supset$ $(\forall (\text{pe}: (\text{range_pt}(lvl))):$ $\text{accessible}?(pe, \text{access}) \wedge$ $\text{privileged}(pe, \text{priv}) \wedge \text{present}(pe) \wedge \text{paging_type}(lvl, pe)$ $\supset \text{range_pt}(lvl)(\text{set_reference}(pe, \text{access})))$
--

Repeatedly Skolemizing and flattening,

Rewriting using `set_reference_range`, matching in `*`,

This completes the proof of `translate_TCC1`.

Q.E.D.

C.114.12 Linear_Memory.translate_memory_address

Terse proof for `translate_memory_address`.

`translate_memory_address`:

$\{1\} \quad \forall (lvl: \text{Level}, \text{base}, \text{addr}: \text{Memory_Address_4G}, \text{access}: \text{Memory_access}, \text{priv}: \text{Memory_privilege},$ $s: \text{Linear_memory}):$ $\text{aligned}?(bits_per_level + pe_size)(\text{offset}(\text{base})) \wedge$ $\text{OK}?(translate(lvl, \text{base}, \text{addr}, \text{access}, \text{priv})(s))$ $\supset \text{data}(translate(lvl, \text{base}, \text{addr}, \text{access}, \text{priv})(s))^2 < \text{max_linear}$

Expanding the definition of `translate`,

Repeatedly Skolemizing and flattening,

Installing automatic rewrites from: (`##!` `<! ok_result!` `exception_result!` `fatal_result!` `raise_fault!` `max_linear!` `Mem!`)

Case splitting on `OK?(read_data(pm, paging_data_type(lvl!1)) (xlat_idx(lvl!1, base!1, addr!1))(s!1))`
`AND present?(data(read_data(pm, paging_data_type(lvl!1)) (xlat_idx(lvl!1, base!1, addr!1))(s!1)))`,

we get 2 subgoals:

translate_memory_address.1:

```

{-1} OK?(read_data(pm, paging_data_type(lvl'))(xlat_idx(lvl', base', addr'))(s')) ∧
      present?(data(read_data(pm, paging_data_type(lvl'))(xlat_idx(lvl', base', addr'))
                    (s')))
{-2} lvl' < max_level
{-3} Mem?(base' type_of)
{-4} 0 ≤ base' offset
{-5} base' offset < max_linear_offset
{-6} Mem?(addr' type_of)
{-7} 0 ≤ addr' offset
{-8} addr' offset < max_linear_offset
{-9} aligned?(bits_per_level + pe_size)(offset(base'))
{-10} OK?((read_data(pm, paging_data_type(lvl'))(xlat_idx(lvl', base', addr')) ##
          (λ (pe: (range_pt(lvl'))):
            IF paging_type?(lvl', pe)
              THEN IF present?(pe)
                THEN IF accessible?(pe, access') ∧ privileged?(pe, priv')
                  THEN write_data(pm, paging_data_type(lvl'))
                        (xlat_idx(lvl', base', addr'),
                          set_reference(pe, access'))
                  ## ok_result(is_leaf?(pe), base(pe))
                  ELSE raise_fault(priv', access', TRUE, addr')
                ENDIF
              ELSE raise_fault(priv', access', FALSE, addr')
            ENDIF
          ELSE fatal_result
          ENDIF))
      (s'))
-----
{1} data((read_data(pm, paging_data_type(lvl'))(xlat_idx(lvl', base', addr')) ##
        (λ (pe: (range_pt(lvl'))):
          IF paging_type?(lvl', pe)
            THEN IF present?(pe)
              THEN IF accessible?(pe, access') ∧ privileged?(pe, priv')
                THEN write_data(pm, paging_data_type(lvl'))
                      (xlat_idx(lvl', base', addr'),
                        set_reference(pe, access'))
                ## ok_result(is_leaf?(pe), base(pe))
              ELSE raise_fault(priv', access', TRUE, addr')
            ENDIF
          ELSE raise_fault(priv', access', FALSE, addr')
          ENDIF
        ELSE fatal_result
        ENDIF))
      (s'))'2
      < max_linear

```

Adding type constraints for base(data(read_data(pm, paging_data_type(lvl!1)) (xlat_idx(lvl!1, base!1, addr!1))(s!1))),

we get 2 subgoals:

translate_memory_address.1.1:

```

{-1} Mem?(base(data(read_data(pm, paging_data_type(lvl'))(xlat_idx(lvl', base', addr'))
                    (s')))'type_of)
{-2} 0 ≤
      base(data(read_data(pm, paging_data_type(lvl'))(xlat_idx(lvl', base', addr'))
                    (s')))'offset
{-3} base(data(read_data(pm, paging_data_type(lvl'))(xlat_idx(lvl', base', addr'))
                    (s')))'offset
      < max_linear_offset
{-4} OK?(read_data(pm, paging_data_type(lvl'))(xlat_idx(lvl', base', addr'))(s')) ∧
      present?(data(read_data(pm, paging_data_type(lvl'))(xlat_idx(lvl', base', addr'))
                    (s'))))
{-5} lvl' < max_level
{-6} Mem?(base' 'type_of)
{-7} 0 ≤ base' 'offset
{-8} base' 'offset < max_linear_offset
{-9} Mem?(addr' 'type_of)
{-10} 0 ≤ addr' 'offset
{-11} addr' 'offset < max_linear_offset
{-12} aligned?(bits_per_level + pe_size)(offset(base'))
{-13} OK?((read_data(pm, paging_data_type(lvl'))(xlat_idx(lvl', base', addr')) ##
          (λ (pe: (range_pt(lvl'))):
            IF paging_type?(lvl', pe)
              THEN IF present?(pe)
                THEN IF accessible?(pe, access') ∧ privileged?(pe, priv')
                  THEN write_data(pm, paging_data_type(lvl'))
                        (xlat_idx(lvl', base', addr'),
                          set_reference(pe, access'))
                  ## ok_result(is_leaf?(pe), base(pe))
                ELSE raise_fault(priv', access', TRUE, addr')
              ENDIF
            ELSE raise_fault(priv', access', FALSE, addr')
            ENDIF
          ELSE fatal_result
          ENDIF))
      (s'))
-----
{1} data((read_data(pm, paging_data_type(lvl'))(xlat_idx(lvl', base', addr')) ##
          (λ (pe: (range_pt(lvl'))):
            IF paging_type?(lvl', pe)
              THEN IF present?(pe)
                THEN IF accessible?(pe, access') ∧ privileged?(pe, priv')
                  THEN write_data(pm, paging_data_type(lvl'))
                        (xlat_idx(lvl', base', addr'),
                          set_reference(pe, access'))
                  ## ok_result(is_leaf?(pe), base(pe))
                ELSE raise_fault(priv', access', TRUE, addr')
              ENDIF
            ELSE raise_fault(priv', access', FALSE, addr')
            ENDIF
          ELSE fatal_result
          ENDIF))
      (s'))'2
      < max_linear

```

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `translate_memory_address.1.1`.

`translate_memory_address.1.2`:

```

{-1} OK?(read_data(pm, paging_data_type(lvl'))(xlat_idx(lvl', base', addr'))(s')) ∧
      present?(data(read_data(pm, paging_data_type(lvl'))(xlat_idx(lvl', base', addr'))
                    (s')))
{-2} lvl' < max_level
{-3} Mem?(base' type_of)
{-4} 0 ≤ base' offset
{-5} base' offset < max_linear_offset
{-6} Mem?(addr' type_of)
{-7} 0 ≤ addr' offset
{-8} addr' offset < max_linear_offset
{-9} aligned?(bits_per_level + pe_size)(offset(base'))
{-10} OK?((read_data(pm, paging_data_type(lvl'))(xlat_idx(lvl', base', addr')) ##
          (λ (pe: (range_pt(lvl'))):
            IF paging_type?(lvl', pe)
              THEN IF present?(pe)
                THEN IF accessible?(pe, access') ∧ privileged?(pe, priv')
                  THEN write_data(pm, paging_data_type(lvl'))
                        (xlat_idx(lvl', base', addr'),
                          set_reference(pe, access'))
                  ## ok_result(is_leaf?(pe), base(pe))
                ELSE raise_fault(priv', access', TRUE, addr')
              ENDIF
            ELSE raise_fault(priv', access', FALSE, addr')
            ENDIF
          ELSE fatal_result
          ENDIF))
      (s'))
-----
{1} present?(data[Physical_memory, ((range_pt(lvl')))]
          (read_data[Physical_memory, ((range_pt(lvl')))]
            (pm, paging_data_type(lvl'))(xlat_idx(lvl', base', addr'))
            (s')))
{2} data((read_data(pm, paging_data_type(lvl'))(xlat_idx(lvl', base', addr')) ##
          (λ (pe: (range_pt(lvl'))):
            IF paging_type?(lvl', pe)
              THEN IF present?(pe)
                THEN IF accessible?(pe, access') ∧ privileged?(pe, priv')
                  THEN write_data(pm, paging_data_type(lvl'))
                        (xlat_idx(lvl', base', addr'),
                          set_reference(pe, access'))
                  ## ok_result(is_leaf?(pe), base(pe))
                ELSE raise_fault(priv', access', TRUE, addr')
              ENDIF
            ELSE raise_fault(priv', access', FALSE, addr')
            ENDIF
          ELSE fatal_result
          ENDIF))
      (s'))'2
      < max_linear

```

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `translate_memory_address.1.2`.

`translate_memory_address.2`:

```

{-1}  lvl' < max_level
{-2}  Mem?(base' 'type_of)
{-3}  0 ≤ base' 'offset
{-4}  base' 'offset < max_linear_offset
{-5}  Mem?(addr' 'type_of)
{-6}  0 ≤ addr' 'offset
{-7}  addr' 'offset < max_linear_offset
{-8}  aligned?(bits_per_level + pe_size)(offset(base'))
{-9}  OK?((read_data(pm, paging_data_type(lvl'))(xlat_idx(lvl', base', addr'))) ##
        (λ (pe: (range_pt(lvl'))):
          IF paging_type?(lvl', pe)
            THEN IF present?(pe)
              THEN IF accessible?(pe, access') ∧ privileged?(pe, priv')
                THEN write_data(pm, paging_data_type(lvl'))
                      (xlat_idx(lvl', base', addr')),
                      set_reference(pe, access'))
                ## ok_result(is_leaf?(pe), base(pe))
              ELSE raise_fault(priv', access', TRUE, addr')
            ENDIF
          ELSE raise_fault(priv', access', FALSE, addr')
          ENDIF
        ELSE fatal_result
        ENDIF))
        (s'))
-----
{1}   OK?(read_data(pm, paging_data_type(lvl'))(xlat_idx(lvl', base', addr'))(s')) ∧
      present?(data(read_data(pm, paging_data_type(lvl'))(xlat_idx(lvl', base', addr'))
                    (s')))
{2}   data((read_data(pm, paging_data_type(lvl'))(xlat_idx(lvl', base', addr'))) ##
        (λ (pe: (range_pt(lvl'))):
          IF paging_type?(lvl', pe)
            THEN IF present?(pe)
              THEN IF accessible?(pe, access') ∧ privileged?(pe, priv')
                THEN write_data(pm, paging_data_type(lvl'))
                      (xlat_idx(lvl', base', addr')),
                      set_reference(pe, access'))
                ## ok_result(is_leaf?(pe), base(pe))
              ELSE raise_fault(priv', access', TRUE, addr')
            ENDIF
          ELSE raise_fault(priv', access', FALSE, addr')
          ENDIF
        ELSE fatal_result
        ENDIF))
        (s'))'2
      < max_linear

```

Hiding formulas: 2,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `translate_memory_address.2`.

Q.E.D.

C.114.13 Linear_Memory.translate_memory_address2

Terse proof for translate_memory_address2.

translate_memory_address2:

$\{1\} \quad \forall (lvl: \text{Level}, \text{base}, \text{addr}: \text{Memory_Address_4G}, \text{access}: \text{Memory_access}, \text{priv}: \text{Memory_privilege}, \\ s: \text{Linear_memory}): \\ \text{aligned?}(\text{bits_per_level} + \text{pe_size})(\text{offset}(\text{base})) \wedge \\ \text{OK?}(\text{translate}(lvl, \text{base}, \text{addr}, \text{access}, \text{priv})(s)) \\ \supset 0 \leq \text{offset}(\text{data}(\text{translate}(lvl, \text{base}, \text{addr}, \text{access}, \text{priv})(s)))^2$
--

Expanding the definition of translate,

Repeatedly Skolemizing and flattening,

Installing automatic rewrites from: (### <! ok_result! exception_result! fatal_result! raise_fault!
max_linear! Mem!)

Case splitting on $\text{OK?}(\text{read_data}(\text{pm}, \text{paging_data_type}(lvl!1)) (\text{xlat_idx}(lvl!1, \text{base}!1, \text{addr}!1))(s!1))$
AND $\text{present?}(\text{data}(\text{read_data}(\text{pm}, \text{paging_data_type}(lvl!1)) (\text{xlat_idx}(lvl!1, \text{base}!1, \text{addr}!1))(s!1)))$,

we get 2 subgoals:

translate_memory_address2.1:

```

{-1} OK?(read_data(pm, paging_data_type(lvl'))(xlat_idx(lvl', base', addr'))(s')) ∧
      present?(data(read_data(pm, paging_data_type(lvl'))(xlat_idx(lvl', base', addr'))
                    (s')))
{-2} lvl' < max_level
{-3} Mem?(base' type_of)
{-4} 0 ≤ base' offset
{-5} base' offset < max_linear_offset
{-6} Mem?(addr' type_of)
{-7} 0 ≤ addr' offset
{-8} addr' offset < max_linear_offset
{-9} aligned?(bits_per_level + pe_size)(offset(base'))
{-10} OK?((read_data(pm, paging_data_type(lvl'))(xlat_idx(lvl', base', addr'))) ##
          (λ (pe: (range_pt(lvl'))):
            IF paging_type?(lvl', pe)
              THEN IF present?(pe)
                THEN IF accessible?(pe, access') ∧ privileged?(pe, priv')
                  THEN write_data(pm, paging_data_type(lvl'))
                        (xlat_idx(lvl', base', addr'),
                         set_reference(pe, access'))
                  ## ok_result(is_leaf?(pe), base(pe))
                ELSE raise_fault(priv', access', TRUE, addr')
              ENDIF
            ELSE raise_fault(priv', access', FALSE, addr')
            ENDIF
          ELSE fatal_result
          ENDIF))
      (s'))
-----
{1} 0 ≤
     offset
     (data((read_data(pm, paging_data_type(lvl'))(xlat_idx(lvl', base', addr'))) ##
           (λ (pe: (range_pt(lvl'))):
             IF paging_type?(lvl', pe)
               THEN IF present?(pe)
                 THEN IF accessible?(pe, access') ∧ privi-
leged?(pe, priv')
                   THEN write_data(pm, paging_data_type(lvl'))
                         (xlat_idx(lvl', base', addr'),
                          set_reference(pe, access'))
                   ## ok_result(is_leaf?(pe), base(pe))
                 ELSE raise_fault(priv', access', TRUE, addr')
               ENDIF
             ELSE raise_fault(priv', access', FALSE, addr')
             ENDIF
           ELSE fatal_result
           ENDIF))
     (s'))'2)

```

Adding type constraints for base(data(read_data(pm, paging_data_type(lvl!1)) (xlat_idx(lvl!1, base!1, addr!1))(s!1))),

we get 2 subgoals:

```
translate_memory_address2.1.1:
```

```
{-1} Mem?(base(data(read_data(pm, paging_data_type(lvl'))(xlat_idx(lvl', base', addr'))
(s')))'type_of)
{-2} 0 ≤
base(data(read_data(pm, paging_data_type(lvl'))(xlat_idx(lvl', base', addr'))
(s')))'offset
{-3} base(data(read_data(pm, paging_data_type(lvl'))(xlat_idx(lvl', base', addr'))
(s')))'offset
< max_linear_offset
{-4} OK?(read_data(pm, paging_data_type(lvl'))(xlat_idx(lvl', base', addr'))(s')) ∧
present?(data(read_data(pm, paging_data_type(lvl'))(xlat_idx(lvl', base', addr'))
(s'))))
{-5} lvl' < max_level
{-6} Mem?(base' 'type_of)
{-7} 0 ≤ base' 'offset
{-8} base' 'offset < max_linear_offset
{-9} Mem?(addr' 'type_of)
{-10} 0 ≤ addr' 'offset
{-11} addr' 'offset < max_linear_offset
{-12} aligned?(bits_per_level + pe_size)(offset(base'))
{-13} OK?((read_data(pm, paging_data_type(lvl'))(xlat_idx(lvl', base', addr')) ##
(λ (pe: (range_pt(lvl'))):
IF paging_type?(lvl', pe)
THEN IF present?(pe)
THEN IF accessible?(pe, access') ∧ privileged?(pe, priv')
THEN write_data(pm, paging_data_type(lvl'))
(xlat_idx(lvl', base', addr')),
set_reference(pe, access'))
## ok_result(is_leaf?(pe), base(pe))
ELSE raise_fault(priv', access', TRUE, addr')
ENDIF
ELSE raise_fault(priv', access', FALSE, addr')
ENDIF
ELSE fatal_result
ENDIF))
(s'))
```

```
{1} 0 ≤
offset
(data((read_data(pm, paging_data_type(lvl'))(xlat_idx(lvl', base', addr')) ##
(λ (pe: (range_pt(lvl'))):
IF paging_type?(lvl', pe)
THEN IF present?(pe)
THEN IF accessible?(pe, access') ∧ privi-
leged?(pe, priv')
THEN write_data(pm, paging_data_type(lvl'))
(xlat_idx(lvl', base', addr')),
set_reference(pe, access'))
## ok_result(is_leaf?(pe), base(pe))
ELSE raise_fault(priv', access', TRUE, addr')
ENDIF
ELSE raise_fault(priv', access', FALSE, addr')
ENDIF
ELSE fatal_result
ENDIF))
(s'))'2)
```

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `translate_memory_address2.1.1`.

translate_memory_address2.1.2:

```

{-1} OK?(read_data(pm, paging_data_type(lvl'))(xlat_idx(lvl', base', addr'))(s')) ∧
      present?(data(read_data(pm, paging_data_type(lvl'))(xlat_idx(lvl', base', addr'))
                    (s')))
{-2} lvl' < max_level
{-3} Mem?(base' type_of)
{-4} 0 ≤ base' offset
{-5} base' offset < max_linear_offset
{-6} Mem?(addr' type_of)
{-7} 0 ≤ addr' offset
{-8} addr' offset < max_linear_offset
{-9} aligned?(bits_per_level + pe_size)(offset(base'))
{-10} OK?((read_data(pm, paging_data_type(lvl'))(xlat_idx(lvl', base', addr'))) ##
          (λ (pe: (range_pt(lvl'))):
            IF paging_type?(lvl', pe)
            THEN IF present?(pe)
                  THEN IF accessible?(pe, access') ∧ privileged?(pe, priv')
                        THEN write_data(pm, paging_data_type(lvl'))
                               (xlat_idx(lvl', base', addr')),
                               set_reference(pe, access'))
                        ## ok_result(is_leaf?(pe), base(pe))
                  ELSE raise_fault(priv', access', TRUE, addr')
            ENDIF
            ELSE raise_fault(priv', access', FALSE, addr')
            ENDIF
          ELSE fatal_result
          ENDIF))
      (s'))
-----
{1} present?(data[Physical_memory, ((range_pt(lvl')))]
           (read_data[Physical_memory, ((range_pt(lvl')))]
             (pm, paging_data_type(lvl'))(xlat_idx(lvl', base', addr'))
             (s')))
{2} 0 ≤
     offset
     (data((read_data(pm, paging_data_type(lvl'))(xlat_idx(lvl', base', addr'))) ##
           (λ (pe: (range_pt(lvl'))):
             IF paging_type?(lvl', pe)
             THEN IF present?(pe)
                   THEN IF accessible?(pe, access') ∧ privi-
leged?(pe, priv')
                         THEN write_data(pm, paging_data_type(lvl'))
                                (xlat_idx(lvl', base', addr')),
                                set_reference(pe, access'))
                         ## ok_result(is_leaf?(pe), base(pe))
                   ELSE raise_fault(priv', access', TRUE, addr')
             ENDIF
             ELSE raise_fault(priv', access', FALSE, addr')
             ENDIF
           ELSE fatal_result
           ENDIF))
     (s'))'2)

```

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `translate_memory_address2.1.2`.

`translate_memory_address2.2`:

{-1}	<code>lvl' < max_level</code>
{-2}	<code>Mem?(base' type_of)</code>
{-3}	<code>0 ≤ base' offset</code>
{-4}	<code>base' offset < max_linear_offset</code>
{-5}	<code>Mem?(addr' type_of)</code>
{-6}	<code>0 ≤ addr' offset</code>
{-7}	<code>addr' offset < max_linear_offset</code>
{-8}	<code>aligned?(bits_per_level + pe_size)(offset(base'))</code>
{-9}	<code>OK?((read_data(pm, paging_data_type(lvl'))(xlat_idx(lvl', base', addr')) ##</code> <code> (λ (pe: (range_pt(lvl'))):</code> <code> IF paging_type?(lvl', pe)</code> <code> THEN IF present?(pe)</code> <code> THEN IF accessible?(pe, access') ∧ privileged?(pe, priv')</code> <code> THEN write_data(pm, paging_data_type(lvl'))</code> <code> (xlat_idx(lvl', base', addr'),</code> <code> set_reference(pe, access'))</code> <code> ## ok_result(is_leaf?(pe), base(pe))</code> <code> ELSE raise_fault(priv', access', TRUE, addr')</code> <code> ENDIF</code> <code> ELSE raise_fault(priv', access', FALSE, addr')</code> <code> ENDIF</code> <code> ELSE fatal_result</code> <code> ENDIF))</code> <code> (s')</code>
<hr/>	
{1}	<code>OK?(read_data(pm, paging_data_type(lvl'))(xlat_idx(lvl', base', addr'))(s')) ∧</code> <code> present?(data(read_data(pm, paging_data_type(lvl'))(xlat_idx(lvl', base', addr'))</code> <code> (s'))</code>
{2}	<code>0 ≤</code> <code> offset</code> <code> (data((read_data(pm, paging_data_type(lvl'))(xlat_idx(lvl', base', addr')) ##</code> <code> (λ (pe: (range_pt(lvl'))):</code> <code> IF paging_type?(lvl', pe)</code> <code> THEN IF present?(pe)</code> <code> THEN IF accessible?(pe, access') ∧ privi-</code> <code> leged?(pe, priv')</code> <code> THEN write_data(pm, paging_data_type(lvl'))</code> <code> (xlat_idx(lvl', base', addr'),</code> <code> set_reference(pe, access'))</code> <code> ## ok_result(is_leaf?(pe), base(pe))</code> <code> ELSE raise_fault(priv', access', TRUE, addr')</code> <code> ENDIF</code> <code> ELSE raise_fault(priv', access', FALSE, addr')</code> <code> ENDIF</code> <code> ELSE fatal_result</code> <code> ENDIF))</code> <code> (s'))'2</code>

Hiding formulas: 2,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `translate_memory_address2.2`.

Q.E.D.

C.114.14 Linear_Memory.translate_result_leaf_TCC1

Terse proof for `translate_result_leaf_TCC1`.

`translate_result_leaf_TCC1`:

```
{1}  ∀ (lvl: Level, base, addr: Memory_Address_4G, access: Mem-
      ory_access, priv: Memory_privilege,
      s: Linear_memory):
      aligned?(bits_per_level + pe_size)(offset(base)) ∧
      OK?(translate(lvl, base, addr, access, priv)(s)) ∧
      data(translate(lvl, base, addr, access, priv)(s))'1
      ⊃
      offset
      (data[Physical_memory, [bool, Address]]
       (translate(lvl, base, addr, access, priv)(s))'2)
      ≥ 0
```

Repeatedly Skolemizing and flattening,
 Using lemma `translate_memory_address2`,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 This completes the proof of `translate_result_leaf_TCC1`.
 Q.E.D.

C.114.15 Linear_Memory.translate_result_leaf_TCC2

Terse proof for `translate_result_leaf_TCC2`.

`translate_result_leaf_TCC2`:

```
{1}  ∀ (lvl: Level, base, addr: Memory_Address_4G, access: Mem-
      ory_access, priv: Memory_privilege,
      s: Linear_memory):
      aligned?(bits_per_level + pe_size)(offset(base)) ∧
      OK?(translate(lvl, base, addr, access, priv)(s)) ∧
      data(translate(lvl, base, addr, access, priv)(s))'1
      ⊃ bus_width - bits_per_level × (lvl + 1) ≥ 0
```

Repeatedly Skolemizing and flattening,
 Keeping (-1 1) and hiding *,
 Installing automatic rewrites from: `bus_width`
 Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of `translate_result_leaf_TCC2`.
 Q.E.D.

C.114.16 Linear_Memory.translate_result_leaf

Terse proof for `translate_result_leaf`.

translate_result_leaf:

```
{1}  ∀ (lvl: Level, base, addr: Memory_Address_4G, access: Mem-
      ory_access, priv: Memory_privilege,
      s: Linear_memory):
      aligned?(bits_per_level + pe_size)(offset(base)) ∧
      OK?(translate(lvl, base, addr, access, priv)(s)) ∧
      data(translate(lvl, base, addr, access, priv)(s))'1
      ⊃
      aligned?(bus_width - bits_per_level × (lvl + 1))
        (offset(data(translate(lvl, base, addr, access, priv)(s))'2))
```

Repeatedly Skolemizing and flattening,

Expanding the definition of translate,

Installing automatic rewrites from: (##! ok_result! exception_result! fatal_result! raise_fault!
is_leaf?! bus_width! pe_size! bits_per_level! base! paging_tape?! pdir_entry?! ptab_entry?! max_level!)

Expanding the definition of paging_type?,

Expanding the definition of pdir_entry?,

Expanding the definition of ptab_entry?,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of translate_result_leaf.

Q.E.D.

C.114.17 Linear_Memory.translate_result_no_leaf_TCC1

Terse proof for translate_result_no_leaf_TCC1.

translate_result_no_leaf_TCC1:

```
{1}  ∀ (lvl: Level, base, addr: Memory_Address_4G, access: Mem-
      ory_access, priv: Memory_privilege,
      s: Linear_memory):
      aligned?(bits_per_level + pe_size)(offset(base)) ∧
      OK?(translate(lvl, base, addr, access, priv)(s)) ∧
      ¬ data(translate(lvl, base, addr, access, priv)(s))'1
      ⊃
      offset
        (data[Physical_memory, [bool, Address]]
          (translate(lvl, base, addr, access, priv)(s))'2)
      ≥ 0
```

Repeatedly Skolemizing and flattening,

Using lemma translate_memory_address2,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of translate_result_no_leaf_TCC1.

Q.E.D.

C.114.18 Linear_Memory.translate_result_no_leaf

Terse proof for translate_result_no_leaf.

`translate_result_no_leaf:`

$\{1\} \quad \forall (lvl: \text{Level}, \text{base}, \text{addr}: \text{Memory_Address_4G}, \text{access}: \text{Memory_access}, \text{priv}: \text{Memory_privilege}, \\ s: \text{Linear_memory}): \\ \text{aligned?}(\text{bits_per_level} + \text{pe_size})(\text{offset}(\text{base})) \wedge \\ \text{OK?}(\text{translate}(lvl, \text{base}, \text{addr}, \text{access}, \text{priv})(s)) \wedge \\ \neg \text{data}(\text{translate}(lvl, \text{base}, \text{addr}, \text{access}, \text{priv})(s))'1 \\ \supset \\ \text{aligned?}(\text{bits_per_level} + \text{pe_size}) \\ (\text{offset}(\text{data}(\text{translate}(lvl, \text{base}, \text{addr}, \text{access}, \text{priv})(s))'2))$

Repeatedly Skolemizing and flattening,

Expanding the definition of `translate`,

Installing automatic rewrites from: (`##!` `ok_result!` `exception_result!` `fatal_result!` `raise_fault!` `is_leaf?!` `bus_width!` `pe_size!` `bits_per_level!` `base!` `paging_tape?!` `pdir_entry?!` `ptab_entry?!` `max_level!`)

Expanding the definition of `paging_type?`,

Expanding the definition of `pdir_entry?`,

Expanding the definition of `ptab_entry?`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `translate_result_no_leaf`.

Q.E.D.

C.114.19 Linear_Memory.linear_resolve_TCC1

Terse proof for `linear_resolve_TCC1`.

`linear_resolve_TCC1:`

$\{1\} \quad \forall (\text{addr}: \text{Memory_Address_4G}, \text{cs}: \text{Segment_Reg_type}, \text{priv}: \text{Memory_privilege}): \\ \text{priv} = \text{segment_to_priv}(\text{cs}) \supset \\ (\forall (\text{pdbr}: \text{Pdbr_type}, \text{lvl}: \text{nat}): \text{lvl} = \text{pdir_lvl} \supset \text{lvl} < \text{max_level})$
--

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `linear_resolve_TCC1`.

Q.E.D.

C.114.20 Linear_Memory.linear_resolve_TCC2

Terse proof for `linear_resolve_TCC2`.

linear_resolve_TCC2:

```
{1}  ∀ (addr: Memory_Address_4G, access: Memory_access, cs: Segment_Reg_type,
      priv: Memory_privilege):
      priv = segment_to_priv(cs) ⊃
      (∀ (pdir: Pdir_type, lvl: nat):
        lvl = pdir.lvl ⊃
        (∀ (x1: Physical_memory):
          every[Physical_memory, [bool, Address]]
            (λ (x: Physical_memory): TRUE,
              λ (t: [bool, Address]):
                Mem?(t'2'type_of) ∧
                (0 ≤ t'2'offset ∧ t'2'offset < max_linear_offset) ∧
                IF t'1
                  THEN aligned?(-1 × lvl × bits_per_level - bits_per_level + bus_width)
                        (offset(t'2))
                  ELSE aligned?(bits_per_level + pe_size)(offset(t'2))
                ENDIF)
            (translate(lvl, pdir'base_addr, addr, access, priv)(x1))))
```

Repeatedly Skolemizing and flattening,
 Expanding the definition of every,
 Using lemma translate_result_leaf,
 Using lemma translate_result_no_leaf,
 Using lemma translate_memory_address,
 Using lemma translate_memory_address2,
 Installing automatic rewrites from: max_linear Mem < bits_per_level pe_size
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 This completes the proof of linear_resolve_TCC2.
 Q.E.D.

C.114.21 Linear_Memory.linear_resolve_TCC3

Terse proof for linear_resolve_TCC3.

linear_resolve_TCC3:

```
{1}  ∀ (addr: Memory_Address_4G, cs: Segment_Reg_type, priv: Memory_privilege):
      priv = segment_to_priv(cs) ⊃
      (∀ (pdir: Pdir_type, lvl: nat):
        lvl = pdir.lvl ⊃ (∀ (leaf: bool, b: Memory_Address_4G): leaf ⊃ off-
          set(b) ≥ 0))
```

Repeatedly Skolemizing and flattening,
 Installing automatic rewrites from: bus_width bits_per_level max_level
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 This completes the proof of linear_resolve_TCC3.
 Q.E.D.

C.114.22 Linear_Memory.linear_resolve_TCC4

Terse proof for linear_resolve_TCC4.

linear_resolve_TCC4:

```
{1}  ∀ (addr: Memory_Address_4G, cs: Segment_Reg_type, priv: Memory_privilege):
      priv = segment_to_priv(cs) ⊃
        (∀ (pdir: Pdir_type, lvl: nat):
          lvl = pdir_lvl ⊃
            (∀ (leaf: bool, b: Memory_Address_4G):
              leaf ⊃ bus_width - bits_per_level × (lvl + 1) ≥ 0))
```

Repeatedly Skolemizing and flattening,
 Installing automatic rewrites from: bus_width bits_per_level max_level pdir_lvl
 Simplifying, rewriting, and recording with decision procedures,
 This completes the proof of linear_resolve_TCC4.
 Q.E.D.

C.114.23 Linear_Memory.linear_resolve_TCC5

Terse proof for linear_resolve_TCC5.

linear_resolve_TCC5:

```
{1}  ∀ (addr: Memory_Address_4G, cs: Segment_Reg_type, priv: Memory_privilege):
      priv = segment_to_priv(cs) ⊃
        (∀ (pdir: Pdir_type, lvl: nat):
          lvl = pdir_lvl ⊃
            (∀ (leaf: bool, b: Memory_Address_4G): ¬ leaf ⊃ offset(b) ≥ 0))
```

Repeatedly Skolemizing and flattening,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 This completes the proof of linear_resolve_TCC5.
 Q.E.D.

C.114.24 Linear_Memory.linear_resolve_TCC6

Terse proof for linear_resolve_TCC6.

linear_resolve_TCC6:

```
{1}  ∀ (addr: Memory_Address_4G, cs: Segment_Reg_type, priv: Memory_privilege):
      priv = segment_to_priv(cs) ⊃
        (∀ (pdir: Pdir_type, lvl: nat):
          lvl = pdir_lvl ⊃
            (∀ (leaf: bool,
              base:
                {b: Memory_Address_4G |
                  IF leaf
                    THEN aligned?(bus_width - bits_per_level × (lvl + 1))
                          (offset(b))
                  ELSE aligned?(bits_per_level + pe_size)(offset(b))
                  ENDIF}):
              ¬ leaf ⊃ (∀ (lvl1: nat): lvl1 = ptab_lvl ⊃ lvl1 < max_level))))
```

Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of linear_resolve_TCC6.

Q.E.D.

C.114.25 Linear_Memory.linear_resolve_TCC7

Terse proof for linear_resolve_TCC7.

linear_resolve_TCC7:

```

{1}  ∀ (addr: Memory_Address_4G, access: Memory_access, cs: Segment_Reg_type,
      priv: Memory_privilege):
  priv = segment_to_priv(cs) ⊃
  (∀ (pdir: Pdir_type, lvl: nat):
    lvl = pdir_lvl ⊃
    (∀ (leaf: bool,
        base:
          {b: Memory_Address_4G |
            IF leaf
              THEN aligned?(bus_width - bits_per_level × (lvl + 1))
                    (offset(b))
            ELSE aligned?(bits_per_level + pe_size)(offset(b))
            ENDIF}):
      ¬ leaf ⊃
      (∀ (lvl1: nat):
        lvl1 = ptab_lvl ⊃
        (∀ (x1: Physical_memory):
          every[Physical_memory, [bool, Address]]
            (λ (x: Physical_memory): TRUE,
              λ (t: [bool, Address]):
                Mem?(t'2'type_of) ∧
                0 ≤ t'2'offset ∧ t'2'offset < max_linear_offset)
            (translate(lvl1, base, addr, access, priv)(x1))))))

```

Repeatedly Skolemizing and flattening,

Expanding the definition of every,

Using lemma translate_memory_address,

Using lemma translate_memory_address2,

Installing automatic rewrites from: max_linear Mem < bits_per_level pe_size

Using lemma translate_result_leaf,

Using lemma translate_result_no_leaf,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of linear_resolve_TCC7.

Q.E.D.

C.114.26 Linear_Memory.linear_resolve_TCC8

Terse proof for linear_resolve_TCC8.

`linear_resolve_TCC8:`

<pre>{1} ∀ (leaf: bool, addr: Memory_Address_4G, cs: Segment_Reg_type, priv: Mem- ory_privilege): priv = segment_to_priv(cs) ⊃ (∀ (pdir: Pdir_type, lvl: nat): lvl = pdir_lvl ⊃ (∀ (b: Memory_Address_4G): leaf ⊃ -1 × lvl × bits_per_level - bits_per_level + bus_width ≥ 0))</pre>

Repeatedly Skolemizing and flattening,

Keeping (-9 1) and hiding *,

Installing automatic rewrites from: bus_width

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `linear_resolve_TCC8`.

Q.E.D.

C.114.27 Linear_Memory.linear_resolve_memory_address

Terse proof for `linear_resolve_memory_address`.

`linear_resolve_memory_address:`

<pre>{1} ∀ (access: Memory_access, addr: Memory_Address_4G, x1: Linear_memory): every [Linear_memory, Address] (λ (x: Linear_memory): TRUE, λ (x: Address): Mem?(x'type_of) ∧ 0 ≤ x'offset ∧ x'offset < max_linear_offset) (linear_resolve(addr, access)(x1))</pre>

Repeatedly Skolemizing and flattening,

Expanding the definition of every,

Expanding the definition of linear_resolve,

Expanding the definition of ##,

Expanding the definition of ok_result,

Installing automatic rewrites from: max_linear Mem < bits_per_level bus_width min_linear
 Address_Helpers.<= max_level pe_size xlat_ofs_memory_address! pdir_lvl ptab_lvl trans-
 late_result_leaf translate_result_no_leaf translate_memory_address! translate_memory_address!
 xlat_ofs_memory_address2! translate_result_leaf translate_result_no_leaf

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 6 subgoals:

linear_resolve_memory_address.1:

{-1}	Mem?(addr' 'type_of)
{-2}	$0 \leq \text{addr}' \text{'offset}$
{-3}	$\text{addr}' \text{'offset} < \text{max_linear_offset}$
{-4}	OK?(read_data(pm, segment_reg_data_type)(CS)(x ₁ '))
{-5}	OK?(read_data(pm, pdb_r_data_type)(PDBR) (state(read_data(pm, segment_reg_data_type)(CS)(x ₁ '))))
{-6}	OK?(translate(0, data(read_data(pm, pdb_r_data_type)(PDBR) (state(read_data(pm, segment_reg_data_type)(CS) (x ₁ ')))) 'base_addr, addr', access', segment_to_priv(data(read_data(pm, seg- ment_reg_data_type)(CS)(x ₁ ')))) (state(read_data(pm, pdb_r_data_type)(PDBR) (state(read_data(pm, segment_reg_data_type)(CS) (x ₁ '))))))))
{-7}	data(translate(0, data(read_data(pm, pdb_r_data_type)(PDBR) (state(read_data(pm, segment_reg_data_type)(CS) (x ₁ ')))) 'base_addr, addr', access', segment_to_priv(data(read_data(pm, seg- ment_reg_data_type)(CS)(x ₁ ')))) (state(read_data(pm, pdb_r_data_type)(PDBR) (state(read_data(pm, segment_reg_data_type)(CS) (x ₁ ')))))))) '1
{1}	xlat_ofs(0, data(translate(0, data(read_data(pm, pdb_r_data_type)(PDBR) (state(read_data(pm, seg- ment_reg_data_type) (CS) (x ₁ ')))) 'base_addr, addr', access', segment_to_priv(data(read_data(pm, seg- ment_reg_data_type)(CS) (x ₁ ')))) (state(read_data(pm, pdb_r_data_type)(PDBR) (state(read_data(pm, seg- ment_reg_data_type) (CS) (x ₁ ')))))))) '2, addr' 'offset < max_linear_offset

Using lemma xlat_ofs_memory_address,

Using lemma translate_result_leaf,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of linear_resolve_memory_address.1.

linear_resolve_memory_address.2:

{-1}	Mem?(addr' 'type_of)
{-2}	$0 \leq \text{addr}'\text{'offset}$
{-3}	$\text{addr}'\text{'offset} < \text{max_linear_offset}$
{-4}	OK?(read_data(pm, segment_reg_data_type)(CS)(x ₁ '))
{-5}	OK?(read_data(pm, pdb_r_data_type)(PDBR) (state(read_data(pm, segment_reg_data_type)(CS)(x ₁ '))))
{-6}	OK?(translate(0, data(read_data(pm, pdb_r_data_type)(PDBR) (state(read_data(pm, segment_reg_data_type)(CS) (x ₁ ')))) 'base_addr, addr', access', segment_to_priv(data(read_data(pm, seg- ment_reg_data_type)(CS)(x ₁ ')))) (state(read_data(pm, pdb_r_data_type)(PDBR) (state(read_data(pm, segment_reg_data_type)(CS) (x ₁ ')))))))
{-7}	data(translate(0, data(read_data(pm, pdb_r_data_type)(PDBR) (state(read_data(pm, segment_reg_data_type)(CS) (x ₁ ')))) 'base_addr, addr', access', segment_to_priv(data(read_data(pm, seg- ment_reg_data_type)(CS)(x ₁ ')))) (state(read_data(pm, pdb_r_data_type)(PDBR) (state(read_data(pm, segment_reg_data_type)(CS) (x ₁ '))))))) '1
{1}	$0 \leq$ xlat_ofs(0, data(translate(0, data(read_data(pm, pdb_r_data_type)(PDBR) (state(read_data(pm, seg- ment_reg_data_type) (CS) (x ₁ ')))) 'base_addr, addr', access', segment_to_priv(data(read_data(pm, seg- ment_reg_data_type) (CS) (x ₁ ')))) (state(read_data(pm, pdb_r_data_type)(PDBR) (state(read_data(pm, seg- ment_reg_data_type) (CS) (x ₁ '))))))) '2, addr') 'offset

Using lemma xlat_ofs_memory_address2,

Using lemma translate_result_leaf,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of linear_resolve_memory_address.2.

linear_resolve_memory_address.3:

{-1}	Mem?(addr' 'type_of)	
{-2}	$0 \leq \text{addr}'\text{'offset}$	
{-3}	$\text{addr}'\text{'offset} < \text{max_linear_offset}$	
{-4}	OK?(read_data(pm, segment_reg_data_type)(CS)(x ₁ '))	
{-5}	OK?(read_data(pm, pdb_r_data_type)(PDBR)	(state(read_data(pm, segment_reg_data_type)(CS)(x ₁ '))))
{-6}	OK?(translate(0,	data(read_data(pm, pdb_r_data_type)(PDBR)
		(state(read_data(pm, segment_reg_data_type)(CS)
		(x ₁ ')) 'base_addr,
	addr', access',	
	segment_to_priv(data(read_data(pm, seg-	
	ment_reg_data_type)(CS)(x ₁ '))))	
	(state(read_data(pm, pdb_r_data_type)(PDBR)	
		(state(read_data(pm, segment_reg_data_type)(CS)
		(x ₁ ')))))))
{-7}	data(translate(0,	data(read_data(pm, pdb_r_data_type)(PDBR)
		(state(read_data(pm, segment_reg_data_type)(CS)
		(x ₁ ')) 'base_addr,
	addr', access',	
	segment_to_priv(data(read_data(pm, seg-	
	ment_reg_data_type)(CS)(x ₁ '))))	
	(state(read_data(pm, pdb_r_data_type)(PDBR)	
		(state(read_data(pm, segment_reg_data_type)(CS)
		(x ₁ '))))))) '1
{1}	Mem?(xlat_ofs(0,	data(translate(0,
		data(read_data(pm, pdb_r_data_type)(PDBR)
		(state(read_data(pm, seg-
	ment_reg_data_type)	(CS)
		(x ₁ ')) 'base_addr,
	addr', access',	
	segment_to_priv(data(read_data(pm, seg-	
	ment_reg_data_type)	(CS)
		(x ₁ '))))
	(state(read_data(pm, pdb_r_data_type)(PDBR)	
		(state(read_data(pm, seg-
	ment_reg_data_type)	(CS)
		(x ₁ '))))))) '2,
	addr' 'type_of)	

Using lemma xlat_ofs_memory_address,

Using lemma translate_result_leaf,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of linear_resolve_memory_address.3.

linear_resolve_memory_address.4:

```

{-1} Mem?(addr' 'type_of)
{-2} 0 ≤ addr' 'offset
{-3} addr' 'offset < max_linear_offset
{-4} OK?(read_data(pm, segment_reg_data_type)(CS)(x₁))
{-5} OK?(read_data(pm, pdb_r_data_type)(PDBR)
      (state(read_data(pm, segment_reg_data_type)(CS)(x₁))))
{-6} OK?(translate(0,
      data(read_data(pm, pdb_r_data_type)(PDBR)
            (state(read_data(pm, segment_reg_data_type)(CS)
                    (x₁)))) 'base_addr,
      addr', access',
      segment_to_priv(data(read_data(pm, seg-
      ment_reg_data_type)(CS)(x₁))))
      (state(read_data(pm, pdb_r_data_type)(PDBR)
            (state(read_data(pm, segment_reg_data_type)(CS)
                    (x₁))))))
{-7} OK?(translate(1,
      data(translate(0,
      data(read_data(pm, pdb_r_data_type)(PDBR)
            (state(read_data(pm, seg-
      ment_reg_data_type)
      (CS)
      (x₁)))) 'base_addr,
      addr', access',
      segment_to_priv(data(read_data(pm, seg-
      ment_reg_data_type)
      (CS)
      (x₁))))
      (state(read_data(pm, pdb_r_data_type)(PDBR)
            (state(read_data(pm, seg-
      ment_reg_data_type)
      (CS)
      (x₁)))))) '2,
      addr', access',
      segment_to_priv(data(read_data(pm, seg-
      ment_reg_data_type)(CS)(x₁))))
      (state(translate(0,
      data(read_data(pm, pdb_r_data_type)(PDBR)
            (state(read_data(pm, seg-
      ment_reg_data_type)
      (CS)
      (x₁)))) 'base_addr,
      addr', access',
      segment_to_priv(data(read_data(pm, seg-
      ment_reg_data_type)
      (CS)
      (x₁))))
      (state(read_data(pm, pdb_r_data_type)(PDBR)
            (state(read_data(pm, seg-
      ment_reg_data_type)
      (CS)
      (x₁)))))))))

```

1356

```

{1} data(translate(0,
      data(read_data(pm, pdb_r_data_type)(PDBR)
            (state(read_data(pm, segment_reg_data_type)(CS)
                    (x₁)))) 'base_addr,
      addr', access',
      segment_to_priv(data(read_data(pm, seg-
      ment_reg_data_type)(CS)(x₁))))
      (state(read_data(pm, pdb_r_data_type)(PDBR)
            (state(read_data(pm, segment_reg_data_type)(CS)
                    (x₁)))))) '1
{2} xlat_ofs(1,

```

Using lemma `translate_result_no_leaf`,

Using lemma `translate_result_no_leaf`,

Using lemma `translate_result_leaf`,

Using lemma `xlat_ofs_memory_address`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `linear_resolve_memory_address.4`.

linear_resolve_memory_address.5:

```

{-1} Mem?(addr' 'type_of)
{-2} 0 ≤ addr' 'offset
{-3} addr' 'offset < max_linear_offset
{-4} OK?(read_data(pm, segment_reg_data_type)(CS)(x'_1))
{-5} OK?(read_data(pm, pdb_r_data_type)(PDBR)
      (state(read_data(pm, segment_reg_data_type)(CS)(x'_1))))
{-6} OK?(translate(0,
      data(read_data(pm, pdb_r_data_type)(PDBR)
            (state(read_data(pm, segment_reg_data_type)(CS)
                      (x'_1)))) 'base_addr,
      addr', access',
      segment_to_priv(data(read_data(pm, seg-
ment_reg_data_type)(CS)(x'_1))))
      (state(read_data(pm, pdb_r_data_type)(PDBR)
            (state(read_data(pm, segment_reg_data_type)(CS)
                      (x'_1))))))
{-7} OK?(translate(1,
      data(translate(0,
      data(read_data(pm, pdb_r_data_type)(PDBR)
            (state(read_data(pm, seg-
ment_reg_data_type)
                                (CS)
                                (x'_1)))) 'base_addr,
      addr', access',
      segment_to_priv(data(read_data(pm, seg-
ment_reg_data_type)
                                (CS)
                                (x'_1))))
      (state(read_data(pm, pdb_r_data_type)(PDBR)
            (state(read_data(pm, seg-
ment_reg_data_type)
                                (CS)
                                (x'_1)))))) '2,
      addr', access',
      segment_to_priv(data(read_data(pm, seg-
ment_reg_data_type)(CS)(x'_1))))
      (state(translate(0,
      data(read_data(pm, pdb_r_data_type)(PDBR)
            (state(read_data(pm, seg-
ment_reg_data_type)
                                (CS)
                                (x'_1)))) 'base_addr,
      addr', access',
      segment_to_priv(data(read_data(pm, seg-
ment_reg_data_type)
                                (CS)
                                (x'_1))))
      (state(read_data(pm, pdb_r_data_type)(PDBR)
            (state(read_data(pm, seg-
ment_reg_data_type)
                                (CS)
                                (x'_1)))))))))
{1} data(translate(0,
      data(read_data(pm, pdb_r_data_type)(PDBR)
            (state(read_data(pm, segment_reg_data_type)(CS)
                      (x'_1)))) 'base_addr,
      addr', access',
      segment_to_priv(data(read_data(pm, seg-
ment_reg_data_type)(CS)(x'_1))))
      (state(read_data(pm, pdb_r_data_type)(PDBR)
            (state(read_data(pm, segment_reg_data_type)(CS)
                      (x'_1)))))) '1
{2} 0 ≤

```

Using lemma `xlat_ofs_memory_address2`,

Using lemma `translate_result_no_leaf`,

Using lemma `translate_result_no_leaf`,

Using lemma `translate_result_leaf`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `linear_resolve_memory_address.5`.

Using lemma translate_result_no_leaf,
 Using lemma translate_result_leaf,
 Using lemma translate_result_no_leaf,
 Using lemma translate_memory_address,
 Using lemma xlat_ofs_memory_address,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 This completes the proof of `linear_resolve_memory_address.6`.
 Q.E.D.

C.114.28 Linear_Memory.linear_read_TCC1

Terse proof for `linear_read_TCC1`.

`linear_read_TCC1`:

`{1}` Mem?(min_linear 'type_of)

Expanding the definition of `min_linear`,
 Rewriting using `Mem`, matching in `*`,
 Simplifying, rewriting, and recording with decision procedures,
 This completes the proof of `linear_read_TCC1`.
 Q.E.D.

C.114.29 Linear_Memory.linear_read_TCC2

Terse proof for `linear_read_TCC2`.

`linear_read_TCC2`:

`{1}` Mem?(max_linear 'type_of)

Expanding the definition of `max_linear`,
 Expanding the definition of `Mem`,
 which is trivially true.
 This completes the proof of `linear_read_TCC2`.
 Q.E.D.

C.114.30 Linear_Memory.linear_read_TCC3

Terse proof for `linear_read_TCC3`.

`linear_read_TCC3`:

`{1}` $\forall (a: \text{Address}):$
 $\text{Mem}?(\text{type_of}(a)) \wedge \text{in_memory}(\text{min_linear}, \text{max_linear})(a) \supset$
 $0 \leq a' \text{offset} \wedge a' \text{offset} < \text{max_linear_offset}$

Expanding the definition of `min_linear`,
 Trying repeated skolemization, instantiation, and if-lifting,
 Expanding the definition of `max_linear`,
 Expanding the definition of `max_linear_offset`,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 This completes the proof of `linear_read_TCC3`.
 Q.E.D.

C.114.31 Linear_Memory.linear_read_side_effect_TCC1

Terse proof for `linear_read_side_effect_TCC1`.

`linear_read_side_effect_TCC1`:

<pre> {1} ∃ (a: Address, bl: list[Byte], s: Linear_memory): Mem?(type_of(a)) ∧ (null?(bl) ∨ (reg_base(min_linear)(type_of(a)) ≤ a ∧ a + length(bl) ≤ reg_size(max_linear)(type_of(a)))) ⊃ every[[Address, list[Byte]]] (λ (t: [Address, list[Byte]]): Mem?(t'1'type_of) ∧ 0 ≤ t'1'offset ∧ t'1'offset < max_linear_offset) (split(min_page, a, bl)) </pre>

Repeatedly Skolemizing and flattening,

Hiding formulas: -1,

Case splitting on `bl' = null`,

we get 2 subgoals:

`linear_read_side_effect_TCC1.1`:

<pre> {-1} bl' = null {-2} Mem?(type_of(a')) {-3} null?(bl') ∨ (reg_base(min_linear)(type_of(a')) ≤ a' ∧ a' + length(bl') ≤ reg_size(max_linear)(type_of(a'))) </pre>
<pre> {1} every[[Address, list[Byte]]] (λ (t: [Address, list[Byte]]): Mem?(t'1'type_of) ∧ 0 ≤ t'1'offset ∧ t'1'offset < max_linear_offset) (split(min_page, a', bl')) </pre>

Using lemma `split_null`,

Expanding the definition of `every`,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `linear_read_side_effect_TCC1.1`.

`linear_read_side_effect_TCC1.2`:

<pre> {-1} Mem?(type_of(a')) {-2} null?(bl') ∨ (reg_base(min_linear)(type_of(a')) ≤ a' ∧ a' + length(bl') ≤ reg_size(max_linear)(type_of(a'))) </pre>
<pre> {1} bl' = null {2} every[[Address, list[Byte]]] (λ (t: [Address, list[Byte]]): Mem?(t'1'type_of) ∧ 0 ≤ t'1'offset ∧ t'1'offset < max_linear_offset) (split(min_page, a', bl')) </pre>

Using lemma `split_range`,

Using lemma `split_no_null`,

Using lemma `every_implied`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 2 subgoals:

linear_read_side_effect_TCC1.2.1:

{-1}	cons?(bl')
{-2}	every($\lambda (a_2: \text{Address}, \text{bl2}: \text{list}[\text{Byte}]): \text{cons?}(\text{bl2})(\text{split}(\text{min_page}, a', \text{bl}'))$)
{-3}	every($\lambda (a_2: \text{Address}, \text{bl2}: \text{list}[\text{Byte}]):$ $a' \leq a_2 \wedge a_2 + \text{length}(\text{bl2}) \leq a' + \text{length}(\text{bl}')$ $(\text{split}(\text{min_page}, a', \text{bl}'))$)
{-4}	Mem?(type_of(a'))
{-5}	reg_base(min_linear)(type_of(a')) $\leq a'$
{-6}	$a' + \text{length}(\text{bl}') \leq \text{reg_size}(\text{max_linear})(\text{type_of}(a'))$
{1}	every($(\lambda (a_2: \text{Address}, \text{bl2}: \text{list}[\text{Byte}]):$ $\text{cons?}(\text{bl2}) \wedge a' \leq a_2 \wedge a_2 + \text{length}(\text{bl2}) \leq a' + \text{length}(\text{bl}')$ $(\text{split}(\text{min_page}, a', \text{bl}'))$)
{2}	every([[Address, list[Byte]]] $(\lambda (t: [\text{Address}, \text{list}[\text{Byte}]]):$ $\text{Mem?}(t'1'\text{type_of}) \wedge 0 \leq t'1'\text{offset} \wedge t'1'\text{offset} < \text{max_linear_offset}$ $(\text{split}(\text{min_page}, a', \text{bl}'))$)

Hiding formulas: 2,

Using lemma every_conjunct_left,

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of linear_read_side_effect_TCC1.2.1.

linear_read_side_effect_TCC1.2.2:

{-1}	cons?(bl')
{-2}	every($\lambda (a_2: \text{Address}, \text{bl2}: \text{list}[\text{Byte}]): \text{cons?}(\text{bl2})(\text{split}(\text{min_page}, a', \text{bl}'))$)
{-3}	every($\lambda (a_2: \text{Address}, \text{bl2}: \text{list}[\text{Byte}]):$ $a' \leq a_2 \wedge a_2 + \text{length}(\text{bl2}) \leq a' + \text{length}(\text{bl}')$ $(\text{split}(\text{min_page}, a', \text{bl}'))$)
{-4}	Mem?(type_of(a'))
{-5}	reg_base(min_linear)(type_of(a')) $\leq a'$
{-6}	$a' + \text{length}(\text{bl}') \leq \text{reg_size}(\text{max_linear})(\text{type_of}(a'))$
{1}	$\forall (t_1: [\text{Address}, \text{list}[\text{Byte}]]):$ $\text{cons?}(t_1'2) \wedge a' \leq t_1'1 \wedge t_1'1 + \text{length}(t_1'2) \leq a' + \text{length}(\text{bl}')$ $\supset \text{Mem?}(t_1'1'\text{type_of}) \wedge 0 \leq t_1'1'\text{offset} \wedge t_1'1'\text{offset} < \text{max_linear_offset}$
{2}	every([[Address, list[Byte]]] $(\lambda (t: [\text{Address}, \text{list}[\text{Byte}]]):$ $\text{Mem?}(t'1'\text{type_of}) \wedge 0 \leq t'1'\text{offset} \wedge t'1'\text{offset} < \text{max_linear_offset}$ $(\text{split}(\text{min_page}, a', \text{bl}'))$)

Hiding formulas: (-2 -3 2),

Trying repeated skolemization, instantiation, and if-lifting,

Expanding the definition of max_linear,

Expanding the definition of Mem,

Expanding the definition of max_linear_offset,

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of linear_read_side_effect_TCC1.2.2.

Q.E.D.

C.115 Proofs for Linear_Memory_Blessing (challenge-linear.pvs)

C.115.1 Linear_Memory_Blessing.PTab_Address_TCC1

Terse proof for PTab_Address_TCC1.

PTab_Address_TCC1:

$$\{1\} \quad \forall (a: \text{Memory_Address_4G}): \text{offset}(a) \geq 0$$

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of PTab_Address_TCC1.

Q.E.D.

C.115.2 Linear_Memory_Blessing.pe_in_pdir_range?_TCC1

Terse proof for pe_in_pdir_range?_TCC1.

pe_in_pdir_range?_TCC1:

$$\{1\} \quad \forall (s: \text{Linear_memory}[\text{Physical_memory}, \text{pm_phy}], \text{linear_range}: \text{PRED}[\text{Memory_Address_4G}], \\ \text{phy_a}: \text{Memory_Address_4G}, \text{res}: \text{ExprResult}[\text{Physical_memory}, \text{Pdbr_type}]): \\ \text{OK?}(\text{res}) \wedge \text{res} = \text{read_data}(\text{pm_phy}, \text{pdbr_data_type})(\text{PDBR})(s) \supset \\ (\forall (\text{lin_a}: (\text{linear_range})): \text{pdir_lvl}[\text{Physical_memory}, \text{pm_phy}] < \text{max_level})$$

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of pe_in_pdir_range?_TCC1.

Q.E.D.

C.115.3 Linear_Memory_Blessing.pe_in_ptab_range?_TCC1

Terse proof for pe_in_ptab_range?_TCC1.

pe_in_ptab_range?_TCC1:

$$\{1\} \quad \forall (s: \text{Linear_memory}[\text{Physical_memory}, \text{pm_phy}], \text{linear_range}: \text{PRED}[\text{Memory_Address_4G}], \\ \text{phy_a}: \text{Memory_Address_4G}, \text{pde}: (\text{pe_in_pdir_range?}(s, \text{linear_range})), \\ \text{lin_a}: (\text{linear_range})): \\ \text{pdir_lvl}[\text{Physical_memory}, \text{pm_phy}] < \text{max_level}$$

Repeatedly Skolemizing and flattening,

Keeping (1) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of pe_in_ptab_range?_TCC1.

Q.E.D.

C.115.4 Linear_Memory_Blessing.pe_in_ptab_range?_TCC2

Terse proof for pe_in_ptab_range?_TCC2.

pe_in_ptab_range?_TCC2:

$\{1\} \quad \forall (s: \text{Linear_memory}[\text{Physical_memory}, \text{pm_phy}], \text{linear_range}: \text{PRED}[\text{Memory_Address_4G}], \\ \text{phy_a}: \text{Memory_Address_4G}, \text{pde}: (\text{pe_in_pdir_range?}(s, \text{linear_range})), \\ \text{lin_a}: (\text{linear_range}), \\ \text{res}: \text{ExprResult}[\text{Physical_memory}, ((\text{range_pt}(\text{pdir_lvl}[\text{Physical_memory}, \text{pm_phy}])))]): \\ \text{OK?}(\text{res}) \wedge \text{res} = \text{read_data}(\text{pm_phy}, \text{paging_data_type}(\text{pdir_lvl}))(\text{pde})(s) \supset \\ \text{pdir?}(\text{data}[\text{Physical_memory}, ((\text{range_pt}(\text{pdir_lvl}))])(\text{res}))$

Repeatedly Skolemizing and flattening,
 Adding type constraints for data[Physical_memory, ((range_pt(pdir_lvl)))(res!1),
 Expanding the definition of range_pt,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 Expanding the definition of pdir_lvl,
 which is trivially true.
 This completes the proof of pe_in_ptab_range?_TCC2.
 Q.E.D.

C.115.5 Linear_Memory_Blessing.pe_in_ptab_range?_TCC3

Terse proof for pe_in_ptab_range?_TCC3.

pe_in_ptab_range?_TCC3:

$\{1\} \quad \forall (s: \text{Linear_memory}[\text{Physical_memory}, \text{pm_phy}], \text{linear_range}: \text{PRED}[\text{Memory_Address_4G}], \\ \text{phy_a}: \text{Memory_Address_4G}, \text{pde}: (\text{pe_in_pdir_range?}(s, \text{linear_range})), \\ \text{lin_a}: (\text{linear_range}), \\ \text{res}: \text{ExprResult}[\text{Physical_memory}, ((\text{range_pt}(\text{pdir_lvl}[\text{Physical_memory}, \text{pm_phy}])))]): \\ \text{pde_pt?}(\text{pdir}(\text{data}(\text{res}))) \wedge \\ \text{OK?}(\text{res}) \wedge \text{res} = \text{read_data}(\text{pm_phy}, \text{paging_data_type}(\text{pdir_lvl}))(\text{pde})(s) \\ \supset \text{ptab_lvl}[\text{Physical_memory}, \text{pm_phy}] < \text{max_level}$
--

Repeatedly Skolemizing and flattening,
 Keeping (1) and hiding *,
 Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of pe_in_ptab_range?_TCC3.
 Q.E.D.

C.115.6 Linear_Memory_Blessing.pe_in_ptab_range?_TCC4

Terse proof for pe_in_ptab_range?_TCC4.

pe_in_ptab_range?_TCC4:

$\{1\} \quad \forall (s: \text{Linear_memory}[\text{Physical_memory}, \text{pm_phy}], \text{linear_range}: \text{PRED}[\text{Memory_Address_4G}], \\ \text{phy_a}: \text{Memory_Address_4G}, \text{pde}: (\text{pe_in_pdir_range?}(s, \text{linear_range})), \\ \text{lin_a}: (\text{linear_range}), \\ \text{res}: \text{ExprResult}[\text{Physical_memory}, ((\text{range_pt}(\text{pdir_lvl}[\text{Physical_memory}, \text{pm_phy}])))]): \\ \text{pde_pt?}(\text{pdir}(\text{data}(\text{res}))) \wedge \\ \text{OK?}(\text{res}) \wedge \text{res} = \text{read_data}(\text{pm_phy}, \text{paging_data_type}(\text{pdir_lvl}))(\text{pde})(s) \\ \supset \text{present?}(\text{data}[\text{Physical_memory}, ((\text{range_pt}(\text{pdir_lvl}))])(\text{res}))$
--

Repeatedly Skolemizing and flattening,
 Expanding the definition of present?,

Adding type constraints for data(res!1),
 Expanding the definition of range_pt,
 Expanding the definition of pdir_lvl,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 This completes the proof of pe_in_ptab_range?_TCC4.
 Q.E.D.

C.115.7 Linear_Memory_Blessing.pe_in_pt_range?_TCC1

Terse proof for pe_in_pt_range?_TCC1.

pe_in_pt_range?_TCC1:

$$\frac{\{1\} \quad \forall (s: \text{Linear_memory}[\text{Physical_memory}, \text{pm_phy}], \text{lvl}: \text{Level}, \text{phy_a}: \text{Memory_Address_4G}):}{\neg (\text{lvl} = \text{pdir_lvl}[\text{Physical_memory}, \text{pm_phy}] \wedge \text{lvl} = \text{ptab_lvl}[\text{Physical_memory}, \text{pm_phy}])}$$

Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of pe_in_pt_range?_TCC1.
 Q.E.D.

C.115.8 Linear_Memory_Blessing.pe_in_pt_range?_TCC2

Terse proof for pe_in_pt_range?_TCC2.

pe_in_pt_range?_TCC2:

$$\frac{\{1\} \quad \forall (s: \text{Linear_memory}[\text{Physical_memory}, \text{pm_phy}], \text{lvl}: \text{Level}, \text{phy_a}: \text{Memory_Address_4G}):}{\text{lvl} = \text{pdir_lvl}[\text{Physical_memory}, \text{pm_phy}] \vee \text{lvl} = \text{ptab_lvl}[\text{Physical_memory}, \text{pm_phy}]}$$

Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of pe_in_pt_range?_TCC2.
 Q.E.D.

C.115.9 Linear_Memory_Blessing.pe_in_pt_range_aligned

Terse proof for pe_in_pt_range_aligned.

pe_in_pt_range_aligned:

$$\frac{\{1\} \quad \forall (\text{linear_range}: \text{PRED}[\text{Memory_Address_4G}], \text{lvl}: \text{Level}, s: \text{Linear_memory}[\text{Physical_memory}, \text{pm_phy}], \text{phy_a}: \text{Memory_Address_4G}):}{\text{pe_in_pt_range?}(s, \text{linear_range}, \text{lvl})(\text{phy_a}) \supset \text{aligned?}(\text{pe_size})(\text{offset}(\text{phy_a}))}$$

Expanding the definition of pe_in_pt_range?,
 Expanding the definition of pe_in_ptab_range?,
 Expanding the definition of pe_in_pdir_range?,
 Repeatedly Skolemizing and flattening,
 Installing automatic rewrites from: pe_size bits_per_level xlat_idx_pe_aligned
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 we get 2 subgoals:

pe_in_pt_range_aligned.1:

{-1}	$lvl' < \text{max_level}$
{-2}	$\text{Mem?}(\text{phy_a}'\text{'type_of})$
{-3}	$0 \leq \text{phy_a}'\text{'offset}$
{-4}	$\text{phy_a}'\text{'offset} < \text{max_linear_offset}$
{-5}	$lvl' = \text{pdir_lvl}$
{-6}	$\text{OK?}(\text{read_data}(\text{pm_phy}, \text{pdir_data_type})(\text{PDBR})(s'))$
{-7}	$\exists (\text{lin_a}: (\text{linear_range}')):$ $\text{xlat_idx}(\text{pdir_lvl}, \text{data}(\text{read_data}(\text{pm_phy}, \text{pdir_data_type})(\text{PDBR})(s'))\text{'base_addr}, \text{lin_a})$ $= \text{phy_a}'$
{1}	$\text{aligned?}(2)(\text{offset}(\text{phy_a}'))$

Repeatedly Skolemizing and flattening,

Adding type constraints for $(\text{data}(\text{read_data}(\text{pm_phy}, \text{pdir_data_type})(\text{PDBR})(s1))\text{'base_addr})$,

Using lemma `xlat_idx_pe_aligned`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `pe_in_pt_range_aligned.1`.

pe_in_pt_range_aligned.2:

{-1}	$lvl' < \text{max_level}$
{-2}	$\text{Mem?}(\text{phy_a}'\text{'type_of})$
{-3}	$0 \leq \text{phy_a}'\text{'offset}$
{-4}	$\text{phy_a}'\text{'offset} < \text{max_linear_offset}$
{-5}	$\exists (\text{pde}: (\text{pe_in_pdir_range?}(s', \text{linear_range}'))):$ $\exists (\text{lin_a}: (\text{linear_range}')):$ $\text{OK?}(\text{read_data}(\text{pm_phy}, \text{paging_data_type}(\text{pdir_lvl}))(\text{pde})(s')) \wedge$ $\text{pde_pt?}(\text{pdir}(\text{data}(\text{read_data}(\text{pm_phy}, \text{paging_data_type}(\text{pdir_lvl}))(\text{pde})(s')))) \wedge$ $\text{xlat_idx}(\text{ptab_lvl},$ $\quad \text{base}(\text{data}(\text{read_data}(\text{pm_phy}, \text{paging_data_type}(\text{pdir_lvl}))(\text{pde})(s'))),$ $\quad \text{lin_a})$ $= \text{phy_a}'$
{1}	$lvl' = \text{pdir_lvl}$
{2}	$\text{aligned?}(2)(\text{offset}(\text{phy_a}'))$

Repeatedly Skolemizing and flattening,

Using lemma `xlat_idx_pe_aligned`,

Expanding the definition of `base`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `pe_in_pt_range_aligned.2`.

Q.E.D.

C.115.10 Linear_Memory_Blessing.pe_in_pt_address_in_pt

Terse proof for `pe_in_pt_address_in_pt`.

pe_in_pt_address_in_pt:

{1}	$\forall (\text{linear_range}: \text{PRED}[\text{Memory_Address_4G}], \text{lvl}: \text{Level},$ $s: \text{Linear_memory}[\text{Physical_memory}, \text{pm_phy}], \text{base}: \text{Memory_Address_4G}):$ $\text{pe_in_pt_range?}(s, \text{linear_range}, \text{lvl})(\text{base}) \supset$ $(\text{address_block}(\text{base}, \text{expt}(2, \text{pe_size})) \subseteq \text{extend} [\text{Address}, \text{Memory_Address_4G}, \text{bool}, \text{FALSE}] (\text{address_i}$
-----	---

Repeatedly Skolemizing and flattening,

Expanding the definition of `subset?`,

C Proof scripts

Expanding the definition of member,
 Repeatedly Skolemizing and flattening,
 Expanding the definition of address_in_pt_range?,
 Expanding the definition of extend,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 we get 4 subgoals:

pe_in_pt_address_in_pt.1:

{-1}	$lvl' < \text{max_level}$
{-2}	$\text{Mem?}(\text{base}' \text{ type_of})$
{-3}	$0 \leq \text{base}' \text{ offset}$
{-4}	$\text{base}' \text{ offset} < \text{max_linear_offset}$
{-5}	$\text{pe_in_pt_range?}(s', \text{linear_range}', lvl')(\text{base}')$
{-6}	$\text{address_block}(\text{base}', \text{expt}(2, \text{pe_size}))(x')$
{1}	$\exists (\text{pe}: \text{Memory_Address_4G}, lvl: \text{Level}):$ $\text{pe_in_pt_range?}(s', \text{linear_range}', lvl)(\text{pe}) \wedge$ $\text{address_block}(\text{pe}, \text{expt}(2, \text{pe_size}))(x')$

Instantiating quantified variables,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 This completes the proof of pe_in_pt_address_in_pt.1.

pe_in_pt_address_in_pt.2:

{-1}	$lvl' < \text{max_level}$
{-2}	$\text{Mem?}(\text{base}' \text{ type_of})$
{-3}	$0 \leq \text{base}' \text{ offset}$
{-4}	$\text{base}' \text{ offset} < \text{max_linear_offset}$
{-5}	$\text{pe_in_pt_range?}(s', \text{linear_range}', lvl')(\text{base}')$
{-6}	$\text{address_block}(\text{base}', \text{expt}(2, \text{pe_size}))(x')$
{1}	$x' \text{ offset} < \text{max_linear_offset}$

Using lemma pe_in_pt_range_aligned,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 Expanding the definition of address_block,
 Applying disjunctive simplification to flatten sequent,
 Using lemma aligned_add_below,
 we get 2 subgoals:

pe_in_pt_address_in_pt.2.1:

{-1}	$\text{pe_size} \leq \text{bus_width} \wedge$ $\text{offset}(\text{base}') < \text{expt}(2, \text{bus_width}) \wedge$ $\text{aligned?}(\text{pe_size})(\text{offset}(\text{base}')) \wedge \text{expt}(2, \text{pe_size}) - 1 < \text{expt}(2, \text{pe_size})$ $\supset \text{offset}(\text{base}') + \text{expt}(2, \text{pe_size}) - 1 < \text{expt}(2, \text{bus_width})$
{-2}	$\text{pe_in_pt_range?}(s', \text{linear_range}', lvl')(\text{base}')$
{-3}	$\text{aligned?}(\text{pe_size})(\text{offset}(\text{base}'))$
{-4}	$lvl' < \text{max_level}$
{-5}	$\text{Mem?}(\text{base}' \text{ type_of})$
{-6}	$0 \leq \text{base}' \text{ offset}$
{-7}	$\text{base}' \text{ offset} < \text{max_linear_offset}$
{-8}	$\text{type_of}(\text{base}') = \text{type_of}(x')$
{-9}	$\text{offset}(x') < \text{offset}(\text{base}') + \text{expt}(2, \text{pe_size})$
{1}	$x' \text{ offset} < \text{max_linear_offset}$

Installing automatic rewrites from: (max_linear_offset! bus_width! pe_size!)
 Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `pe_in_pt_address_in_pt.2.1`.

`pe_in_pt_address_in_pt.2.2`:

{-1}	<code>pe_in_pt_range?(s', linear_range', lvl')(base')</code>
{-2}	<code>aligned?(pe_size)(offset(base'))</code>
{-3}	<code>lvl' < max_level</code>
{-4}	<code>Mem?(base' type_of)</code>
{-5}	<code>0 ≤ base' offset</code>
{-6}	<code>base' offset < max_linear_offset</code>
{-7}	<code>type_of(base') = type_of(x')</code>
{-8}	<code>offset(x') < offset(base') + expt(2, pe_size)</code>
{1}	<code>expt(2, pe_size) - 1 ≥ 0</code>
{2}	<code>x' offset < max_linear_offset</code>

Keeping (1) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `pe_in_pt_address_in_pt.2.2`.

`pe_in_pt_address_in_pt.3`:

{-1}	<code>lvl' < max_level</code>
{-2}	<code>Mem?(base' type_of)</code>
{-3}	<code>0 ≤ base' offset</code>
{-4}	<code>base' offset < max_linear_offset</code>
{-5}	<code>pe_in_pt_range?(s', linear_range', lvl')(base')</code>
{-6}	<code>address_block(base', expt(2, pe_size))(x')</code>
{1}	<code>0 ≤ x' offset</code>

Expanding the definition of `address_block`,
which is trivially true.

This completes the proof of `pe_in_pt_address_in_pt.3`.

`pe_in_pt_address_in_pt.4`:

{-1}	<code>lvl' < max_level</code>
{-2}	<code>Mem?(base' type_of)</code>
{-3}	<code>0 ≤ base' offset</code>
{-4}	<code>base' offset < max_linear_offset</code>
{-5}	<code>pe_in_pt_range?(s', linear_range', lvl')(base')</code>
{-6}	<code>address_block(base', expt(2, pe_size))(x')</code>
{1}	<code>Mem?(x' type_of)</code>

Expanding the definition of `address_block`,
which is trivially true.

This completes the proof of `pe_in_pt_address_in_pt.4`.

Q.E.D.

C.115.11 Linear_Memory_Blessing.is_linear_plain_memory?_TCC1

Terse proof for `is_linear_plain_memory?_TCC1`.

is_linear_plain_memory?_TCC1:

$\{1\} \quad \forall (pm: \text{Plain_Memory}[\text{Linear_memory}[\text{Physical_memory}, pm_phy]]):$ $(\forall (s: (pm' \text{states})): \text{OK?}(\text{read_data}(pm_phy, \text{segment_reg_data_type})(CS)(s))) \wedge$ $pm' \text{mem} = \text{linear_pm}$ \supset $(\forall (s_1, s_2: (pm' \text{states})):$ $\text{OK?}[\text{Physical_memory}, \text{Segment_Reg_type}]$ $(\text{read_data}[\text{Physical_memory}, \text{Segment_Reg_type}]$ $(pm_phy, \text{segment_reg_data_type})(CS)(s_1)))$

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of is_linear_plain_memory?_TCC1.
Q.E.D.

C.115.12 Linear_Memory_Blessing.is_linear_plain_memory?_TCC2

Terse proof for is_linear_plain_memory?_TCC2.

is_linear_plain_memory?_TCC2:

$\{1\} \quad \forall (pm: \text{Plain_Memory}[\text{Linear_memory}[\text{Physical_memory}, pm_phy]]):$ $(\forall (s: (pm' \text{states})): \text{OK?}(\text{read_data}(pm_phy, \text{segment_reg_data_type})(CS)(s))) \wedge$ $pm' \text{mem} = \text{linear_pm}$ \supset $(\forall (s_1, s_2: (pm' \text{states})):$ $\text{OK?}[\text{Physical_memory}, \text{Segment_Reg_type}]$ $(\text{read_data}[\text{Physical_memory}, \text{Segment_Reg_type}]$ $(pm_phy, \text{segment_reg_data_type})(CS)(s_2)))$

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of is_linear_plain_memory?_TCC2.
Q.E.D.

C.115.13 Linear_Memory_Blessing.is_linear_plain_memory?_TCC3

Terse proof for is_linear_plain_memory?_TCC3.

is_linear_plain_memory?_TCC3:

$\{1\} \quad \forall (pm: \text{Plain_Memory}[\text{Linear_memory}[\text{Physical_memory}, pm_phy]]):$ $(\forall (s: (pm' \text{states})): \text{OK?}(\text{read_data}(pm_phy, \text{pdbr_data_type})(PDBR)(s))) \wedge$ $(\forall (s_1, s_2: (pm' \text{states})):$ $\text{data}(\text{read_data}(pm_phy, \text{segment_reg_data_type})(CS)(s_1)) =$ $\text{data}(\text{read_data}(pm_phy, \text{segment_reg_data_type})(CS)(s_2)))$ \wedge $(\forall (s: (pm' \text{states})): \text{OK?}(\text{read_data}(pm_phy, \text{segment_reg_data_type})(CS)(s))) \wedge$ $pm' \text{mem} = \text{linear_pm}$ \supset $(\forall (s_{11}, s_{21}: (pm' \text{states})):$ $\text{OK?}[\text{Physical_memory}, \text{Pdbr_type}]$ $(\text{read_data}[\text{Physical_memory}, \text{Pdbr_type}](pm_phy, \text{pdbr_data_type})(PDBR)(s_{11})))$

Repeatedly Skolemizing and flattening,

Instantiating quantified variables,

This completes the proof of `is_linear_plain_memory?_TCC3`.

Q.E.D.

C.115.14 Linear_Memory_Blessing.is_linear_plain_memory?_TCC4

Terse proof for `is_linear_plain_memory?_TCC4`.

`is_linear_plain_memory?_TCC4`:

$$\begin{array}{l}
 \{1\} \quad \forall (pm: \text{Plain_Memory}[\text{Linear_memory}[\text{Physical_memory}, pm_phy]]): \\
 \quad (\forall (s: (pm'states)): \text{OK?}(\text{read_data}(pm_phy, pdr_data_type)(\text{PDBR})(s))) \wedge \\
 \quad (\forall (s_1, s_2: (pm'states)): \\
 \quad \quad \text{data}(\text{read_data}(pm_phy, \text{segment_reg_data_type})(\text{CS})(s_1)) = \\
 \quad \quad \text{data}(\text{read_data}(pm_phy, \text{segment_reg_data_type})(\text{CS})(s_2))) \\
 \quad \wedge \\
 \quad (\forall (s: (pm'states)): \text{OK?}(\text{read_data}(pm_phy, \text{segment_reg_data_type})(\text{CS})(s))) \wedge \\
 \quad pm'mem = \text{linear_pm} \\
 \quad \supset \\
 \quad (\forall (s_{11}, s_{21}: (pm'states)): \\
 \quad \quad \text{OK?}[\text{Physical_memory}, \text{Pdr_type}] \\
 \quad \quad (\text{read_data}[\text{Physical_memory}, \text{Pdr_type}](pm_phy, pdr_data_type)(\text{PDBR})(s_{21})))
 \end{array}$$

Repeatedly Skolemizing and flattening,

Instantiating quantified variables,

This completes the proof of `is_linear_plain_memory?_TCC4`.

Q.E.D.

C.115.15 Linear_Memory_Blessing.is_linear_plain_memory?_TCC5

Terse proof for `is_linear_plain_memory?_TCC5`.

is_linear_plain_memory?_TCC5:

$$\begin{array}{l}
\{1\} \quad \forall (pm: \text{Plain_Memory}[\text{Linear_memory}[\text{Physical_memory}, pm_phy]]): \\
\quad (\forall (s: (pm'states), lvl: \text{Level}, \\
\quad \quad a: \\
\quad \quad \quad (\text{pe_in_pt_range?}(s, \\
\quad \quad \quad \quad \text{restrict}[\text{Address}, \text{Memory_Address_4G}, \text{boolean}] \\
\quad \quad \quad \quad \quad ((pm'ro_addr \cup pm'rw_addr)), \\
\quad \quad \quad \quad \quad \quad lvl))))): \\
\quad \quad \text{OK?}(\text{read_data}(pm_phy, \text{paging_data_type}(lvl))(a)(s))) \\
\quad \wedge \\
\quad (\forall (s_1, s_2: (pm'states)): \\
\quad \quad \text{pe_in_pdir_range?}(s_1, \\
\quad \quad \quad \text{restrict}[\text{Address}, \text{Memory_Address_4G}, \text{boolean}] \\
\quad \quad \quad \quad ((pm'ro_addr \cup pm'rw_addr))) \\
\quad \quad = \\
\quad \quad \quad \text{pe_in_pdir_range?}(s_2, \\
\quad \quad \quad \quad \text{restrict}[\text{Address}, \text{Memory_Address_4G}, \text{boolean}] \\
\quad \quad \quad \quad \quad ((pm'ro_addr \cup pm'rw_addr))) \\
\quad \quad \wedge \\
\quad \quad \quad \text{pe_in_ptab_range?}(s_1, \\
\quad \quad \quad \quad \text{restrict}[\text{Address}, \text{Memory_Address_4G}, \text{boolean}] \\
\quad \quad \quad \quad \quad ((pm'ro_addr \cup pm'rw_addr))) \\
\quad \quad = \\
\quad \quad \quad \text{pe_in_ptab_range?}(s_2, \\
\quad \quad \quad \quad \text{restrict}[\text{Address}, \text{Memory_Address_4G}, \text{boolean}] \\
\quad \quad \quad \quad \quad ((pm'ro_addr \cup pm'rw_addr)))) \\
\quad \wedge \\
\quad (\forall (s_1, s_2: (pm'states)): \\
\quad \quad \text{data}(\text{read_data}(pm_phy, \text{pdbr_data_type})(\text{PDBR})(s_1)) = \\
\quad \quad \quad \text{data}(\text{read_data}(pm_phy, \text{pdbr_data_type})(\text{PDBR})(s_2))) \\
\quad \wedge \\
\quad (\forall (s: (pm'states)): \text{OK?}(\text{read_data}(pm_phy, \text{pdbr_data_type})(\text{PDBR})(s))) \wedge \\
\quad (\forall (s_1, s_2: (pm'states)): \\
\quad \quad \text{data}(\text{read_data}(pm_phy, \text{segment_reg_data_type})(\text{CS})(s_1)) = \\
\quad \quad \quad \text{data}(\text{read_data}(pm_phy, \text{segment_reg_data_type})(\text{CS})(s_2))) \\
\quad \wedge \\
\quad (\forall (s: (pm'states)): \text{OK?}(\text{read_data}(pm_phy, \text{segment_reg_data_type})(\text{CS})(s))) \\
\quad \wedge pm'mem = \text{linear_pm} \\
\quad \supset \\
\quad (\forall (s_{11}, s_{21}: (pm'states), lvl1: \text{Level}, \\
\quad \quad a_1: \\
\quad \quad \quad (\text{pe_in_pt_range?}(s_{11}, \\
\quad \quad \quad \quad \text{restrict}[\text{Address}, \text{Memory_Address_4G}, \text{boolean}] \\
\quad \quad \quad \quad \quad ((pm'ro_addr \cup pm'rw_addr)), \\
\quad \quad \quad \quad \quad \quad lvl1))))): \\
\quad \quad \text{OK?}[\text{Physical_memory}, ((\text{range_pt}(lvl1)))] \\
\quad \quad \quad (\text{read_data}[\text{Physical_memory}, ((\text{range_pt}(lvl1)))] \\
\quad \quad \quad \quad (pm_phy, \text{paging_data_type}(lvl1))(a_1)(s_{11})))
\end{array}$$

Repeatedly Skolemizing and flattening,
Instantiating quantified variables,

This completes the proof of `is_linear_plain_memory?_TCC5`.

Q.E.D.

C.115.16 Linear_Memory_Blessing.is_linear_plain_memory?_TCC6

Terse proof for `is_linear_plain_memory?_TCC6`.

is_linear_plain_memory?_TCC6:

```

{1}  ∃ (pm: Plain_Memory[Linear_memory[Physical_memory, pm_phy]]):
      (∃ (s: (pm'states), lvl: Level,
          a:
            (pe_in_pt_range?(s,
                          restrict[Address, Memory_Address_4G, boolean]
                                ((pm'ro_addr ∪ pm'rw_addr)),
                                lvl)))):
      OK?(read_data(pm_phy, paging_data_type(lvl))(a)(s)))
  ∧
  (∃ (s1, s2: (pm'states)):
    pe_in_pdir_range?(s1,
                      restrict[Address, Memory_Address_4G, boolean]
                            ((pm'ro_addr ∪ pm'rw_addr)))
    =
    pe_in_pdir_range?(s2,
                      restrict[Address, Memory_Address_4G, boolean]
                            ((pm'ro_addr ∪ pm'rw_addr)))
    ∧
    pe_in_ptab_range?(s1,
                      restrict[Address, Memory_Address_4G, boolean]
                            ((pm'ro_addr ∪ pm'rw_addr)))
    =
    pe_in_ptab_range?(s2,
                      restrict[Address, Memory_Address_4G, boolean]
                            ((pm'ro_addr ∪ pm'rw_addr))))
  ∧
  (∃ (s1, s2: (pm'states)):
    data(read_data(pm_phy, pdir_data_type)(PDIR)(s1)) =
    data(read_data(pm_phy, pdir_data_type)(PDIR)(s2)))
  ∧
  (∃ (s: (pm'states)): OK?(read_data(pm_phy, pdir_data_type)(PDIR)(s))) ∧
  (∃ (s1, s2: (pm'states)):
    data(read_data(pm_phy, segment_reg_data_type)(CS)(s1)) =
    data(read_data(pm_phy, segment_reg_data_type)(CS)(s2)))
  ∧
  (∃ (s: (pm'states)): OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s)))
  ∧ pm'mem = linear_pm
  ⊃
  (∃ (s11, s21: (pm'states), lvl1: Level,
      a1:
        (pe_in_pt_range?(s11,
                          restrict[Address, Memory_Address_4G, boolean]
                                ((pm'ro_addr ∪ pm'rw_addr)),
                                lvl1)))):
    OK?[Physical_memory, ((range_pt(lvl1)))]
      (read_data[Physical_memory, ((range_pt(lvl1)))]
        (pm_phy, paging_data_type(lvl1))(a1)(s21)))

```

Repeatedly Skolemizing and flattening,
Instantiating quantified variables,

Instantiating quantified variables,

we get 2 subgoals:

is_linear_plain_memory?_TCC6.1:

{-1}	pm' states(s'_{11})
{-2}	pm' states(s'_{21})
{-3}	lvl1' < max_level
{-4}	Mem?(a'_1 ' type_of)
{-5}	$0 \leq a'_1$ ' offset
{-6}	a'_1 ' offset < max_linear_offset
{-7}	pe_in_pt_range?(s'_{11} , restrict[Address, Memory_Address_4G, boolean] ((pm'ro_addr \cup pm'rw_addr)), lvl1')
{-8}	pe_in_pdir_range?(s'_{11} , restrict[Address, Memory_Address_4G, boolean] ((pm'ro_addr \cup pm'rw_addr))) = pe_in_pdir_range?(s'_{21} , restrict[Address, Memory_Address_4G, boolean] ((pm'ro_addr \cup pm'rw_addr))) \wedge pe_in_ptab_range?(s'_{11} , restrict[Address, Memory_Address_4G, boolean] ((pm'ro_addr \cup pm'rw_addr))) = pe_in_ptab_range?(s'_{21} , restrict[Address, Memory_Address_4G, boolean] ((pm'ro_addr \cup pm'rw_addr)))
{-9}	$\forall (s_1, s_2: (pm'$ states)): data(read_data(pm_phy, pdbr_data_type)(PDBR)(s_1)) = data(read_data(pm_phy, pdbr_data_type)(PDBR)(s_2))
{-10}	$\forall (s: (pm'$ states)): OK?(read_data(pm_phy, pdbr_data_type)(PDBR)(s))
{-11}	$\forall (s_1, s_2: (pm'$ states)): data(read_data(pm_phy, segment_reg_data_type)(CS)(s_1)) = data(read_data(pm_phy, segment_reg_data_type)(CS)(s_2))
{-12}	$\forall (s: (pm'$ states)): OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-13}	pm' mem = linear_pm
{1}	pe_in_pt_range?(s'_{21} , restrict[Address, Memory_Address_4G, boolean] ((pm'ro_addr \cup pm'rw_addr)), lvl1')
{2}	OK?[Physical_memory, ((range_pt(lvl1')))] (read_data[Physical_memory, ((range_pt(lvl1')))] (pm_phy, paging_data_type(lvl1'))(a'_1)(s'_{21}))

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Expanding the definition of pe_in_pt_range?,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of is_linear_plain_memory?_TCC6.1.

is_linear_plain_memory?_TCC6.2:

<pre> {-1} pm' 'states(s'11) {-2} pm' 'states(s'21) {-3} lvl1' < max_level {-4} Mem?(a'1 'type_of) {-5} 0 ≤ a'1 'offset {-6} a'1 'offset < max_linear_offset {-7} pe_in_pt_range?(s'11, restrict[Address, Memory_Address_4G, boolean] ((pm' 'ro_addr ∪ pm' 'rw_addr)), lvl1') (a'1) {-8} ∀ (s: (pm' 'states), lvl: Level, a: (pe_in_pt_range?(s, restrict[Address, Memory_Address_4G, boolean] ((pm' 'ro_addr ∪ pm' 'rw_addr)), lvl)))): OK?(read_data(pm_phy, paging_data_type(lvl))(a)(s)) {-9} pe_in_pdir_range?(s'11, restrict[Address, Memory_Address_4G, boolean] ((pm' 'ro_addr ∪ pm' 'rw_addr))) = pe_in_pdir_range?(s'21, restrict[Address, Memory_Address_4G, boolean] ((pm' 'ro_addr ∪ pm' 'rw_addr))) ∧ pe_in_ptab_range?(s'11, restrict[Address, Memory_Address_4G, boolean] ((pm' 'ro_addr ∪ pm' 'rw_addr))) = pe_in_ptab_range?(s'21, restrict[Address, Memory_Address_4G, boolean] ((pm' 'ro_addr ∪ pm' 'rw_addr))) {-10} ∀ (s1, s2: (pm' 'states)): data(read_data(pm_phy, pdir_data_type)(PDBR)(s1)) = data(read_data(pm_phy, pdir_data_type)(PDBR)(s2)) {-11} ∀ (s: (pm' 'states)): OK?(read_data(pm_phy, pdir_data_type)(PDBR)(s)) {-12} ∀ (s1, s2: (pm' 'states)): data(read_data(pm_phy, segment_reg_data_type)(CS)(s1)) = data(read_data(pm_phy, segment_reg_data_type)(CS)(s2)) {-13} ∀ (s: (pm' 'states)): OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s)) {-14} pm' 'mem = linear_pm </pre>	<pre> {1} pe_in_pt_range?(s'21, restrict[Address, Memory_Address_4G, boolean] ((pm' 'ro_addr ∪ pm' 'rw_addr)), lvl1') (a'1) {2} OK?[Physical_memory, ((range_pt(lvl1')))] (read_data[Physical_memory, ((range_pt(lvl1')))] (pm_phy, paging_data_type(lvl1'))(a'1)(s'21)) </pre>
--	---

Expanding the definition of pe_in_pt_range?,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 This completes the proof of is_linear_plain_memory?_TCC6.2.
 Q.E.D.

C.116 Proofs for Linear_Memory_Blessing_Properties (challenge-linear.pvs)

C.116.1

Linear_Memory_Blessing_Properties.linear_plain_transformers_ok

Terse proof for linear_plain_transformers_ok.

linear_plain_transformers_ok:

$\{1\} \quad \forall (pm: \text{Plain_Memory}[\text{Physical_memory}]):$ $\text{is_linear_plain_memory?}(pm) \supset$ $\text{transformers_ok?}(pm' \text{states},$ $\quad ((\text{memory_read_transformers}(pm' \text{mem}, (pm' \text{ro_addr} \cup pm' \text{rw_addr})) \cup \text{memory_write}.$	
---	--

Installing automatic rewrites from: (linear_resolve_read_transformers_ok linear_resolve_write_transformers_ok
 linear_resolve_transformer_invariant singleton member)

Repeatedly Skolemizing and flattening,

Rewriting using transformers_ok_all_transformers, matching in *,

Hiding formulas: 2,

Repeatedly Skolemizing and flattening,

Expanding the definition of union,

Using lemma pm_states,

Using lemma pm_plain_phy,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 4 subgoals:

linear_plain_transformers_ok.1:

$\{-1\} \quad \text{is_linear_plain_memory?}(pm')$ $\{-2\} \quad \text{plain_memory?}(pm_phy)$ $\{-3\} \quad pm' \text{states} = pm_phy' \text{states}$ $\{-4\} \quad \text{memory_read_transformers}(pm' \text{mem}, (pm' \text{ro_addr} \cup pm' \text{rw_addr}))(q')$	$\{1\} \quad \text{transformers_ok?}(pm' \text{states}, \text{singleton}(q'))$
--	---

Expanding the definition of memory_read_transformers,

Repeatedly Skolemizing and flattening,

Replacing using formula -5,

Hiding formulas: -5,

Rewriting using pm_read_linear, matching in *,

Expanding the definition of linear_read,

Using lemma pm_memory_addr,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Rewriting using fexpr_composition_transformers_ok, matching in * where P gets LAMBDA
 (pa: Address): FORALL (s: (pm!1'states)): OK?(linear_resolve(a!1, Read)(s)) IMPLIES pa =
 data(linear_resolve(a!1, Read)(s)),

we get 4 subgoals:

`linear_plain_transformers_ok.1.1:`

{-1}	<code>is_linear_plain_memory?(pm')</code>
{-2}	<code>union(pm' 'ro_addr, pm' 'rw_addr)(a')</code>
{-3}	<code>in_memory(min_linear, max_linear)(a')</code>
{-4}	<code>Mem?(type_of(a'))</code>
{-5}	<code>plain_memory?(pm_phy)</code>
{-6}	<code>pm' 'states = pm_phy' 'states</code>
{1}	<code>transformers_ok?(pm' 'states, singleton(expr_2_super(linear_resolve(a', Read))))</code>
{2}	<code>transformers_ok?(pm' 'states, singleton(expr_2_super(linear_resolve(a', Read) ## (λ (pa: Address): mem-ory_read(pm_phy' mem)(pa))))</code>

Hiding formulas: 2,

Using lemma `linear_resolve_read_transformers_ok`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Keeping (-3 1) and hiding *,

Expanding the definition of `transformers_ok?`,

Repeatedly Skolemizing and flattening,

Instantiating the top quantifier in -1 with the terms: (s!1 q!2),

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Replacing using formula -2,

Hiding formulas: (-2 2),

Applying `decompose-equality`,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `linear_plain_transformers_ok.1.1`.

`linear_plain_transformers_ok.1.2:`

{-1}	<code>is_linear_plain_memory?(pm')</code>
{-2}	<code>union(pm' 'ro_addr, pm' 'rw_addr)(a')</code>
{-3}	<code>in_memory(min_linear, max_linear)(a')</code>
{-4}	<code>Mem?(type_of(a'))</code>
{-5}	<code>plain_memory?(pm_phy)</code>
{-6}	<code>pm' 'states = pm_phy' 'states</code>
{1}	<code>transformer_invariant?(pm' 'states, singleton(expr_2_super(linear_resolve(a', Read))))</code>
{2}	<code>transformers_ok?(pm' 'states, singleton(expr_2_super(linear_resolve(a', Read) ## (λ (pa: Address): mem-ory_read(pm_phy' mem)(pa))))</code>

Hiding formulas: 2,

Using lemma `linear_resolve_transformer_invariant`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Keeping (-3 1) and hiding *,

Expanding the definition of `transformer_invariant?`,

Repeatedly Skolemizing and flattening,

Instantiating quantified variables,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Replacing using formula -2,

Applying `decompose-equality`,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `linear_plain_transformers_ok.1.2`.

`linear_plain_transformers_ok.1.3`:

{-1}	<code>is_linear_plain_memory?(pm')</code>
{-2}	<code>union(pm'ro_addr, pm'rw_addr)(a')</code>
{-3}	<code>in_memory(min_linear, max_linear)(a')</code>
{-4}	<code>Mem?(type_of(a'))</code>
{-5}	<code>plain_memory?(pm_phy)</code>
{-6}	<code>pm'states = pm_phy'states</code>
<hr/>	
{1}	$\forall (s_1: (pm'states)):$ $OK?(linear_resolve(a', Read)(s_1)) \supset$ $(\forall (s: (pm'states)):$ $OK?(linear_resolve(a', Read)(s)) \supset$ $data(linear_resolve(a', Read)(s_1)) = data(linear_resolve(a', Read)(s)))$
{2}	<code>transformers_ok?(pm'states,</code> $singleton(expr_2_super(linear_resolve(a', Read) \#\#$ $(\lambda (pa: Address):$ $mem-$ $ory_read(pm_phy'mem)(pa))))$

Hiding formulas: 2,

Repeatedly Skolemizing and flattening,

Using lemma `linear_resolve_same_result`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `linear_plain_transformers_ok.1.3`.

`linear_plain_transformers_ok.1.4`:

{-1}	<code>is_linear_plain_memory?(pm')</code>
{-2}	<code>union(pm'ro_addr, pm'rw_addr)(a')</code>
{-3}	<code>in_memory(min_linear, max_linear)(a')</code>
{-4}	<code>Mem?(type_of(a'))</code>
{-5}	<code>plain_memory?(pm_phy)</code>
{-6}	<code>pm'states = pm_phy'states</code>
<hr/>	
{1}	$\forall (d:$ $(\lambda (pa: Address):$ $\forall (s: (pm'states)):$ $OK?(linear_resolve(a', Read)(s)) \supset$ $pa = data(linear_resolve(a', Read)(s)))):$ $transformers_ok?(pm'states, singleton(expr_2_super(memory_read(pm_phy'mem)(d))))$
{2}	<code>transformers_ok?(pm'states,</code> $singleton(expr_2_super(linear_resolve(a', Read) \#\#$ $(\lambda (pa: Address):$ $mem-$ $ory_read(pm_phy'mem)(pa))))$

Repeatedly Skolemizing and flattening,

Hiding formulas: 2,

Using lemma `plain_memory_transformers_ok_read_ro_rw`,

Using lemma `transformers_ok_mono_transformers`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Expanding the definition of `transformers_ok?`,

Expanding the definition of `subset?`,

Repeatedly Skolemizing and flattening,

Hiding formulas: (-3 -5 2),

C Proof scripts

Instantiating quantified variables,
 Using lemma `pm_linear_resolve_read_ok`,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 Replacing using formula -3,
 Expanding the definition of `memory_read_transformers`,
 Instantiating quantified variables,
 Using lemma `pm_linear_blessed`,
 Expanding the definition of `linear_blessed?`,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 Hiding formulas: (-1 -2 -4 -5 -6 -7),
 Expanding the definition of `restrict`,
 Expanding the definition of `subset?`,
 Expanding the definition of `virt_to_phys_range`,
 Instantiating the top quantifier in -1 with the terms: (d!1),
 we get 2 subgoals:

`linear_plain_transformers_ok.1.4.1:`

{-1}	$(d' \in \lambda (\text{phy_a} : \text{Memory_Address_4G}) : \exists (\text{lin_a} : ((\lambda (s : \text{Memory_Address_4G}) : \text{pm}'\text{ro_addr} \Rightarrow (d' \in \lambda (s : \text{Memory_Address_4G}) : \text{union}(\text{pm_phy}'\text{ro_addr}, \text{pm_phy}'\text{rw_addr})(s))))$
{-2}	<code>is_linear_plain_memory?(pm')</code>
{-3}	<code>OK?(linear_resolve(a', Read)(s'))</code>
{-4}	$x' = \text{expr_2_super}(\text{memory_read}(\text{pm_phy}'\text{mem})(d'))$
{-5}	<code>pm}'states(s')</code>
{-6}	<code>plain_memory?(pm_phy)</code>
{-7}	$d' = \text{data}(\text{linear_resolve}(a', \text{Read})(s'))$
{-8}	<code>union(pm}'ro_addr, pm}'rw_addr)(a')</code>
{-9}	<code>in_memory(min_linear, max_linear)(a')</code>
{-10}	<code>Mem?(type_of(a'))</code>
{-11}	<code>pm}'states = pm_phy'states</code>
{1}	$\text{union}(\text{pm_phy}'\text{ro_addr}, \text{pm_phy}'\text{rw_addr})(d')$

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 Instantiating quantified variables,
 we get 3 subgoals:

`linear_plain_transformers_ok.1.4.1.1:`

{-1}	<code>is_linear_plain_memory?(pm')</code>
{-2}	<code>OK?(linear_resolve(a', Read)(s'))</code>
{-3}	$x' = \text{expr_2_super}(\text{memory_read}(\text{pm_phy}'\text{mem})(d'))$
{-4}	<code>pm}'states(s')</code>
{-5}	<code>plain_memory?(pm_phy)</code>
{-6}	$d' = \text{data}(\text{linear_resolve}(a', \text{Read})(s'))$
{-7}	<code>union(pm}'ro_addr, pm}'rw_addr)(a')</code>
{-8}	<code>in_memory(min_linear, max_linear)(a')</code>
{-9}	<code>Mem?(type_of(a'))</code>
{-10}	<code>pm}'states = pm_phy'states</code>
{1}	$\text{OK}?(\text{linear_resolve}(a', \text{Read})(s')) \wedge \text{data}(\text{linear_resolve}(a', \text{Read})(s')) = d'$
{2}	$\text{union}(\text{pm_phy}'\text{ro_addr}, \text{pm_phy}'\text{rw_addr})(d')$

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 This completes the proof of `linear_plain_transformers_ok.1.4.1.1`.

linear_plain_transformers_ok.1.4.1.2:

{-1}	is_linear_plain_memory?(pm')
{-2}	OK?(linear_resolve(a', Read)(s'))
{-3}	x' = expr_2_super(memory_read(pm_phy' mem)(d'))
{-4}	pm' states(s')
{-5}	plain_memory?(pm_phy)
{-6}	d' = data(linear_resolve(a', Read)(s'))
{-7}	union(pm' ro_addr, pm' rw_addr)(a')
{-8}	in_memory(min_linear, max_linear)(a')
{-9}	Mem?(type_of(a'))
{-10}	pm' states = pm_phy' states
{1}	union[Memory_access] (singleton[Memory_access](Read), singleton[Memory_access](Execute))(Read)
{2}	union(pm_phy' ro_addr, pm_phy' rw_addr)(d')

Keeping (1) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of linear_plain_transformers_ok.1.4.1.2.

linear_plain_transformers_ok.1.4.1.3:

{-1}	is_linear_plain_memory?(pm')
{-2}	OK?(linear_resolve(a', Read)(s'))
{-3}	x' = expr_2_super(memory_read(pm_phy' mem)(d'))
{-4}	pm' states(s')
{-5}	plain_memory?(pm_phy)
{-6}	d' = data(linear_resolve(a', Read)(s'))
{-7}	union(pm' ro_addr, pm' rw_addr)(a')
{-8}	in_memory(min_linear, max_linear)(a')
{-9}	Mem?(type_of(a'))
{-10}	pm' states = pm_phy' states
{1}	union[Memory_Address_4G] (λ (s: Memory_Address_4G): pm' ro_addr(s), λ (s: Memory_Address_4G): pm' rw_addr(s)) (a')
{2}	union(pm_phy' ro_addr, pm_phy' rw_addr)(d')

Keeping (-7 1) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of linear_plain_transformers_ok.1.4.1.3.

linear_plain_transformers_ok.1.4.2:

{-1}	is_linear_plain_memory?(pm')
{-2}	OK?(linear_resolve(a', Read)(s'))
{-3}	x' = expr_2_super(memory_read(pm_phy' mem)(d'))
{-4}	pm' states(s')
{-5}	plain_memory?(pm_phy)
{-6}	d' = data(linear_resolve(a', Read)(s'))
{-7}	union(pm' ro_addr, pm' rw_addr)(a')
{-8}	in_memory(min_linear, max_linear)(a')
{-9}	Mem?(type_of(a'))
{-10}	pm' states = pm_phy' states
{1}	Mem?(d' type_of) ∧ 0 ≤ d' offset ∧ d' offset < max_linear_offset
{2}	union(pm_phy' ro_addr, pm_phy' rw_addr)(d')

C Proof scripts

Using lemma `linear_resolve_memory_address`,
 Expanding the definition of `every`,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 This completes the proof of `linear_plain_transformers_ok.1.4.2`.

`linear_plain_transformers_ok.2`:

{-1}	<code>is_linear_plain_memory?(pm')</code>
{-2}	<code>plain_memory?(pm_phy)</code>
{-3}	<code>pm' 'states = pm_phy' 'states</code>
{-4}	<code>memory_write_transformers(pm' 'mem, pm' 'rw_addr)(q')</code>
{1}	<code>transformers_ok?(pm' 'states, singleton(q'))</code>

Expanding the definition of `memory_write_transformers`,
 Repeatedly Skolemizing and flattening,
 Replacing using formula -6,
 Hiding formulas: -6,
 Rewriting using `pm_write_linear`, matching in *,
 Expanding the definition of `linear_write`,
 Using lemma `pm_memory_addr`,
 Installing automatic rewrites from: `union_right`
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 Rewriting using `fexpr_composition_transformers_ok`, matching in * where P gets LAMBDA
 (pa: Address): FORALL (s: (pm!1'states)): OK?(linear_resolve(a!1, Write)(s)) IMPLIES pa =
 data(linear_resolve(a!1, Write)(s)),
 we get 4 subgoals:

`linear_plain_transformers_ok.2.1`:

{-1}	<code>is_linear_plain_memory?(pm')</code>
{-2}	<code>in_memory(min_linear, max_linear)(a')</code>
{-3}	<code>Mem?(type_of(a'))</code>
{-4}	<code>b' < max_byte</code>
{-5}	<code>plain_memory?(pm_phy)</code>
{-6}	<code>pm' 'states = pm_phy' 'states</code>
{-7}	<code>pm' 'rw_addr(a')</code>
{1}	<code>transformers_ok?(pm' 'states, singleton(expr_2_super(linear_resolve(a', Write))))</code>
{2}	<code>transformers_ok?(pm' 'states, singleton(expr_2_super(linear_resolve(a', Write) ## (λ (pa: Address): mem-ory_write(pm_phy' mem)(pa, b'))))</code>

Hiding formulas: 2,
 Using lemma `linear_resolve_write_transformers_ok`,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 Keeping (-3 1) and hiding *,
 Expanding the definition of `transformers_ok?`,
 Repeatedly Skolemizing and flattening,
 Instantiating quantified variables,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 Replacing using formula -2,
 Applying `decompose-equality`,
 Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of `linear_plain_transformers_ok.2.1`.

linear_plain_transformers_ok.2.2:

<pre> {-1} is_linear_plain_memory?(pm') {-2} in_memory(min_linear, max_linear)(a') {-3} Mem?(type_of(a')) {-4} b' < max_byte {-5} plain_memory?(pm_phy) {-6} pm' 'states = pm_phy 'states {-7} pm' 'rw_addr(a') </pre>	<pre> {1} transformer_invariant?(pm' 'states, singleton(expr_2_super(linear_resolve(a', Write)))) {2} transformers_ok?(pm' 'states, singleton(expr_2_super(linear_resolve(a', Write) ## (λ (pa: Address): mem- ory_write(pm_phy 'mem)(pa, b'))))) </pre>
---	--

Hiding formulas: 2,

Using lemma linear_resolve_transformer_invariant,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Keeping (-2 1) and hiding *,

Expanding the definition of transformer_invariant?,

Repeatedly Skolemizing and flattening,

Instantiating quantified variables,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Replacing using formula -2,

Applying decompose-equality,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of linear_plain_transformers_ok.2.2.

linear_plain_transformers_ok.2.3:

<pre> {-1} is_linear_plain_memory?(pm') {-2} in_memory(min_linear, max_linear)(a') {-3} Mem?(type_of(a')) {-4} b' < max_byte {-5} plain_memory?(pm_phy) {-6} pm' 'states = pm_phy 'states {-7} pm' 'rw_addr(a') </pre>	<pre> {1} ∀ (s_1: (pm' 'states)): OK?(linear_resolve(a', Write)(s_1)) ⊃ (∀ (s: (pm' 'states)): OK?(linear_resolve(a', Write)(s)) ⊃ data(linear_resolve(a', Write)(s_1)) = data(linear_resolve(a', Write)(s))) {2} transformers_ok?(pm' 'states, singleton(expr_2_super(linear_resolve(a', Write) ## (λ (pa: Address): mem- ory_write(pm_phy 'mem)(pa, b'))))) </pre>
---	--

Hiding formulas: 2,

Repeatedly Skolemizing and flattening,

Using lemma linear_resolve_same_result,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of linear_plain_transformers_ok.2.3.

linear_plain_transformers_ok.2.4:

{-1}	is_linear_plain_memory?(pm')
{-2}	in_memory(min_linear, max_linear)(a')
{-3}	Mem?(type_of(a'))
{-4}	b' < max_byte
{-5}	plain_memory?(pm_phy)
{-6}	pm' 'states = pm_phy 'states
{-7}	pm' 'rw_addr(a')
{1}	$\forall (d: (\lambda (pa: \text{Address}): \forall (s: (\text{pm}' \text{'states})): \text{OK}?(linear_resolve(a', \text{Write})(s)) \supset pa = \text{data}(linear_resolve(a', \text{Write})(s)))): \text{transformers_ok}?(pm' \text{'states}, \text{singleton}(\text{expr_2_super}(\text{memory_write}(\text{pm_phy}' \text{'mem})(d, b'))))$
{2}	$\text{transformers_ok}?(pm' \text{'states}, \text{singleton}(\text{expr_2_super}(linear_resolve(a', \text{Write}) \#\# (\lambda (pa: \text{Address}): \text{memory_write}(\text{pm_phy}' \text{'mem})(pa, b')))))$

Repeatedly Skolemizing and flattening,

Hiding formulas: 2,

Using lemma plain_memory_transformers_ok_write_block,

Using lemma transformers_ok_mono_transformers,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 3 subgoals:

linear_plain_transformers_ok.2.4.1:

{-1}	plain_memory?(pm_phy)
{-2}	$\forall (s: (\text{pm}' \text{'states})): \text{OK}?(linear_resolve(a', \text{Write})(s)) \supset d' = \text{data}(linear_resolve(a', \text{Write})(s))$
{-3}	is_linear_plain_memory?(pm')
{-4}	in_memory(min_linear, max_linear)(a')
{-5}	Mem?(type_of(a'))
{-6}	b' < max_byte
{-7}	pm' 'states = pm_phy 'states
{-8}	pm' 'rw_addr(a')
{1}	transformers_ok?(pm' 'states, memory_write_transformers(pm_phy 'mem, address_block(d', 1)))
{2}	transformers_ok?(pm' 'states, singleton(expr_2_super(memory_write(pm_phy 'mem)(d', b'))))
{3}	(address_block(d', 1) \subseteq pm_phy 'rw_addr)

Expanding the definition of transformers_ok?,

Repeatedly Skolemizing and flattening,

Hiding formulas: (1 2),

Instantiating quantified variables,

Expanding the definition of address_block,

Hiding formulas: (-2),

Using lemma pm_linear_resolve_write_ok,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Using lemma pm_linear_blessed,

Expanding the definition of linear_blessed?,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Hiding formulas: (-1 -2 -3 -5 -6 -7),

Expanding the definition of subset?,

Repeatedly Skolemizing and flattening,

Instantiating quantified variables,

we get 3 subgoals:

`linear_plain_transformers_ok.2.4.1.1:`

{-1}	$(x' \in \text{virt_to_phys_range}(s', \text{restrict} [\text{Address}, \text{Memory_Address_4G}, \text{boolean}] (\text{pm}'\text{rw_addr}), \text{singleton}(\text{Write}(a', \text{Write}(s'))))) \Rightarrow (x' \in \text{restrict} [\text{Address}, \text{Memory_Address_4G}, \text{boolean}] (\text{pm_phy}'\text{rw_addr}))$
{-2}	<code>is_linear_plain_memory?(pm')</code>
{-3}	<code>OK?(linear_resolve(a', Write)(s'))</code>
{-4}	<code>pm' 'states(s')</code>
{-5}	<code>plain_memory?(pm_phy)</code>
{-6}	$(x' \in \lambda (a : \text{Address}) : \text{type_of}(d') = \text{type_of}(a) \wedge \text{offset}(d') \leq \text{offset}(a) \wedge \text{offset}(a) < 1 + \text{offset}(d'))$
{-7}	$d' = \text{data}(\text{linear_resolve}(a', \text{Write})(s'))$
{-8}	<code>in_memory(min_linear, max_linear)(a')</code>
{-9}	<code>Mem?(type_of(a'))</code>
{-10}	$b' < \text{max_byte}$
{-11}	<code>pm' 'states = pm_phy' 'states</code>
{-12}	<code>pm' 'rw_addr(a')</code>
{1}	$(x' \in \text{pm_phy}'\text{rw_addr})$

Expanding the definition of restrict,

Expanding the definition of virt_to_phys_range,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Instantiating quantified variables,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Case splitting on $d!1 = x!1$,

we get 2 subgoals:

`linear_plain_transformers_ok.2.4.1.1.1:`

{-1}	$d' = x'$
{-2}	<code>OK?(linear_resolve(a', Write)(s'))</code>
{-3}	<code>is_linear_plain_memory?(pm')</code>
{-4}	<code>pm' 'states(s')</code>
{-5}	<code>plain_memory?(pm_phy)</code>
{-6}	$\text{type_of}(d') = \text{type_of}(x')$
{-7}	$\text{offset}(d') \leq \text{offset}(x')$
{-8}	$\text{offset}(x') < 1 + \text{offset}(d')$
{-9}	$d' = \text{data}(\text{linear_resolve}(a', \text{Write})(s'))$
{-10}	<code>in_memory(min_linear, max_linear)(a')</code>
{-11}	<code>Mem?(type_of(a'))</code>
{-12}	$b' < \text{max_byte}$
{-13}	<code>pm' 'states = pm_phy' 'states</code>
{-14}	<code>pm' 'rw_addr(a')</code>
{1}	$\text{data}(\text{linear_resolve}(a', \text{Write})(s')) = x'$
{2}	<code>pm_phy' 'rw_addr(x')</code>

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `linear_plain_transformers_ok.2.4.1.1.1`.

C Proof scripts

linear_plain_transformers_ok.2.4.1.1.2:

{-1}	OK?(linear_resolve(a' , Write)(s'))
{-2}	is_linear_plain_memory?(pm')
{-3}	pm' 'states(s')
{-4}	plain_memory?(pm_phy)
{-5}	$type_of(d') = type_of(x')$
{-6}	$offset(d') \leq offset(x')$
{-7}	$offset(x') < 1 + offset(d')$
{-8}	$d' = data(linear_resolve(a', Write)(s'))$
{-9}	in_memory(min_linear, max_linear)(a')
{-10}	Mem?($type_of(a')$)
{-11}	$b' < max_byte$
{-12}	pm' 'states = pm_phy' states
{-13}	pm' 'rw_addr(a')
{1}	$d' = x'$
{2}	$data(linear_resolve(a', Write)(s')) = x'$
{3}	pm_phy' rw_addr(x')

Keeping (-5 -6 -7 1) and hiding *,

Applying decompose-equality,

This completes the proof of linear_plain_transformers_ok.2.4.1.1.2.

linear_plain_transformers_ok.2.4.1.2:

{-1}	is_linear_plain_memory?(pm')
{-2}	OK?(linear_resolve(a' , Write)(s'))
{-3}	pm' 'states(s')
{-4}	plain_memory?(pm_phy)
{-5}	$(x' \in \lambda (a : Address) : type_of(d') = type_of(a) \wedge offset(d') \leq offset(a) \wedge offset(a) < 1$
{-6}	$d' = data(linear_resolve(a', Write)(s'))$
{-7}	in_memory(min_linear, max_linear)(a')
{-8}	Mem?($type_of(a')$)
{-9}	$b' < max_byte$
{-10}	pm' 'states = pm_phy' states
{-11}	pm' 'rw_addr(a')
{1}	Mem?(x' 'type_of) $\wedge 0 \leq x'$ 'offset $\wedge x'$ 'offset $< max_linear_offset$
{2}	$(x' \in pm_phy'$ rw_addr)

Using lemma linear_resolve_memory_address,

Expanding the definition of every,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of linear_plain_transformers_ok.2.4.1.2.

linear_plain_transformers_ok.2.4.1.3:

{-1}	$\forall (x: \text{Memory_Address_4G}):$ $(x \in \text{virt_to_phys_range}(s', \text{restrict} [\text{Address}, \text{Memory_Address_4G}, \text{boolean}](\text{pm}'\text{rw_addr}), \text{singleton}(W)))$ $\Rightarrow (x \in \text{restrict} [\text{Address}, \text{Memory_Address_4G}, \text{boolean}](\text{pm_phy}'\text{rw_addr}))$
{-2}	$\text{is_linear_plain_memory?}(\text{pm}')$
{-3}	$\text{OK?}(\text{linear_resolve}(a', \text{Write})(s'))$
{-4}	$\text{pm}'\text{states}(s')$
{-5}	$\text{plain_memory?}(\text{pm_phy})$
{-6}	$(x' \in \lambda (a: \text{Address}): \text{type_of}(d') = \text{type_of}(a) \wedge \text{offset}(d') \leq \text{offset}(a) \wedge \text{offset}(a) < 1 + \text{offset}(d'))$
{-7}	$d' = \text{data}(\text{linear_resolve}(a', \text{Write})(s'))$
{-8}	$\text{in_memory}(\text{min_linear}, \text{max_linear})(a')$
{-9}	$\text{Mem?}(\text{type_of}(a'))$
{-10}	$b' < \text{max_byte}$
{-11}	$\text{pm}'\text{states} = \text{pm_phy}'\text{states}$
{-12}	$\text{pm}'\text{rw_addr}(a')$
{1}	$\text{Mem?}(x'\text{type_of}) \wedge 0 \leq x'\text{offset} \wedge x'\text{offset} < \text{max_linear_offset}$
{2}	$(x' \in \text{pm_phy}'\text{rw_addr})$

Using lemma linear_resolve_memory_address,

Expanding the definition of every,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of linear_plain_transformers_ok.2.4.1.3.

linear_plain_transformers_ok.2.4.2:

{-1}	$\text{plain_memory?}(\text{pm_phy})$
{-2}	$\text{transformers_ok?}(\text{pm_phy}'\text{states}, \text{memory_write_transformers}(\text{pm_phy}'\text{mem}, \text{address_block}(d', 1)))$
{-3}	$\forall (s: (\text{pm}'\text{states})):$ $\text{OK?}(\text{linear_resolve}(a', \text{Write})(s)) \supset d' = \text{data}(\text{linear_resolve}(a', \text{Write})(s))$
{-4}	$\text{is_linear_plain_memory?}(\text{pm}')$
{-5}	$\text{in_memory}(\text{min_linear}, \text{max_linear})(a')$
{-6}	$\text{Mem?}(\text{type_of}(a'))$
{-7}	$b' < \text{max_byte}$
{-8}	$\text{pm}'\text{states} = \text{pm_phy}'\text{states}$
{-9}	$\text{pm}'\text{rw_addr}(a')$
{1}	$(\text{singleton}(\text{expr_2_super}(\text{memory_write}(\text{pm_phy}'\text{mem})(d', b')))) \subseteq \text{memory_write_transformers}(\text{pm_phy}'\text{mem}, \text{address_block}(d', 1))$
{2}	$\text{transformers_ok?}(\text{pm}'\text{states}, \text{singleton}(\text{expr_2_super}(\text{memory_write}(\text{pm_phy}'\text{mem})(d', b'))))$

Expanding the definition of subset?,

Repeatedly Skolemizing and flattening,

Simplifying, rewriting, and recording with decision procedures,

Replacing using formula -1,

Expanding the definition of memory_write_transformers,

Instantiating quantified variables,

Expanding the definition of address_block,

which is trivially true.

This completes the proof of linear_plain_transformers_ok.2.4.2.

C Proof scripts

`linear_plain_transformers_ok.2.4.3:`

{-1}	<code>plain_memory?(pm_phy)</code>
{-2}	$\forall (s: (pm' \text{'states})): \text{OK?}(\text{linear_resolve}(a', \text{Write})(s)) \supset d' = \text{data}(\text{linear_resolve}(a', \text{Write})(s))$
{-3}	<code>is_linear_plain_memory?(pm')</code>
{-4}	<code>in_memory(min_linear, max_linear)(a')</code>
{-5}	<code>Mem?(type_of(a'))</code>
{-6}	<code>b' < max_byte</code>
{-7}	<code>pm' \text{'states} = pm_phy \text{'states}</code>
{-8}	<code>pm' \text{'rw_addr}(a')</code>
{1}	$(\text{singleton}(\text{expr_2_super}(\text{memory_write}(\text{pm_phy}'\text{mem})(d', b')))) \subseteq \text{memory_write_transformers}(\text{pm_phy}'\text{mem})$
{2}	$\text{transformers_ok?}(\text{pm}' \text{'states}, \text{singleton}(\text{expr_2_super}(\text{memory_write}(\text{pm_phy}'\text{mem})(d', b'))))$
{3}	$(\text{address_block}(d', 1) \subseteq \text{pm_phy}'\text{rw_addr})$

Expanding the definition of `subset?`,

Repeatedly Skolemizing and flattening,

Simplifying, rewriting, and recording with decision procedures,

Replacing using formula -1,

Expanding the definition of `memory_write_transformers`,

Instantiating quantified variables,

Expanding the definition of `address_block`,

which is trivially true.

This completes the proof of `linear_plain_transformers_ok.2.4.3`.

`linear_plain_transformers_ok.3:`

{-1}	<code>is_linear_plain_memory?(pm')</code>
{-2}	<code>plain_memory?(pm_phy)</code>
{-3}	<code>pm' \text{'states} = pm_phy \text{'states}</code>
{-4}	<code>memory_read_side_effect_super_transformers(pm' \text{'mem}, (pm' \text{'ro_addr} \cup pm' \text{'rw_addr}))(q')</code>
{1}	$\text{transformers_ok?}(\text{pm}' \text{'states}, \text{singleton}(q'))$

Expanding the definition of `memory_read_side_effect_super_transformers`,

Repeatedly Skolemizing and flattening,

Replacing using formula -6,

Hiding formulas: (-1 -6),

Rewriting using `pm_read_side_effect_linear`, matching in `*`,

Expanding the definition of `linear_read_side_effect`,

Case splitting on `null?(bl!1) OR (reg_base(min_linear)(type_of(a!1)) <= a!1 AND a!1 + length(bl!1) <= reg_size(max_linear)(type_of(a!1)))`,

we get 4 subgoals:

linear_plain_transformers_ok.3.1:

<pre> {-1} null?(bl') ∨ (reg_base(min_linear)(type_of(a')) ≤ a' ∧ a' + length(bl') ≤ reg_size(max_linear)(type_of(a'))) {-2} is_linear_plain_memory?(pm') {-3} plain_memory?(pm_phy) {-4} pm' 'states = pm_phy 'states {-5} (address_block(a', length(bl')) ⊆ (pm' 'ro_addr ∪ pm' 'rw_addr)) </pre>	<pre> {1} transformers_ok?(pm' 'states, singleton(expr_2_super(λ (s: Linear_memory[Physical_memory, pm_phy]): IF null?(bl') ∨ (reg_base(min_linear)(type_of(a')) ≤ a' ∧ a' + length(bl') ≤ reg_size(max_linear)(type_of(a'))) THEN IF Mem?(type_of(a')) THEN ap- (min_page, a', bl'), lin- (s) ELSE mem- (a', bl', cp')(s) ENDIF ELSE Fatal ENDIF))) </pre>
--	--

Case splitting on Mem?(type_of(a!1)),

we get 2 subgoals:

linear_plain_transformers_ok.3.1.1.1:

{-1}	Mem?(type_of(a'))	
{-2}	null?(bl') ∨ (reg_base(min_linear)(type_of(a')) ≤ a' ∧ a' + length(bl') ≤ reg_size(max_linear)(type_of(a')))	
{-3}	is_linear_plain_memory?(pm')	
{-4}	plain_memory?(pm_phy)	
{-5}	pm' 'states = pm_phy' 'states	
{-6}	(address_block(a', length(bl')) ⊆ (pm' 'ro_addr ∪ pm' 'rw_addr))	
{1}	transformers_ok?(pm' 'states, singleton(expr_2_super(λ (s: Linear_memory[Physical_memory, pm_phy]): IF null?(bl') ∨ (reg_base(min_linear)(type_of(a')) ≤ a' ∧ a' + length(bl') ≤ reg_size(max_linear)(type_of(a')) THEN IF Mem?(type_of(a')) THEN ap- ply_side_effects(split ear_read_side_effect_in_page) ory_read_side_effect(pm_phy' 'mem) ELSE mem- (a', bl', cp')(s) ENDIF ELSE Fatal ENDIF)))	(min a' bl' lin- (s)

Replacing using formula -2,
Simplifying, rewriting, and recording with decision procedures,
Case splitting on null?(bl'),
we get 2 subgoals:

linear_plain_transformers_ok.3.1.1.1.1:

{-1}	null?(bl')	
{-2}	Mem?(type_of(a'))	
{-3}	null?(bl') ∨ (reg_base(min_linear)(type_of(a')) ≤ a' ∧ a' + length(bl') ≤ reg_size(max_linear)(type_of(a')))	
{-4}	is_linear_plain_memory?(pm')	
{-5}	plain_memory?(pm_phy)	
{-6}	pm' 'states = pm_phy' 'states	
{-7}	(address_block(a', length(bl')) ⊆ (pm' 'ro_addr ∪ pm' 'rw_addr))	
{1}	transformers_ok?(pm' 'states, singleton(expr_2_super(λ (s: Linear_memory[Physical_memory, pm_phy]): apply_side_effects(split(min_page, a', bl'), lin- ear_read_side_effect_in_page) (s))))	(s))))

Hiding formulas: -3,
 Using lemma split_null,
 Expanding the definition of apply_side_effects,
 Expanding the definition of reduce,
 Expanding the definition of ok_result,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 Keeping (1) and hiding *,
 Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of linear_plain_transformers_ok.3.1.1.1.
 linear_plain_transformers_ok.3.1.1.2:

<pre> {-1} Mem?(type_of(a')) {-2} null?(bl') ∨ (reg_base(min_linear)(type_of(a')) ≤ a' ∧ a' + length(bl') ≤ reg_size(max_linear)(type_of(a'))) {-3} is_linear_plain_memory?(pm') {-4} plain_memory?(pm_phy) {-5} pm' 'states = pm_phy 'states {-6} (address_block(a', length(bl')) ⊆ (pm' 'ro_addr ∪ pm' 'rw_addr)) </pre>	<hr style="border: 0.5px solid black;"/> <pre> {1} null?(bl') {2} transformers_ok?(pm' 'states, singleton(expr_2_super(λ (s: Linear_memory[Physical_memory, pm_phy]): apply_side_effects(split(min_page, a', bl'), lin- ear_read_side_effect_in_page) (s)))) </pre>
---	--

Hiding formulas: -2,
 Expanding the definition of transformers_ok?,
 Repeatedly Skolemizing and flattening,
 Expanding the definition of singleton,
 Replacing using formula -7,
 Hiding formulas: -7,
 Case splitting on $0 \leq a!1'offset$ AND $a!1'offset < max_linear_offset$,
 we get 2 subgoals:
 linear_plain_transformers_ok.3.1.1.2.1:

<pre> {-1} 0 ≤ a' 'offset ∧ a' 'offset < max_linear_offset {-2} Mem?(type_of(a')) {-3} is_linear_plain_memory?(pm') {-4} plain_memory?(pm_phy) {-5} pm' 'states = pm_phy 'states {-6} (address_block(a', length(bl')) ⊆ (pm' 'ro_addr ∪ pm' 'rw_addr)) {-7} pm' 'states(s') </pre>	<hr style="border: 0.5px solid black;"/> <pre> {1} null?(bl') {2} OK?(expr_2_super(λ (s: Linear_memory[Physical_memory, pm_phy]): apply_side_effects(split(min_page, a', bl'), linear_read_side_effect_in_page) (s)) (s')) </pre>
---	---

Using lemma address_block_split_type,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 Rewriting using ok_expr_2_super, matching in *,

C Proof scripts

Using lemma `apply_side_effects_ok`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 2 subgoals:

`linear_plain_transformers_ok.3.1.1.2.1.1:`

{-1}	<code>is_linear_plain_memory?(pm')</code>
{-2}	<code>pm' states(s')</code>
{-3}	<code>(address_block(a', length(bl')) ⊆ (pm'ro_addr ∪ pm'rw_addr))</code>
{-4}	<code>every(λ (t: [Address, list[Byte]]):</code> $\text{Mem?}(t'1'\text{type_of}) \wedge 0 \leq t'1'\text{offset} \wedge t'1'\text{offset} < \text{max_linear_offset}$ <code>(split(min_page, a', bl'))</code>
{-5}	<code>0 ≤ a'offset</code>
{-6}	<code>a'offset < max_linear_offset</code>
{-7}	<code>Mem?(type_of(a'))</code>
{-8}	<code>plain_memory?(pm_phy)</code>
{-9}	<code>pm' states = pm_phy states</code>
{1}	<code>every(λ (e: [Memory_Address_4G, list[Byte]]):</code> $\text{transformer_invariant?}(pm' \text{ states},$ $\text{singleton}(\text{expr_2_super}(\text{linear_read_side_effect_in_page}(e))))$ <code>(split(min_page, a', bl'))</code>
{2}	<code>OK?(apply_side_effects(split(min_page, a', bl'), linear_read_side_effect_in_page)(s'))</code>
{3}	<code>null?(bl')</code>

Using lemma `split_linear_read_side_effects_states`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `linear_plain_transformers_ok.3.1.1.2.1.1`.

`linear_plain_transformers_ok.3.1.1.2.1.2:`

{-1}	<code>is_linear_plain_memory?(pm')</code>
{-2}	<code>pm' states(s')</code>
{-3}	<code>(address_block(a', length(bl')) ⊆ (pm'ro_addr ∪ pm'rw_addr))</code>
{-4}	<code>every(λ (t: [Address, list[Byte]]):</code> $\text{Mem?}(t'1'\text{type_of}) \wedge 0 \leq t'1'\text{offset} \wedge t'1'\text{offset} < \text{max_linear_offset}$ <code>(split(min_page, a', bl'))</code>
{-5}	<code>0 ≤ a'offset</code>
{-6}	<code>a'offset < max_linear_offset</code>
{-7}	<code>Mem?(type_of(a'))</code>
{-8}	<code>plain_memory?(pm_phy)</code>
{-9}	<code>pm' states = pm_phy states</code>
{1}	$\forall (s_1: (pm' \text{ states})):$ <code>every(λ (e: [Memory_Address_4G, list[Byte]]):</code> $\text{OK?}(\text{linear_read_side_effect_in_page}(e)(s_1))$ <code>(split(min_page, a', bl'))</code>
{2}	<code>OK?(apply_side_effects(split(min_page, a', bl'), linear_read_side_effect_in_page)(s'))</code>
{3}	<code>null?(bl')</code>

Repeatedly Skolemizing and flattening,

Using lemma `split_linear_read_side_effects_ok`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `linear_plain_transformers_ok.3.1.1.2.1.2`.

linear_plain_transformers_ok.3.1.1.2.2:

<pre> {-1} Mem?(type_of(a')) {-2} is_linear_plain_memory?(pm') {-3} plain_memory?(pm_phy) {-4} pm'`states = pm_phy`states {-5} (address_block(a', length(bl')) ⊆ (pm'`ro_addr ∪ pm'`rw_addr)) {-6} pm'`states(s') </pre>	<hr style="border: 0.5px solid black;"/> <pre> {1} 0 ≤ a'`offset ∧ a'`offset < max_linear_offset {2} null?(bl') {3} OK?(expr_2_super(λ (s : Linear_memory[Physical_memory, pm_phy]): apply_side_effects(split(min_page, a', bl'), linear_read_side_effect_in_page) (s)) (s')) </pre>
--	---

Hiding formulas: 3,

Expanding the definition of length,

Simplifying, rewriting, and recording with decision procedures,

Expanding the definition of subset?,

Instantiating quantified variables,

Using lemma pm_memory_addr,

Expanding the definition of address_block,

Expanding the definition of in_memory,

Expanding the definition of reg_size,

Expanding the definition of < ,

Expanding the definition of max_linear,

Expanding the definition of reg_base,

Expanding the definition of min_linear,

Expanding the definition of Mem,

Expanding the definition of <= ,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of linear_plain_transformers_ok.3.1.1.2.2.

linear_plain_transformers_ok.3.1.2:

<pre> {-1} null?(bl') ∨ (reg_base(min_linear)(type_of(a')) ≤ a' ∧ a' + length(bl') ≤ reg_size(max_linear)(type_of(a'))) {-2} is_linear_plain_memory?(pm') {-3} plain_memory?(pm_phy) {-4} pm' 'states = pm_phy 'states {-5} (address_block(a', length(bl')) ⊆ (pm' 'ro_addr ∪ pm' 'rw_addr)) </pre>	<pre> {1} Mem?(type_of(a')) {2} transformers_ok?(pm' 'states, singleton(expr_2_super(λ (s: Linear_memory[Physical_memory, pm_phy]): IF null?(bl') ∨ (reg_base(min_linear)(type_of(a')) ≤ a' ∧ a' + length(bl') ≤ reg_size(max_linear)(type_of(a'))) THEN IF Mem?(type_of(a')) THEN ap- (min a' bl lin- (s) ELSE mem- (a', bl', cp')(s) ENDIF ELSE Fatal ENDIF))) </pre>
--	--

Replacing using formula -1,
Simplifying, rewriting, and recording with decision procedures,
Using lemma plain_memory_transformers_ok_read_side_effects_ro_rw,
Using lemma transformers_ok_mono_transformers,
Hiding formulas: -3,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Hiding formulas: (-2 2),
Expanding the definition of memory_read_side_effect_super_transformers,
Expanding the definition of expr_2_super,
Expanding the definition of subset?,
Repeatedly Skolemizing and flattening,
Simplifying, rewriting, and recording with decision procedures,
Instantiating quantified variables,
Replacing using formula -1,
Repeatedly Skolemizing and flattening,
Instantiating the top quantifier in -6 with the terms: x'' ,
Expanding the definition of address_block,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Using lemma pm_memory_addr,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `linear_plain_transformers_ok.3.1.2`.

`linear_plain_transformers_ok.3.2`:

<pre> {-1} is_linear_plain_memory?(pm') {-2} plain_memory?(pm_phy) {-3} pm' 'states = pm_phy 'states {-4} (address_block(a', length(bl')) ⊆ (pm' 'ro_addr ∪ pm' 'rw_addr)) </pre>	<pre> {1} null?(bl') ∨ (reg_base(min_linear)(type_of(a')) ≤ a' ∧ a' + length(bl') ≤ reg_size(max_linear)(type_of(a'))) {2} transformers_ok?(pm' 'states, singleton(expr_2_super(λ (s: Linear_memory[Physical_memory, pm_phy]): IF null?(bl') ∨ (reg_base(min_linear)(type_of(a')) ≤ a' ∧ a' + length(bl') ≤ reg_size(max_linear)(type_of(a'))) THEN IF Mem?(type_of(a')) THEN ap- (min_page, a', bl'), lin- (s) ELSE mem- (a', bl', cp')(s) ENDIF ELSE Fatal ENDIF))) </pre>
---	--

Hiding formulas: 2,

Expanding the definition of length,

Applying disjunctive simplification to flatten sequent,

Using lemma `pm_memory_addr`,

Using lemma `pm_memory_addr`,

we get 2 subgoals:

C Proof scripts

`linear_plain_transformers_ok.3.2.1:`

{-1}	$\text{is_linear_plain_memory?}(pm') \wedge$ $\text{union}(pm'\text{'ro_addr}, pm'\text{'rw_addr})(a' + \text{length}(\text{cdr}(bl')))$ \supset $\text{in_memory}(\text{min_linear}, \text{max_linear})(a' + \text{length}(\text{cdr}(bl'))) \wedge$ $\text{Mem?}(\text{type_of}(a' + \text{length}(\text{cdr}(bl'))))$
{-2}	$\text{is_linear_plain_memory?}(pm') \wedge \text{union}(pm'\text{'ro_addr}, pm'\text{'rw_addr})(a') \supset$ $\text{in_memory}(\text{min_linear}, \text{max_linear})(a') \wedge \text{Mem?}(\text{type_of}(a'))$
{-3}	$\text{is_linear_plain_memory?}(pm')$
{-4}	$\text{plain_memory?}(pm_phy)$
{-5}	$pm'\text{'states} = pm_phy'\text{'states}$
{-6}	$(\text{address_block}(a', \text{CASES } bl' \text{ OF null : 0, cons}(x, y) : \text{length}(y) + 1 \text{ ENDCASES}) \subseteq (pm'\text{'ro_addr}$
{1}	$\text{null?}(bl')$
{2}	$(\text{reg_base}(\text{min_linear})(\text{type_of}(a')) \leq a' \wedge$ $a' + \text{CASES } bl' \text{ OF null : 0, cons}(x, y) : \text{length}(y) + 1 \text{ ENDCASES} \leq$ $\text{reg_size}(\text{max_linear})(\text{type_of}(a')))$

Simplifying, rewriting, and recording with decision procedures,

Hiding formulas: (-4 -3),

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `linear_plain_transformers_ok.3.2.1`.

`linear_plain_transformers_ok.3.2.2:`

{-1}	$\text{is_linear_plain_memory?}(pm') \wedge \text{union}(pm'\text{'ro_addr}, pm'\text{'rw_addr})(a') \supset$ $\text{in_memory}(\text{min_linear}, \text{max_linear})(a') \wedge \text{Mem?}(\text{type_of}(a'))$
{-2}	$\text{is_linear_plain_memory?}(pm')$
{-3}	$\text{plain_memory?}(pm_phy)$
{-4}	$pm'\text{'states} = pm_phy'\text{'states}$
{-5}	$(\text{address_block}(a', \text{CASES } bl' \text{ OF null : 0, cons}(x, y) : \text{length}(y) + 1 \text{ ENDCASES}) \subseteq (pm'\text{'ro_addr}$
{1}	$\text{cons?}[\text{Byte}](bl')$
{2}	$\text{null?}(bl')$
{3}	$(\text{reg_base}(\text{min_linear})(\text{type_of}(a')) \leq a' \wedge$ $a' + \text{CASES } bl' \text{ OF null : 0, cons}(x, y) : \text{length}(y) + 1 \text{ ENDCASES} \leq$ $\text{reg_size}(\text{max_linear})(\text{type_of}(a')))$

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `linear_plain_transformers_ok.3.2.2`.

linear_plain_transformers_ok.3.3:

<pre> {-1} is_linear_plain_memory?(pm') {-2} plain_memory?(pm_phy) {-3} pm'`states = pm_phy`states {-4} (address_block(a', length(bl')) ⊆ (pm'`ro_addr ∪ pm'`rw_addr)) </pre>	<pre> {1} ¬ null?(bl') ∧ reg_base(min_linear)(type_of(a')) ≤ a' ⊃ Mem?(max_linear`type_of) {2} transformers_ok?(pm'`states, singleton(expr_2_super(λ (s: Linear_memory[Physical_memory, pm_phy]): IF null?(bl') ∨ (reg_base(min_linear)(type_of(a')) ≤ a' ∧ a' + length(bl') ≤ reg_size(max_linear)(type_of(a'))) THEN IF Mem?(type_of(a')) THEN ap- (min_page, a', bl'), lin- (s) ELSE mem- (a', bl', cp')(s) ENDIF ELSE Fatal ENDIF))) </pre>
---	--

Expanding the definition of max_linear,

Expanding the definition of Mem,

which is trivially true.

This completes the proof of linear_plain_transformers_ok.3.3.

linear_plain_transformers_ok.3.4:

<pre> {-1} is_linear_plain_memory?(pm') {-2} plain_memory?(pm_phy) {-3} pm'`states = pm_phy'`states {-4} (address_block(a', length(bl')) ⊆ (pm'`ro_addr ∪ pm'`rw_addr)) </pre>	<pre> {1} ¬ null?(bl') ⊃ Mem?(min_linear'`type_of) {2} transformers_ok?(pm'`states, singleton(expr_2_super(λ (s: Linear_memory[Physical_memory, pm_phy]): IF null?(bl') ∨ (reg_base(min_linear)(type_of(a')) ≤ a' ∧ a' + length(bl') ≤ reg_size(max_linear)(type_of(a'))) THEN IF Mem?(type_of(a')) THEN ap- (min a' bl lin- (s) ELSE mem- (a', bl', cp')(s) ENDIF ELSE Fatal ENDIF))) ply_side_effects(split ear_read_side_effect_in_page) ory_read_side_effect(pm_phy'`mem) </pre>
--	--

Expanding the definition of min_linear,

Expanding the definition of Mem,

which is trivially true.

This completes the proof of linear_plain_transformers_ok.3.4.

linear_plain_transformers_ok.4:

<pre> {-1} is_linear_plain_memory?(pm') {-2} plain_memory?(pm_phy) {-3} pm'`states = pm_phy'`states {-4} memory_write_side_effect_super_transformers(pm'`mem, pm'`rw_addr)(q') </pre>	<pre> {1} transformers_ok?(pm'`states, singleton(q')) </pre>
---	--

Expanding the definition of memory_write_side_effect_super_transformers,

Repeatedly Skolemizing and flattening,

Replacing using formula -6,

Hiding formulas: (-1 -6),

Rewriting using pm_write_side_effect_linear, matching in *,

Expanding the definition of linear_write_side_effect,

Case splitting on null?(bl!1) OR (reg_base(min_linear)(type_of(a!1)) ≤ a!1 AND a!1 + length(bl!1) ≤ reg_size(max_linear)(type_of(a!1))),

we get 4 subgoals:

linear_plain_transformers_ok.4.1:

<pre> {-1} null?(bl') ∨ (reg_base(min_linear)(type_of(a')) ≤ a' ∧ a' + length(bl') ≤ reg_size(max_linear)(type_of(a'))) {-2} is_linear_plain_memory?(pm') {-3} plain_memory?(pm_phy) {-4} pm'.'states = pm_phy.'states {-5} (address_block(a', length(bl')) ⊆ pm'.'rw_addr) </pre>	<pre> {1} transformers_ok?(pm'.'states, singleton(expr_2_super(λ (s: Linear_memory[Physical_memory, pm_phy]): IF null?(bl') ∨ (reg_base(min_linear)(type_of(a')) ≤ a' ∧ a' + length(bl') ≤ reg_size(max_linear)(type_of(a'))) THEN IF Mem?(type_of(a')) THEN ap- (min_page, a', bl'), lin- (s) ELSE mem- (a', bl', cp')(s) ENDIF ELSE Fatal ENDIF))) </pre>
---	--

Case splitting on Mem?(type_of(a!1)),

we get 2 subgoals:

linear_plain_transformers_ok.4.1.1.1:

{-1}	Mem?(type_of(a'))	
{-2}	null?(bl') \vee (reg_base(min_linear)(type_of(a')) \leq a' \wedge a' + length(bl') \leq reg_size(max_linear)(type_of(a')))	
{-3}	is_linear_plain_memory?(pm')	
{-4}	plain_memory?(pm_phy)	
{-5}	pm' 'states = pm_phy 'states	
{-6}	(address_block(a', length(bl')) \subseteq pm' 'rw_addr)	
{1}	transformers_ok?(pm' 'states, singleton(expr_2_super(λ (s: Linear_memory[Physical_memory, pm_phy]): IF null?(bl') \vee (reg_base(min_linear)(type_of(a')) \leq a' \wedge a' + length(bl') \leq reg_size(max_linear)(type_of(a'))) THEN IF Mem?(type_of(a')) THEN ap-	(min a' bl lin-
	ply_side_effects(split	(s)
	ear_write_side_effect_in_page)	(s)
	ory_write_side_effect(pm_phy 'mem)	ELSE mem-
		(a', bl', cp')(s)
		ENDIF
		ELSE Fatal
		ENDIF)))

Replacing using formula -2,
Simplifying, rewriting, and recording with decision procedures,
Case splitting on null?(bl'),
we get 2 subgoals:

linear_plain_transformers_ok.4.1.1.1.1:

{-1}	null?(bl')	
{-2}	Mem?(type_of(a'))	
{-3}	null?(bl') \vee (reg_base(min_linear)(type_of(a')) \leq a' \wedge a' + length(bl') \leq reg_size(max_linear)(type_of(a')))	
{-4}	is_linear_plain_memory?(pm')	
{-5}	plain_memory?(pm_phy)	
{-6}	pm' 'states = pm_phy 'states	
{-7}	(address_block(a', length(bl')) \subseteq pm' 'rw_addr)	
{1}	transformers_ok?(pm' 'states, singleton(expr_2_super(λ (s: Linear_memory[Physical_memory, pm_phy]): apply_side_effects(split(min_page, a', bl'), lin-	(s))))
	ear_write_side_effect_in_page)	(s))))

Hiding formulas: -3,
 Using lemma split_null,
 Expanding the definition of apply_side_effects,
 Expanding the definition of reduce,
 Expanding the definition of ok_result,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 Keeping (1) and hiding *,
 Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of linear_plain_transformers_ok.4.1.1.1.
 linear_plain_transformers_ok.4.1.1.2:

<pre> {-1} Mem?(type_of(a')) {-2} null?(bl') ∨ (reg_base(min_linear)(type_of(a')) ≤ a' ∧ a' + length(bl') ≤ reg_size(max_linear)(type_of(a'))) {-3} is_linear_plain_memory?(pm') {-4} plain_memory?(pm_phy) {-5} pm' 'states = pm_phy 'states {-6} (address_block(a', length(bl')) ⊆ pm' 'rw_addr) </pre>	<hr style="border: 0.5px solid black;"/> <pre> {1} null?(bl') {2} transformers_ok?(pm' 'states, singleton(expr_2_super(λ (s: Linear_memory[Physical_memory, pm_phy]): apply_side_effects(split(min_page, a', bl'), lin- ear_write_side_effect_in_page) (s)))) </pre>
--	---

Hiding formulas: -2,
 Expanding the definition of transformers_ok?,
 Repeatedly Skolemizing and flattening,
 Expanding the definition of singleton,
 Replacing using formula -7,
 Hiding formulas: -7,
 Case splitting on $0 \leq a!1'offset$ AND $a!1'offset < max_linear_offset$,
 we get 2 subgoals:
 linear_plain_transformers_ok.4.1.1.2.1:

<pre> {-1} 0 ≤ a' 'offset ∧ a' 'offset < max_linear_offset {-2} Mem?(type_of(a')) {-3} is_linear_plain_memory?(pm') {-4} plain_memory?(pm_phy) {-5} pm' 'states = pm_phy 'states {-6} (address_block(a', length(bl')) ⊆ pm' 'rw_addr) {-7} pm' 'states(s') </pre>	<hr style="border: 0.5px solid black;"/> <pre> {1} null?(bl') {2} OK?(expr_2_super(λ (s: Linear_memory[Physical_memory, pm_phy]): apply_side_effects(split(min_page, a', bl'), linear_write_side_effect_in_page) (s)) (s')) </pre>
--	--

Using lemma address_block_split_type,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 we get 2 subgoals:

`linear_plain_transformers_ok.4.1.1.2.1.1:`

{-1}	<code>is_linear_plain_memory?(pm')</code>
{-2}	<code>every(λ (t: [Address, list[Byte]]):</code> $\text{Mem?}(t'1'\text{type_of}) \wedge 0 \leq t'1'\text{offset} \wedge t'1'\text{offset} < \text{max_linear_offset}$ $(\text{split}(\text{min_page}, a', \text{bl}'))$
{-3}	$0 \leq a'\text{offset}$
{-4}	$a'\text{offset} < \text{max_linear_offset}$
{-5}	<code>Mem?(type_of(a'))</code>
{-6}	<code>plain_memory?(pm_phy)</code>
{-7}	<code>pm' states = pm_phy states</code>
{-8}	$(\text{address_block}(a', \text{length}(\text{bl}')) \subseteq \text{pm}'\text{rw_addr})$
{-9}	<code>pm' states(s')</code>
{1}	<code>null?(bl')</code>
{2}	<code>OK?(expr_2_super(λ (s: Linear_memory[Physical_memory, pm_phy]):</code> $\text{apply_side_effects}(\text{split}(\text{min_page}, a', \text{bl}'),$ $\text{linear_write_side_effect_in_page}$ $(s))$
	(s')

Rewriting using `ok_expr_2_super`, matching in `*`,

Using lemma `apply_side_effects_ok`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 2 subgoals:

`linear_plain_transformers_ok.4.1.1.2.1.1.1:`

{-1}	<code>is_linear_plain_memory?(pm')</code>
{-2}	<code>pm' states(s')</code>
{-3}	<code>every(λ (t: [Address, list[Byte]]):</code> $\text{Mem?}(t'1'\text{type_of}) \wedge 0 \leq t'1'\text{offset} \wedge t'1'\text{offset} < \text{max_linear_offset}$ $(\text{split}(\text{min_page}, a', \text{bl}'))$
{-4}	$0 \leq a'\text{offset}$
{-5}	$a'\text{offset} < \text{max_linear_offset}$
{-6}	<code>Mem?(type_of(a'))</code>
{-7}	<code>plain_memory?(pm_phy)</code>
{-8}	<code>pm' states = pm_phy states</code>
{-9}	$(\text{address_block}(a', \text{length}(\text{bl}')) \subseteq \text{pm}'\text{rw_addr})$
{1}	<code>every(λ (e: [Memory_Address_4G, list[Byte]]):</code> $\text{transformer_invariant?}(\text{pm}'\text{states},$ $\text{singleton}(\text{expr_2_super}(\text{linear_write_side_effect_in_page}(e))))$ $(\text{split}(\text{min_page}, a', \text{bl}'))$
{2}	<code>OK?(apply_side_effects(split(min_page, a', bl'), linear_write_side_effect_in_page)(s'))</code>
{3}	<code>null?(bl')</code>

Using lemma `split_linear_write_side_effects_states`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `linear_plain_transformers_ok.4.1.1.2.1.1.1`.

linear_plain_transformers_ok.4.1.1.2.1.1.2:

<pre> {-1} is_linear_plain_memory?(pm') {-2} pm'`states(s') {-3} every(λ (t: [Address, list[Byte]]): Mem?(t`1`type_of) ∧ 0 ≤ t`1`offset ∧ t`1`offset < max_linear_offset) (split(min_page, a', bl')) {-4} 0 ≤ a'`offset {-5} a'`offset < max_linear_offset {-6} Mem?(type_of(a')) {-7} plain_memory?(pm_phy) {-8} pm'`states = pm_phy`states {-9} (address_block(a', length(bl')) ⊆ pm'`rw_addr) </pre>	<pre> {1} ∃ (s1: (pm'`states)): every(λ (e: [Memory_Address_4G, list[Byte]]): OK?(linear_write_side_effect_in_page(e)(s1)) (split(min_page, a', bl'))) {2} OK?(apply_side_effects(split(min_page, a', bl'), linear_write_side_effect_in_page)(s')) {3} null?(bl') </pre>
---	--

Repeatedly Skolemizing and flattening,

Using lemma split_linear_write_side_effects_ok,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of linear_plain_transformers_ok.4.1.1.2.1.1.2.

linear_plain_transformers_ok.4.1.1.2.1.2:

<pre> {-1} is_linear_plain_memory?(pm') {-2} 0 ≤ a'`offset {-3} a'`offset < max_linear_offset {-4} Mem?(type_of(a')) {-5} plain_memory?(pm_phy) {-6} pm'`states = pm_phy`states {-7} (address_block(a', length(bl')) ⊆ pm'`rw_addr) {-8} pm'`states(s') </pre>	<pre> {1} (address_block(a', length(bl')) ⊆ (pm'`ro_addr ∪ pm'`rw_addr)) {2} null?(bl') {3} OK?(expr_2_super(λ (s: Linear_memory[Physical_memory, pm_phy]): apply_side_effects(split(min_page, a', bl'), linear_write_side_effect_in_page) (s)) (s')) </pre>
---	--

Hiding formulas: 3,

Keeping (-6 1) and hiding *,

Rewriting using subset_transitive, matching in * where a gets address_block(a!1, length(bl!1)), b gets pm!1`rw_addr,

Rewriting using union_subset3, matching in *,

This completes the proof of linear_plain_transformers_ok.4.1.1.2.1.2.

C Proof scripts

`linear_plain_transformers_ok.4.1.1.2.2:`

<pre> {-1} Mem?(type_of(a')) {-2} is_linear_plain_memory?(pm') {-3} plain_memory?(pm_phy) {-4} pm'`states = pm_phy`states {-5} (address_block(a', length(bl')) ⊆ pm'`rw_addr) {-6} pm'`states(s') </pre>	<hr style="border: 0.5px solid black;"/> <pre> {1} 0 ≤ a'`offset ∧ a'`offset < max_linear_offset {2} null?(bl') {3} OK?(expr_2_super(λ (s: Linear_memory[Physical_memory, pm_phy]): apply_side_effects(split(min_page, a', bl'), linear_write_side_effect_in_page) (s)) (s')) </pre>
--	---

Hiding formulas: 3,

Expanding the definition of length,

Simplifying, rewriting, and recording with decision procedures,

Expanding the definition of subset?,

Instantiating quantified variables,

Using lemma `pm_memory_addr`,

Expanding the definition of `address_block`,

Expanding the definition of `in_memory`,

Expanding the definition of `reg_size`,

Expanding the definition of `<`,

Expanding the definition of `max_linear`,

Expanding the definition of `reg_base`,

Expanding the definition of `min_linear`,

Expanding the definition of `Mem`,

Expanding the definition of `<=`,

Rewriting using `union_right`, matching in `*`,

This completes the proof of `linear_plain_transformers_ok.4.1.1.2.2`.

linear_plain_transformers_ok.4.1.2:

<pre> {-1} null?(bl') ∨ (reg_base(min_linear)(type_of(a')) ≤ a' ∧ a' + length(bl') ≤ reg_size(max_linear)(type_of(a'))) {-2} is_linear_plain_memory?(pm') {-3} plain_memory?(pm_phy) {-4} pm' 'states = pm_phy 'states {-5} (address_block(a', length(bl')) ⊆ pm' 'rw_addr) </pre>	<pre> {1} Mem?(type_of(a')) {2} transformers_ok?(pm' 'states, singleton(expr_2_super(λ (s: Linear_memory[Physical_memory, pm_phy]): IF null?(bl') ∨ (reg_base(min_linear)(type_of(a')) ≤ a' ∧ a' + length(bl') ≤ reg_size(max_linear)(type_of(a'))) THEN IF Mem?(type_of(a')) THEN ap- ply_side_effects(split (min_page, a', bl'), lin- ear_write_side_effect_in_page) ory_write_side_effect(pm_phy ' mem) ELSE mem- (a', bl', cp')(s) ENDIF ELSE Fatal ENDIF))) </pre>
---	--

Replacing using formula -1,
Simplifying, rewriting, and recording with decision procedures,
Using lemma plain_memory_transformers_ok_write_side_effects_ro_rw,
Using lemma transformers_ok_mono_transformers,
Hiding formulas: -3,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Hiding formulas: (-2 2),
Expanding the definition of memory_write_side_effect_super_transformers,
Expanding the definition of expr_2_super,
Expanding the definition of subset?,
Repeatedly Skolemizing and flattening,
Simplifying, rewriting, and recording with decision procedures,
Instantiating quantified variables,
Replacing using formula -1,
Repeatedly Skolemizing and flattening,
Instantiating the top quantifier in -6 with the terms: x'' ,
Expanding the definition of address_block,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Using lemma pm_memory_addr,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

C Proof scripts

Rewriting using `union_right`, matching in `*`,

This completes the proof of `linear_plain_transformers_ok.4.1.2`.

`linear_plain_transformers_ok.4.2`:

<pre> {-1} is_linear_plain_memory?(pm') {-2} plain_memory?(pm_phy) {-3} pm'`states = pm_phy`states {-4} (address_block(a', length(bl')) ⊆ pm'`rw_addr) </pre>	<pre> {1} null?(bl') ∨ (reg_base(min_linear)(type_of(a')) ≤ a' ∧ a' + length(bl') ≤ reg_size(max_linear)(type_of(a'))) {2} transformers_ok?(pm'`states, singleton(expr_2_super(λ (s: Linear_memory [Physical_memory, pm_phy]): IF null?(bl') ∨ (reg_base(min_linear)(type_of(a')) ≤ a' ∧ a' + length(bl') ≤ reg_size(max_linear)(type_of(a'))) THEN IF Mem?(type_of(a')) THEN ap- ply_side_effects(split ear_write_side_effect_in_page) ory_write_side_effect(pm_phy`mem) ELSE mem- (a', bl', cp')(s) ENDIF ELSE Fatal ENDIF))) </pre>
---	---

Hiding formulas: 2,

Expanding the definition of `length`,

Using lemma `pm_memory_addr`,

Using lemma `pm_memory_addr`,

we get 2 subgoals:

linear_plain_transformers_ok.4.2.1:

{-1}	is_linear_plain_memory?(pm') \wedge union(pm'ro_addr, pm'rw_addr)(a' + length(cdr(bl'))) \supset in_memory(min_linear, max_linear)(a' + length(cdr(bl'))) \wedge Mem?(type_of(a' + length(cdr(bl'))))
{-2}	is_linear_plain_memory?(pm') \wedge union(pm'ro_addr, pm'rw_addr)(a') \supset in_memory(min_linear, max_linear)(a') \wedge Mem?(type_of(a'))
{-3}	is_linear_plain_memory?(pm')
{-4}	plain_memory?(pm_phy)
{-5}	pm'states = pm_phy'states
{-6}	(address_block(a', CASES bl' OF null : 0, cons(x, y) : length(y) + 1 ENDCASES) \subseteq pm'rw_addr)
{1}	null?(bl') \vee (reg_base(min_linear)(type_of(a')) \leq a' \wedge a' + CASES bl' OF null : 0, cons(x, y) : length(y) + 1 ENDCASES \leq reg_size(max_linear)(type_of(a')))

Simplifying, rewriting, and recording with decision procedures,

Hiding formulas: (-4 -3),

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of linear_plain_transformers_ok.4.2.1.

linear_plain_transformers_ok.4.2.2:

{-1}	is_linear_plain_memory?(pm') \wedge union(pm'ro_addr, pm'rw_addr)(a') \supset in_memory(min_linear, max_linear)(a') \wedge Mem?(type_of(a'))
{-2}	is_linear_plain_memory?(pm')
{-3}	plain_memory?(pm_phy)
{-4}	pm'states = pm_phy'states
{-5}	(address_block(a', CASES bl' OF null : 0, cons(x, y) : length(y) + 1 ENDCASES) \subseteq pm'rw_addr)
{1}	cons?[Byte](bl')
{2}	null?(bl') \vee (reg_base(min_linear)(type_of(a')) \leq a' \wedge a' + CASES bl' OF null : 0, cons(x, y) : length(y) + 1 ENDCASES \leq reg_size(max_linear)(type_of(a')))

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of linear_plain_transformers_ok.4.2.2.

C Proof scripts

linear_plain_transformers_ok.4.3:

<pre> {-1} is_linear_plain_memory?(pm') {-2} plain_memory?(pm_phy) {-3} pm' 'states = pm_phy 'states {-4} (address_block(a', length(bl')) ⊆ pm' 'rw_addr) </pre>	<hr style="border: 0.5px solid black;"/> <pre> {1} ¬ null?(bl') ∧ reg_base(min_linear)(type_of(a')) ≤ a' ⊃ Mem?(max_linear 'type_of) {2} transformers_ok?(pm' 'states, singleton(expr_2_super(λ (s: Linear_memory [Physical_memory, pm_phy]): IF null?(bl') ∨ (reg_base(min_linear)(type_of(a')) ≤ a' ∧ a' + length(bl') ≤ reg_size(max_linear)(type_of(a')) THEN IF Mem?(type_of(a')) THEN ap- (min a' bl lin- (s) ELSE mem- (a', bl', cp')(s) ENDIF ELSE Fatal ENDIF))) </pre>
--	---

Expanding the definition of max_linear,

Expanding the definition of Mem,

which is trivially true.

This completes the proof of linear_plain_transformers_ok.4.3.

linear_plain_transformers_ok.4.4:

<pre> {-1} is_linear_plain_memory?(pm') {-2} plain_memory?(pm_phy) {-3} pm'.'states = pm_phy.'states {-4} (address_block(a', length(bl')) ⊆ pm'.'rw_addr) </pre>	<pre> {1} ¬ null?(bl') ⊃ Mem?(min_linear.'type_of) {2} transformers_ok?(pm'.'states, singleton(expr_2_super(λ (s: Linear_memory[Physical_memory, pm_phy]): IF null?(bl') ∨ (reg_base(min_linear)(type_of(a')) ≤ a' ^ a' + length(bl') ≤ reg_size(max_linear)(type_of(a'))) THEN IF Mem?(type_of(a')) THEN ap- (min_page, a', bl'), lin- (s) ELSE mem- (a', bl', cp')(s) ENDIF ELSE Fatal ENDIF))) </pre>
--	--

Expanding the definition of min_linear,
 Expanding the definition of Mem,
 which is trivially true.

This completes the proof of linear_plain_transformers_ok.4.4.
 Q.E.D.

C.116.2 Linear_Memory_Blessing_Properties.linear_plain_unchanged_memory_invariant

Terse proof for linear_plain_unchanged_memory_invariant.

linear_plain_unchanged_memory_invariant:

<pre> {1} ∀ (pm: Plain_Memory[Physical_memory]): is_linear_plain_memory?(pm) ⊃ unchanged_memory_invariant?(pm.'mem, pm.'states, ((pm.'other_actions ∪ memory_read_transformers(pm.'mem, (pm.'ro_addr (pm.'ro_addr ∪ pm.'rw_addr)) </pre>
--

Repeatedly Skolemizing and flattening,
 Installing automatic rewrites from: has_next_state singleton subset_reflexive pm_write_linear lin-
 ear_write Mem in_memory Mem reg_size union_left max_linear <
 Using lemma linear_unchanged_invariant,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

C Proof scripts

we get 2 subgoals:

`linear_plain_unchanged_memory_invariant.1:`

<pre>{-1} is_linear_plain_memory?(pm') {-2} unchanged_memory_invariant?(pm' mem, pm' states, ((pm' other_actions ∪ memory_read_transformers(pm' mem, (pm extend[Address, Memory_Address_4G, bool, FALSE] (restrict[Address, Mem- ory_Address_4G, boolean] ((pm' ro_addr ∪ pm' rw_addr))))))</pre>	<pre>(pm ory_Address_4G, boolean] ((pm' ro_addr ∪ pm' rw_addr))))))</pre>
<pre>{1} unchanged_memory_invariant?(pm' mem, pm' states, ((pm' other_actions ∪ memory_read_transformers(pm' mem, (pm (pm' ro_addr ∪ pm' rw_addr)))</pre>	<pre>(pm (pm' ro_addr ∪ pm' rw_addr)))</pre>

Case splitting on `extend[Address, Memory_Address_4G, bool, FALSE] (restrict[Address, Memory_Address_4G, boolean] (union(pm!1'ro_addr, pm!1'rw_addr))) = union(pm!1'ro_addr, pm!1'rw_addr)`,

we get 2 subgoals:

`linear_plain_unchanged_memory_invariant.1.1:`

<pre>{-1} extend[Address, Memory_Address_4G, bool, FALSE] (restrict[Address, Memory_Address_4G, boolean]((pm' ro_addr ∪ pm' rw_addr))) = (pm' ro_addr ∪ pm' rw_addr) {-2} is_linear_plain_memory?(pm') {-3} unchanged_memory_invariant?(pm' mem, pm' states, ((pm' other_actions ∪ memory_read_transformers(pm' mem, (pm extend[Address, Memory_Address_4G, bool, FALSE] (restrict[Address, Mem- ory_Address_4G, boolean] ((pm' ro_addr ∪ pm' rw_addr))))))</pre>	<pre>(pm ory_Address_4G, boolean] ((pm' ro_addr ∪ pm' rw_addr))))))</pre>
<pre>{1} unchanged_memory_invariant?(pm' mem, pm' states, ((pm' other_actions ∪ memory_read_transformers(pm' mem, (pm (pm' ro_addr ∪ pm' rw_addr)))</pre>	<pre>(pm (pm' ro_addr ∪ pm' rw_addr)))</pre>

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `linear_plain_unchanged_memory_invariant.1.1`.

`linear_plain_unchanged_memory_invariant.1.2:`

<pre>{-1} is_linear_plain_memory?(pm') {-2} unchanged_memory_invariant?(pm' mem, pm' states, ((pm' other_actions ∪ memory_read_transformers(pm' mem, (pm extend[Address, Memory_Address_4G, bool, FALSE] (restrict[Address, Mem- ory_Address_4G, boolean] ((pm' ro_addr ∪ pm' rw_addr))))))</pre>	<pre>(pm ory_Address_4G, boolean] ((pm' ro_addr ∪ pm' rw_addr))))))</pre>
<pre>{1} extend[Address, Memory_Address_4G, bool, FALSE] (restrict[Address, Memory_Address_4G, boolean]((pm' ro_addr ∪ pm' rw_addr))) = (pm' ro_addr ∪ pm' rw_addr) {2} unchanged_memory_invariant?(pm' mem, pm' states, ((pm' other_actions ∪ memory_read_transformers(pm' mem, (pm (pm' ro_addr ∪ pm' rw_addr)))</pre>	<pre>(pm (pm' ro_addr ∪ pm' rw_addr)))</pre>

Hiding formulas: (-2 2),

Expanding the definition of `extend`,

Expanding the definition of `restrict`,

Applying decompose-equality,

Using lemma pm_memory_addr,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Keeping (-3 1) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of linear_plain_unchanged_memory_invariant.1.2.

linear_plain_unchanged_memory_invariant.2:

{-1}	is_linear_plain_memory?(pm')	
{1}	$\forall (s: (pm' \text{'states})): \text{unchanged_memory_invariant?}(pm_phy' \text{'mem}, pm_phy' \text{'states},$ $((pm' \text{'other_actions} \cup \text{memory_read_transformers}(pm' \text{'mem}, (pm' \text{'ro_addr} \cup$ $\text{extend}[\text{Address}, \text{Memory_Address_4G}, \text{bool}, \text{FALSE}]$ $(\text{virt_to_phys_range}(s,$ $\text{restrict}[\text{Address}, \text{Memory_Address_4G}, \text{boolean}]$ $((pm' \text{'ro_addr} \cup pm' \text{'rw_addr})),$ $(\text{singleton}(\text{Read}) \cup \text{singleton}(\text{Execute}))))))$	
{2}	$\text{unchanged_memory_invariant?}(pm' \text{'mem}, pm' \text{'states},$ $((pm' \text{'other_actions} \cup \text{memory_read_transformers}(pm' \text{'mem}, (pm' \text{'ro_addr} \cup$ $(pm' \text{'ro_addr} \cup pm' \text{'rw_addr})))$	

Hiding formulas: 2,

Repeatedly Skolemizing and flattening,

Rewriting using unchanged_memory_invariant_all_transformers, matching in * where pm gets pm_phy'mem,

Hiding formulas: 2,

Repeatedly Skolemizing and flattening,

Expanding the definition of union,

Expanding the definition of member,

Using lemma pm_plain_phy,

Using lemma pm_states,

Using lemma plain_memory_unchanged_invariant,

Case splitting on subset?(extend[Address, Memory_Address_4G, bool, FALSE] (virt_to_phys_range(s!1, restrict [Address, Memory_Address_4G, boolean] (union(pm!1'ro_addr, pm!1'rw_addr)), union (singleton(Read), singleton(Execute))))), union(pm_phy'ro_addr, pm_phy'rw_addr)),

we get 2 subgoals:

linear_plain_unchanged_memory_invariant.2.1:

<pre>{-1} (extend [Address, Memory_Address_4G, bool, FALSE](virt_to_phys_range(s', restrict [Address, {-2} plain_memory?(pm_phy) ⊃ unchanged_memory_invariant?(pm_phy' mem, pm_phy' states, ((pm_phy' other_actions ∪ memory_read_transformers(pm_phy' mem, (pm_phy' ro_addr ∪ pm_phy' rw_addr)) {-3} is_linear_plain_memory?(pm') ⊃ pm' states = pm_phy' states {-4} is_linear_plain_memory?(pm') ⊃ plain_memory?(pm_phy) {-5} (pm' other_actions(q') ∨ memory_read_transformers(pm' mem, (pm' ro_addr ∪ pm' rw_addr))(q')) ∨ memory_read_side_effect_super_transformers(pm' mem, (pm' ro_addr ∪ pm' rw_addr))(q') ∨ memory_write_side_effect_super_transformers(pm' mem, pm' rw_addr)(q') {-6} pm' states(s') {-7} is_linear_plain_memory?(pm')</pre>	<pre> extend[Address, Memory_Address_4G, bool, FALSE] (virt_to_phys_range(s', restrict [Address, Mem- ory_Address_4G, boolean] ((pm' ro_addr ∪ pm' rw_addr) (singleton(Read) ∪ singleton(Execu</pre>
---	---

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting, we get 4 subgoals:

linear_plain_unchanged_memory_invariant.2.1.1:

<pre>{-1} (extend [Address, Memory_Address_4G, bool, FALSE](virt_to_phys_range(s', restrict [Address, {-2} plain_memory?(pm_phy) {-3} unchanged_memory_invariant?(pm_phy' mem, pm_phy' states, ((pm_phy' other_actions ∪ memory_read_transformers(pm_phy' mem, (pm_phy' ro_addr ∪ pm_phy' rw_addr)) {-4} is_linear_plain_memory?(pm') {-5} pm' states = pm_phy' states {-6} pm' other_actions(q') {-7} pm' states(s')</pre>	<pre> extend[Address, Memory_Address_4G, bool, FALSE] (virt_to_phys_range(s', restrict [Address, Mem- ory_Address_4G, boolean] ((pm' ro_addr ∪ pm' rw_addr) (singleton(Read) ∪ singleton(Execu</pre>
---	---

Hiding formulas: (-1 2),

Using lemma unchanged_memory_invariant_mono,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Expanding the definition of is_linear_plain_memory?,

Applying disjunctive simplification to flatten sequent,

Keeping (-11 -16 1) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `linear_plain_unchanged_memory_invariant.2.1.1`.

`linear_plain_unchanged_memory_invariant.2.1.2`:

{-1}	(extend [Address, Memory_Address_4G, bool, FALSE](virt_to_phys_range(s', restrict [Address, Memory_A
{-2}	plain_memory?(pm_phy)
{-3}	unchanged_memory_invariant?(pm_phy'mem, pm_phy'states, ((pm_phy'other_actions ∪ memory_read_transformers(pm_phy'mem, (pm_p
	(pm_phy'ro_addr ∪ pm_phy'rw_addr))
{-4}	is_linear_plain_memory?(pm')
{-5}	pm'states = pm_phy'states
{-6}	memory_read_transformers(pm'mem, (pm'ro_addr ∪ pm'rw_addr))(q')
{-7}	pm'states(s')
{1}	unchanged_memory_invariant?(pm_phy'mem, pm_phy'states, singleton(q'), extend[Address, Memory_Address_4G, bool, FALSE] (virt_to_phys_range(s', restrict [Address, Mem- ory_Address_4G, boolean] ((pm'ro_addr ∪ pm'rw_addr)), (singleton(Read) ∪ singleton(Execute))))))

Expanding the definition of `memory_read_transformers`,

Repeatedly Skolemizing and flattening,

Replacing using formula -7,

Hiding formulas: -7,

Rewriting using `pm_read_linear`, matching in *

Expanding the definition of `linear_read`,

Using lemma `pm_memory_addr`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Rewriting using `plain_memory_unchanged_composition`, matching in * where P gets LAMBDA (pa: Address): FORALL (s: (pm_phy'states)): OK?(linear_resolve(a!1, Read)(s)) IMPLIES pa = data(linear_resolve(a!1, Read)(s)),

we get 3 subgoals:

C Proof scripts

linear_plain_unchanged_memory_invariant.2.1.2.1:

<pre> {-1} is_linear_plain_memory?(pm') {-2} union(pm'`ro_addr, pm'`rw_addr)(a') {-3} reg_base(min_linear)(a'`type_of) ≤ a' {-4} a'`type_of = a'`type_of {-5} a'`offset < max_linear_offset {-6} Mem?(type_of(a')) {-7} (extend [Address, Memory_Address_4G, bool, FALSE](virt_to_phys_range(s', restrict [Address, {-8} plain_memory?(pm_phy) {-9} unchanged_memory_invariant?(pm_phy`mem, pm_phy`states, ((pm_phy`other_actions ∪ memory_read_transformers(pm_phy`m (pm_phy`ro_addr ∪ pm_phy`rw_addr)) {-10} pm'`states = pm_phy`states {-11} pm'`states(s') </pre>	<pre> {1} ∀ (s_1: (pm_phy`states)): OK?(linear_resolve(a', Read)(s_1)) ⊃ (∀ (s: (pm_phy`states)): OK?(linear_resolve(a', Read)(s)) ⊃ data(linear_resolve(a', Read)(s_1)) = data(linear_resolve(a', Read)(s))) {2} unchanged_memory_invariant?(pm_phy`mem, pm_phy`states, singleton(expr_2_super(linear_resolve(a', Read) ## (λ (pa: Address): mem- ory_read(pm_phy`mem)(pa))))), extend[Address, Memory_Address_4G, bool, FALSE] (virt_to_phys_range(s', restrict [Address, Mem- ory_Address_4G, boolean] ((pm'`ro_addr ∪ pm'`rw_ad (singleton(Read) ∪ singleton(Ex </pre>
--	--

Hiding formulas: 2,

Repeatedly Skolemizing and flattening,

Using lemma linear_resolve_same_result,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of linear_plain_unchanged_memory_invariant.2.1.2.1.

linear_plain_unchanged_memory_invariant.2.1.2.2:

<pre> {-1} is_linear_plain_memory?(pm') {-2} union(pm'ro_addr, pm'rw_addr)(a') {-3} reg_base(min_linear)(a'type_of) ≤ a' {-4} a'type_of = a'type_of {-5} a'offset < max_linear_offset {-6} Mem?(type_of(a')) {-7} (extend [Address, Memory_Address_4G, bool, FALSE](virt_to_phys_range(s', restrict [Address, Memory_ {-8} plain_memory?(pm_phy) {-9} unchanged_memory_invariant?(pm_phy'mem, pm_phy'states, ((pm_phy'other_actions ∪ memory_read_transformers(pm_phy'mem, (pm_ (pm_phy'ro_addr ∪ pm_phy'rw_addr)) {-10} pm'states = pm_phy'states {-11} pm'states(s') </pre>	<pre> {1} unchanged_memory_invariant?(pm_phy'mem, pm_phy'states, singleton(expr_2_super(linear_resolve(a', Read))), extend[Address, Memory_Address_4G, bool, FALSE] (virt_to_phys_range(s', restrict [Address, Mem- ory_Address_4G, boolean] ((pm'ro_addr ∪ pm'rw_addr), (singleton(Read) ∪ singleton(Execute)))))) {2} unchanged_memory_invariant?(pm_phy'mem, pm_phy'states, singleton(expr_2_super(linear_resolve(a', Read) ## (λ (pa: Address): mem- ory_read(pm_phy'mem)(pa))))), extend[Address, Memory_Address_4G, bool, FALSE] (virt_to_phys_range(s', restrict [Address, Mem- ory_Address_4G, boolean] ((pm'ro_addr ∪ pm'rw_addr), (singleton(Read) ∪ singleton(Execute)))))) </pre>
--	---

Hiding formulas: 2,

Using lemma linear_resolve_unchanged_pm_phy,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Using lemma unchanged_memory_invariant_mono,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Installing automatic rewrites from: subset? member extend

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Repeatedly Skolemizing and flattening,

Rewriting using pm_resolve_address, matching in *,

we get 2 subgoals:

linear_plain_unchanged_memory_invariant.2.1.2.2.1:

<p>{-1} extend[Address, Memory_Address_4G, bool, FALSE] (virt_to_phys_range(s',</p>	<p>restrict[Address, Memory_Address_4G, boolean] ((pm'ro_addr \cup pm'rw_addr)), (singleton(Read) \cup singleton(Execute)))</p>
<p>(x') {-2} union(pm'ro_addr, pm'rw_addr)(a') {-3} is_linear_plain_memory?(pm') {-4} pm'states(s') {-5} unchanged_memory_invariant?[Physical_memory]</p>	<p>(pm_phy'mem, pm'states, singleton(expr_2_super[Physical_memory, Address](linear_resolve(a', Read))), ((pm_phy'ro_addr \cup pm_phy'rw_addr) \ extend [Address, Memory_Address_4G, bool, FALSE])</p>
<p>{-6} reg_base(min_linear)(a'type_of) $\leq a'$ {-7} a'type_of = a'type_of {-8} a'offset < max_linear_offset {-9} Mem?(type_of(a')) {-10} $\forall (x: \text{Address}):$</p>	<p>extend[Address, Memory_Address_4G, bool, FALSE] (virt_to_phys_range(s',</p>
<p>(x) \Rightarrow union(pm_phy'ro_addr, pm_phy'rw_addr)(x)</p>	<p>restrict[Address, Memory_Address_4G, boolean] ((pm'ro_addr \cup pm'rw_addr)), (singleton(Read) \cup singleton(Execute)))</p>
<p>{-11} plain_memory?(pm_phy) {-12} unchanged_memory_invariant?(pm_phy'mem, pm_phy'states,</p>	<p>((pm_phy'other_actions \cup memory_read_transformers(pm_phy'm (pm_phy'ro_addr \cup pm_phy'rw_addr))</p>
<p>{-13} pm'states = pm_phy'states</p>	
<p>{1} virt_to_phys_range(s',</p>	<p>restrict[Address, Memory_Address_4G, boolean] ((pm'ro_addr \cup pm'rw_addr)), (singleton(Read) \cup singleton(Execute)))</p>
<p>(x') {2} difference((pm_phy'ro_addr \cup pm_phy'rw_addr), extend[Address, Memory_Address_4G, bool, FALSE] (address_in_pt_range?(s',</p>	<p>restrict[Address, Mem- ory_Address_4G, boolean] ((pm'ro_addr \cup pm'rw_addr))))</p>
<p>(x') {3} unchanged_memory_invariant?(pm_phy'mem, pm_phy'states,</p>	<p>singleton(expr_2_super(linear_resolve(a', Read))), extend[Address, Memory_Address_4G, bool, FALSE] (virt_to_phys_range(s',</p>
<p>restrict [Address, Mem- ory_Address_4G, boolean]</p>	<p>((pm'ro_addr \cup pm'rw_ad (singleton(Read) \cup singleton(Ex</p>

Expanding the definition of extend,

which is trivially true.

This completes the proof of linear_plain_unchanged_memory_invariant.2.1.2.2.1.

linear_plain_unchanged_memory_invariant.2.1.2.2.2:

<p>{-1} extend[Address, Memory_Address_4G, bool, FALSE] (virt_to_phys_range(s',</p>	<p>restrict[Address, Memory_Address_4G, boolean] ((pm'ro_addr \cup pm'rw_addr)), (singleton(Read) \cup singleton(Execute))))</p>
<p> (x')</p>	<p></p>
<p>{-2} union(pm'ro_addr, pm'rw_addr)(a')</p>	<p></p>
<p>{-3} is_linear_plain_memory?(pm')</p>	<p></p>
<p>{-4} pm'states(s')</p>	<p></p>
<p>{-5} unchanged_memory_invariant?[Physical_memory] (pm_phy'mem, pm'states, singleton(expr_2_super[Physical_memory, Address](linear_resolve(a', Read))), ((pm_phy'ro_addr \cup pm_phy'rw_addr) \ extend [Address, Memory_Address_4G, bool, FALSE](address</p>	<p></p>
<p>{-6} reg_base(min_linear)(a''type_of) $\leq a'$</p>	<p></p>
<p>{-7} a''type_of = a''type_of</p>	<p></p>
<p>{-8} a''offset < max_linear_offset</p>	<p></p>
<p>{-9} Mem?(type_of(a'))</p>	<p></p>
<p>{-10} $\forall (x: \text{Address}):$ extend[Address, Memory_Address_4G, bool, FALSE] (virt_to_phys_range(s',</p>	<p>restrict[Address, Memory_Address_4G, boolean] ((pm'ro_addr \cup pm'rw_addr)), (singleton(Read) \cup singleton(Execute))))</p>
<p> (x) \Rightarrow union(pm_phy'ro_addr, pm_phy'rw_addr)(x)</p>	<p></p>
<p>{-11} plain_memory?(pm_phy)</p>	<p></p>
<p>{-12} unchanged_memory_invariant?(pm_phy'mem, pm_phy'states, ((pm_phy'other_actions \cup memory_read_transformers(pm_phy'mem, (pm_phy'ro_addr \cup pm_phy'rw_addr))</p>	<p></p>
<p>{-13} pm'states = pm_phy'states</p>	<p></p>
<p>{1} Mem?(x''type_of) \wedge $0 \leq x'$'offset \wedge x''offset < max_linear_offset</p>	<p></p>
<p>{2} difference((pm_phy'ro_addr \cup pm_phy'rw_addr), extend[Address, Memory_Address_4G, bool, FALSE] (address_in_pt_range?(s',</p>	<p>restrict[Address, Mem- ory_Address_4G, boolean] ((pm'ro_addr \cup pm'rw_addr))))</p>
<p> (x')</p>	<p></p>
<p>{3} unchanged_memory_invariant?(pm_phy'mem, pm_phy'states, singleton(expr_2_super(linear_resolve(a', Read))), extend[Address, Memory_Address_4G, bool, FALSE] (virt_to_phys_range(s',</p>	<p>restrict [Address, Mem- ory_Address_4G, boolean] ((pm'ro_addr \cup pm'rw_addr)), (singleton(Read) \cup singleton(Execute))))</p>

C Proof scripts

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `linear_plain_unchanged_memory_invariant.2.1.2.2.2`.
`linear_plain_unchanged_memory_invariant.2.1.2.3`:

{-1}	<code>is_linear_plain_memory?(pm')</code>	
{-2}	<code>union(pm'ro_addr, pm'rw_addr)(a')</code>	
{-3}	<code>reg_base(min_linear)(a'type_of) ≤ a'</code>	
{-4}	<code>a'type_of = a'type_of</code>	
{-5}	<code>a'offset < max_linear_offset</code>	
{-6}	<code>Mem?(type_of(a'))</code>	
{-7}	<code>(extend [Address, Memory_Address_4G, bool, FALSE](virt_to_phys_range(s', restrict [Address,</code>	
{-8}	<code>plain_memory?(pm_phy</code>	
{-9}	<code>unchanged_memory_invariant?(pm_phy'mem, pm_phy'states,</code>	<code>((pm_phy'other_actions ∪ memory_read_transformers(pm_phy'mem</code>
		<code>(pm_phy'ro_addr ∪ pm_phy'rw_addr))</code>
{-10}	<code>pm'states = pm_phy'states</code>	
{-11}	<code>pm'states(s')</code>	
{1}	<code>∀ (d:</code>	
	<code>(λ (pa: Address):</code>	
	<code> ∀ (s: (pm_phy'states)):</code>	
	<code> OK?(linear_resolve(a', Read)(s)) ⊃</code>	
	<code> pa = data(linear_resolve(a', Read)(s)):</code>	
	<code> unchanged_memory_invariant?(pm_phy'mem, pm_phy'states,</code>	<code>singleton(expr_2_super(memory_read(pm_phy'mem)(d))),</code>
		<code>extend[Address, Memory_Address_4G, bool, FALSE]</code>
		<code>(virt_to_phys_range(s',</code>
		<code> restrict</code>
		<code> [Address, Mem-</code>
	<code>ory_Address_4G, boolean]</code>	<code>((pm'ro_addr ∪ pm'rw_ad</code>
		<code>(singleton(Read) ∪ singleton(I</code>
{2}	<code>unchanged_memory_invariant?(pm_phy'mem, pm_phy'states,</code>	
	<code> singleton(expr_2_super(linear_resolve(a', Read) ##</code>	
		<code>(λ (pa: Address):</code>
		<code> mem-</code>
	<code> ory_read(pm_phy'mem)(pa))),</code>	
		<code>extend[Address, Memory_Address_4G, bool, FALSE]</code>
		<code>(virt_to_phys_range(s',</code>
		<code> restrict</code>
		<code> [Address, Mem-</code>
	<code>ory_Address_4G, boolean]</code>	<code>((pm'ro_addr ∪ pm'rw_ad</code>
		<code>(singleton(Read) ∪ singleton(Ex</code>

Hiding formulas: 2,

Repeatedly Skolemizing and flattening,

Using lemma `unchanged_memory_invariant_mono`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Hiding formulas: 2,

Using lemma `memory_read_transformers_memory_read`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 2 subgoals:

linear_plain_unchanged_memory_invariant.2.1.2.3.1:

{-1}	memory_read_transformers(pm_phy' mem, (pm_phy' ro_addr \cup pm_phy' rw_addr)) (expr_2_super(memory_read(pm_phy' mem)(d')))
{-2}	(extend [Address, Memory_Address_4G, bool, FALSE](virt_to_phys_range(s', restrict [Address, Memory_
{-3}	unchanged_memory_invariant?(pm_phy' mem, pm_phy' states, (pm_phy' other_actions \cup memory_read_transformers(pm_phy' mem, (pm_phy' ro_addr \cup pm_phy' rw_addr))
{-4}	$\forall (s: (pm_phy' states)):$ OK?(linear_resolve(a', Read)(s)) \supset d' = data(linear_resolve(a', Read)(s))
{-5}	is_linear_plain_memory?(pm')
{-6}	union(pm' ro_addr, pm' rw_addr)(a')
{-7}	reg_base(min_linear)(a' type_of) \leq a'
{-8}	a' type_of = a' type_of
{-9}	a' offset < max_linear_offset
{-10}	Mem?(type_of(a'))
{-11}	plain_memory?(pm_phy)
{-12}	pm' states = pm_phy' states
{-13}	pm' states(s')
{1}	(singleton(expr_2_super(memory_read(pm_phy' mem)(d'))) \subseteq ((pm_phy' other_actions \cup memory_read_transfor

Keeping (-1 1) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of linear_plain_unchanged_memory_invariant.2.1.2.3.1.

linear_plain_unchanged_memory_invariant.2.1.2.3.2:

{-1}	(extend [Address, Memory_Address_4G, bool, FALSE](virt_to_phys_range(s', restrict [Address, Memory_
{-2}	unchanged_memory_invariant?(pm_phy' mem, pm_phy' states, (pm_phy' other_actions \cup memory_read_transformers(pm_phy' mem, (pm_phy' ro_addr \cup pm_phy' rw_addr))
{-3}	$\forall (s: (pm_phy' states)):$ OK?(linear_resolve(a', Read)(s)) \supset d' = data(linear_resolve(a', Read)(s))
{-4}	is_linear_plain_memory?(pm')
{-5}	union(pm' ro_addr, pm' rw_addr)(a')
{-6}	reg_base(min_linear)(a' type_of) \leq a'
{-7}	a' type_of = a' type_of
{-8}	a' offset < max_linear_offset
{-9}	Mem?(type_of(a'))
{-10}	plain_memory?(pm_phy)
{-11}	pm' states = pm_phy' states
{-12}	pm' states(s')
{1}	union(pm_phy' ro_addr, pm_phy' rw_addr)(d')
{2}	(singleton(expr_2_super(memory_read(pm_phy' mem)(d'))) \subseteq ((pm_phy' other_actions \cup memory_read_transfor

Instantiating quantified variables,

Using lemma pm_linear_blessed,

Expanding the definition of linear_blessed?,

Using lemma pm_linear_resolve_read_ok,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Hiding formulas: (-3 -4 -6 -7 -8 -9),

Expanding the definition of subset?,

Instantiating quantified variables,

we get 3 subgoals:

linear_plain_unchanged_memory_invariant.2.1.2.3.2.1:

{-1}	is_linear_plain_memory?(pm')
{-2}	OK?(linear_resolve(a', Read)(s'))
{-3}	(d' ∈ virt_to_phys_range(s', (restrict [Address, Memory_Address_4G, boolean](pm'ro_addr) ∪ r \Rightarrow (d' ∈ restrict[Address, Memory_Address_4G, boolean]((pm_phy'ro_addr ∪ pm_phy'rw_addr)))
{-4}	∀ (x: Address): (x ∈ extend [Address, Memory_Address_4G, bool, FALSE](virt_to_phys_range(s', restrict [A \Rightarrow (x ∈ (pm_phy'ro_addr ∪ pm_phy'rw_addr))
{-5}	unchanged_memory_invariant?(pm_phy'mem, pm_phy'states, ((pm_phy'other_actions ∪ memory_read_transformers(pm_phy'm (pm_phy'ro_addr ∪ pm_phy'rw_addr))
{-6}	d' = data(linear_resolve(a', Read)(s'))
{-7}	union(pm'ro_addr, pm'rw_addr)(a')
{-8}	reg_base(min_linear)(a'type_of) ≤ a'
{-9}	a'type_of = a'type_of
{-10}	a'offset < max_linear_offset
{-11}	Mem?(type_of(a'))
{-12}	plain_memory?(pm_phy)
{-13}	pm'states = pm_phy'states
{-14}	pm'states(s')
{1}	union(pm_phy'ro_addr, pm_phy'rw_addr)(d')
{2}	∀ (x: [Physical_memory → SuperResult[Physical_memory]]): (x ∈ singleton(expr_2_super(memory_read(pm_phy'mem)(d')))) \Rightarrow (x ∈ ((pm_phy'other_actions ∪ memory_read_transformers(pm_phy'mem, (pm_phy'ro_addr ∪ p

Expanding the definition of member,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 2 subgoals:

```
linear_plain_unchanged_memory_invariant.2.1.2.3.2.1.1:
```

{-1}	$\text{is_linear_plain_memory?}(pm')$
{-2}	$\text{OK?}(\text{linear_resolve}(a', \text{Read})(s'))$
{-3}	$\text{restrict}[\text{Address}, \text{Memory_Address_4G}, \text{boolean}]((pm_phy'ro_addr \cup pm_phy'rw_addr))(d')$
{-4}	$\forall (x: \text{Address}):$ $\quad \text{extend}[\text{Address}, \text{Memory_Address_4G}, \text{bool}, \text{FALSE}]$ $\quad \quad (\text{virt_to_phys_range}(s',$ $\quad \quad \quad \text{restrict}[\text{Address}, \text{Memory_Address_4G}, \text{boolean}]$ $\quad \quad \quad \quad ((pm'ro_addr \cup pm'rw_addr)),$ $\quad \quad \quad \quad (\text{singleton}(\text{Read}) \cup \text{singleton}(\text{Execute}))))$ $\quad (x)$ $\quad \Rightarrow \text{union}(pm_phy'ro_addr, pm_phy'rw_addr)(x)$
{-5}	$\text{unchanged_memory_invariant?}(pm_phy'mem, pm_phy'states,$ $\quad ((pm_phy'other_actions \cup \text{memory_read_transformers}(pm_phy'mem, (pm_phy'ro_addr \cup pm_phy'rw_addr)))$
{-6}	$d' = \text{data}(\text{linear_resolve}(a', \text{Read})(s'))$
{-7}	$\text{union}(pm'ro_addr, pm'rw_addr)(a')$
{-8}	$\text{reg_base}(\text{min_linear})(a'\text{'type_of}) \leq a'$
{-9}	$a'\text{'type_of} = a'\text{'type_of}$
{-10}	$a'\text{'offset} < \text{max_linear_offset}$
{-11}	$\text{Mem?}(\text{type_of}(a'))$
{-12}	$\text{plain_memory?}(pm_phy)$
{-13}	$pm'states = pm_phy'states$
{-14}	$pm'states(s')$
{1}	$\text{union}(pm_phy'ro_addr, pm_phy'rw_addr)(d')$
{2}	$\forall (x: [\text{Physical_memory} \rightarrow \text{SuperResult}[\text{Physical_memory}]]):$ $x = \text{expr_2_super}(\text{memory_read}(pm_phy'mem)(d')) \Rightarrow$ $\quad \text{union}((pm_phy'other_actions \cup \text{memory_read_transformers}(pm_phy'mem, (pm_phy'ro_addr \cup pm_phy'rw_addr))$ $\quad \quad (\text{memory_read_side_effect_super_transformers}(pm_phy'mem, (pm_phy'ro_addr \cup pm_phy'rw_addr))$ $\quad (x)$

Expanding the definition of restrict,

which is trivially true.

This completes the proof of linear_plain_unchanged_memory_invariant.2.1.2.3.2.1.1.

linear_plain_unchanged_memory_invariant.2.1.2.3.2.1.2:

{-1}	is_linear_plain_memory?(pm')
{-2}	OK?(linear_resolve(a', Read)(s'))
{-3}	$\forall (x: \text{Address}):$ extend[Address, Memory_Address_4G, bool, FALSE] (virt_to_phys_range(s', restrict[Address, Memory_Address_4G, boolean] ((pm'ro_addr \cup pm'rw_addr), (singleton(Read) \cup singleton(Execute)))) (x) \Rightarrow union(pm_phy'ro_addr, pm_phy'rw_addr)(x)
{-4}	unchanged_memory_invariant?(pm_phy'mem, pm_phy'states, ((pm_phy'other_actions \cup memory_read_transformers(pm_phy'mem, (pm_phy'ro_addr \cup pm_phy'rw_addr)))
{-5}	d' = data(linear_resolve(a', Read)(s'))
{-6}	union(pm'ro_addr, pm'rw_addr)(a')
{-7}	reg_base(min_linear)(a'type_of) \leq a'
{-8}	a'type_of = a'type_of
{-9}	a'offset < max_linear_offset
{-10}	Mem?(type_of(a'))
{-11}	plain_memory?(pm_phy)
{-12}	pm'states = pm_phy'states
{-13}	pm'states(s')
{1}	virt_to_phys_range(s', (restrict[Address, Memory_Address_4G, boolean](pm'ro_addr) \cup restrict (singleton(Read) \cup singleton(Execute))) (d')
{2}	union(pm_phy'ro_addr, pm_phy'rw_addr)(d')
{3}	$\forall (x: [\text{Physical_memory} \rightarrow \text{SuperResult}[\text{Physical_memory}]]):$ $x = \text{expr_2_super}(\text{memory_read}(\text{pm_phy'mem})(d')) \Rightarrow$ union((pm_phy'other_actions \cup memory_read_transformers(pm_phy'mem, (pm_phy'ro_addr \cup pm_phy'rw_addr) (memory_read_side_effect_super_transformers(pm_phy'mem, (pm_phy'ro_addr \cup pm_phy'rw_addr)))) (x)

Expanding the definition of virt_to_phys_range,

Instantiating quantified variables,

we get 2 subgoals:

linear_plain_unchanged_memory_invariant.2.1.2.3.2.1.2.1:

{-1}	is_linear_plain_memory?(pm')
{-2}	OK?(linear_resolve(a', Read)(s'))
{-3}	$\forall (x: \text{Address}):$ extend[Address, Memory_Address_4G, bool, FALSE] $(\lambda (\text{phy_a}: \text{Memory_Address_4G}):$ $\quad \exists (\text{lin_a}:$ $\quad \quad (\text{restrict}[\text{Address}, \text{Memory_Address_4G}, \text{boolean}]$ $\quad \quad \quad ((\text{pm}'\text{ro_addr} \cup \text{pm}'\text{rw_addr}))),$ $\quad \quad \text{ac}: ((\text{singleton}(\text{Read}) \cup \text{singleton}(\text{Execute}))))):$ $\quad \quad \text{OK?}(\text{linear_resolve}(\text{lin_a}, \text{ac})(s')) \wedge$ $\quad \quad \text{data}(\text{linear_resolve}(\text{lin_a}, \text{ac})(s')) = \text{phy_a}$ $\quad (x)$ $\Rightarrow \text{union}(\text{pm_phy}'\text{ro_addr}, \text{pm_phy}'\text{rw_addr})(x)$
{-4}	unchanged_memory_invariant?(pm_phy'mem, pm_phy'states, $(\text{pm_phy}'\text{other_actions} \cup \text{memory_read_transformers}(\text{pm_phy}'\text{mem}, (\text{pm_phy}'\text{ro_addr} \cup \text{pm_phy}'\text{rw_addr})))$
{-5}	$d' = \text{data}(\text{linear_resolve}(a', \text{Read})(s'))$
{-6}	$\text{union}(\text{pm}'\text{ro_addr}, \text{pm}'\text{rw_addr})(a')$
{-7}	$\text{reg_base}(\text{min_linear})(a'\text{type_of}) \leq a'$
{-8}	$a'\text{type_of} = a'\text{type_of}$
{-9}	$a'\text{offset} < \text{max_linear_offset}$
{-10}	$\text{Mem?}(\text{type_of}(a'))$
{-11}	plain_memory?(pm_phy)
{-12}	$\text{pm}'\text{states} = \text{pm_phy}'\text{states}$
{-13}	$\text{pm}'\text{states}(s')$
{1}	$\text{OK?}(\text{linear_resolve}(a', \text{Read})(s')) \wedge \text{data}(\text{linear_resolve}(a', \text{Read})(s')) = d'$
{2}	$\text{union}(\text{pm_phy}'\text{ro_addr}, \text{pm_phy}'\text{rw_addr})(d')$
{3}	$\forall (x: [\text{Physical_memory} \rightarrow \text{SuperResult}[\text{Physical_memory}]]):$ $x = \text{expr_2_super}(\text{memory_read}(\text{pm_phy}'\text{mem})(d')) \Rightarrow$ $\quad \text{union}((\text{pm_phy}'\text{other_actions} \cup \text{memory_read_transformers}(\text{pm_phy}'\text{mem}, (\text{pm_phy}'\text{ro_addr} \cup \text{pm_phy}'\text{rw_addr})))$ $\quad \quad (\text{memory_read_side_effect_super_transformers}(\text{pm_phy}'\text{mem}, (\text{pm_phy}'\text{ro_addr} \cup \text{pm_phy}'\text{rw_addr})))$ $\quad (x)$

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of linear_plain_unchanged_memory_invariant.2.1.2.3.2.1.2.1.

linear_plain_unchanged_memory_invariant.2.1.2.3.2.1.2.2:

<pre> {-1} is_linear_plain_memory?(pm') {-2} OK?(linear_resolve(a', Read)(s')) {-3} $\forall (x: \text{Address}):$ extend[Address, Memory_Address_4G, bool, FALSE] (λ (phy_a: Memory_Address_4G): \exists (lin_a: (restrict[Address, Memory_Address_4G, boolean] ((pm'ro_addr \cup pm'rw_addr))), ac: ((singleton(Read) \cup singleton(Execute))): OK?(linear_resolve(lin_a, ac)(s')) \wedge data(linear_resolve(lin_a, ac)(s')) = phy_a) (x) \Rightarrow union(pm_phy'ro_addr, pm_phy'rw_addr)(x)) {-4} unchanged_memory_invariant?(pm_phy'mem, pm_phy'states, ((pm_phy'other_actions \cup memory_read_transformers(pm_phy'mem, (pm_phy'ro_addr \cup pm_phy'rw_addr))) {-5} $d' = \text{data}(\text{linear_resolve}(a', \text{Read})(s'))$ {-6} union(pm'ro_addr, pm'rw_addr)(a') {-7} reg_base(min_linear)(a'type_of) $\leq a'$ {-8} a''type_of = a''type_of {-9} a''offset < max_linear_offset {-10} Mem?(type_of(a')) {-11} plain_memory?(pm_phy) {-12} pm'states = pm_phy'states {-13} pm'states(s') </pre>	<pre> {1} union[Memory_Address_4G] (restrict[Address, Memory_Address_4G, boolean](pm'ro_addr), restrict[Address, Memory_Address_4G, boolean](pm'rw_addr)) (a') {2} union(pm_phy'ro_addr, pm_phy'rw_addr)(d') {3} $\forall (x: [\text{Physical_memory} \rightarrow \text{SuperResult}[\text{Physical_memory}]]):$ $x = \text{expr_2_super}(\text{memory_read}(\text{pm_phy'mem})(d')) \Rightarrow$ union((pm_phy'other_actions \cup memory_read_transformers(pm_phy'mem, (pm_phy'ro_addr \cup pm_phy'rw_addr)) (memory_read_side_effect_super_transformers(pm_phy'mem, (pm_phy'ro_addr \cup pm_phy'rw_addr)) (x) </pre>
--	---

Keeping (-6 1) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of linear_plain_unchanged_memory_invariant.2.1.2.3.2.1.2.2.

linear_plain_unchanged_memory_invariant.2.1.2.3.2.2:

{-1}	is_linear_plain_memory?(pm')
{-2}	OK?(linear_resolve(a', Read)(s'))
{-3}	$\forall (x: \text{Address}):$ $(x \in \text{extend} [\text{Address}, \text{Memory_Address_4G}, \text{bool}, \text{FALSE}](\text{virt_to_phys_range}(s', \text{restrict} [\text{Address}, \text{Memory_Address_4G}, \text{bool}, \text{FALSE}])))$ $\Rightarrow (x \in (\text{pm_phy}'\text{ro_addr} \cup \text{pm_phy}'\text{rw_addr}))$
{-4}	unchanged_memory_invariant?(pm_phy'mem, pm_phy'states, $((\text{pm_phy}'\text{other_actions} \cup \text{memory_read_transformers}(\text{pm_phy}'\text{mem}, (\text{pm_phy}'\text{ro_addr} \cup \text{pm_phy}'\text{rw_addr})))$)
{-5}	$d' = \text{data}(\text{linear_resolve}(a', \text{Read})(s'))$
{-6}	$\text{union}(\text{pm_phy}'\text{ro_addr}, \text{pm_phy}'\text{rw_addr})(a')$
{-7}	$\text{reg_base}(\text{min_linear})(a'\text{'type_of}) \leq a'$
{-8}	$a'\text{'type_of} = a'\text{'type_of}$
{-9}	$a'\text{'offset} < \text{max_linear_offset}$
{-10}	$\text{Mem?}(\text{type_of}(a'))$
{-11}	plain_memory?(pm_phy)
{-12}	pm' states = pm_phy' states
{-13}	pm' states(s')
{1}	$\text{Mem?}(d'\text{'type_of}) \wedge 0 \leq d'\text{'offset} \wedge d'\text{'offset} < \text{max_linear_offset}$
{2}	$\text{union}(\text{pm_phy}'\text{ro_addr}, \text{pm_phy}'\text{rw_addr})(d')$
{3}	$\forall (x: [\text{Physical_memory} \rightarrow \text{SuperResult}[\text{Physical_memory}]]):$ $(x \in \text{singleton}(\text{expr_2_super}(\text{memory_read}(\text{pm_phy}'\text{mem})(d')))) \Rightarrow$ $(x \in ((\text{pm_phy}'\text{other_actions} \cup \text{memory_read_transformers}(\text{pm_phy}'\text{mem}, (\text{pm_phy}'\text{ro_addr} \cup \text{pm_phy}'\text{rw_addr}))))$

Using lemma linear_resolve_memory_address,

Expanding the definition of every,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of linear_plain_unchanged_memory_invariant.2.1.2.3.2.2.

linear_plain_unchanged_memory_invariant.2.1.2.3.2.3:

{-1}	is_linear_plain_memory?(pm')
{-2}	OK?(linear_resolve(a', Read)(s'))
{-3}	$\forall (x: \text{Memory_Address_4G}):$ $(x \in \text{virt_to_phys_range}(s', (\text{restrict} [\text{Address}, \text{Memory_Address_4G}, \text{boolean}](\text{pm}'\text{ro_addr}) \cup \text{pm}'\text{rw_addr})))$ \Rightarrow $(x \in \text{restrict}[\text{Address}, \text{Memory_Address_4G}, \text{boolean}]((\text{pm_phy}'\text{ro_addr} \cup \text{pm_phy}'\text{rw_addr})))$
{-4}	$\forall (x: \text{Address}):$ $(x \in \text{extend} [\text{Address}, \text{Memory_Address_4G}, \text{bool}, \text{FALSE}](\text{virt_to_phys_range}(s', \text{restrict} [\text{Address}, \text{Memory_Address_4G}, \text{boolean}](\text{pm}'\text{ro_addr}) \cup \text{pm}'\text{rw_addr}))))$ $\Rightarrow (x \in (\text{pm_phy}'\text{ro_addr} \cup \text{pm_phy}'\text{rw_addr}))$
{-5}	unchanged_memory_invariant?(pm_phy'mem, pm_phy'states, $((\text{pm_phy}'\text{other_actions} \cup \text{memory_read_transformers}(\text{pm_phy}'\text{mem}, \text{pm_phy}'\text{ro_addr} \cup \text{pm_phy}'\text{rw_addr})))$
{-6}	$d' = \text{data}(\text{linear_resolve}(a', \text{Read})(s'))$
{-7}	$\text{union}(\text{pm}'\text{ro_addr}, \text{pm}'\text{rw_addr})(a')$
{-8}	$\text{reg_base}(\text{min_linear})(a'\text{type_of}) \leq a'$
{-9}	$a'\text{type_of} = a'\text{type_of}$
{-10}	$a'\text{offset} < \text{max_linear_offset}$
{-11}	Mem?(type_of(a'))
{-12}	plain_memory?(pm_phy)
{-13}	pm'states = pm_phy'states
{-14}	pm'states(s')
{1}	Mem?(d'type_of) \wedge $0 \leq d'\text{offset} \wedge d'\text{offset} < \text{max_linear_offset}$
{2}	$\text{union}(\text{pm_phy}'\text{ro_addr}, \text{pm_phy}'\text{rw_addr})(d')$
{3}	$\forall (x: [\text{Physical_memory} \rightarrow \text{SuperResult}[\text{Physical_memory}]]):$ $(x \in \text{singleton}(\text{expr_2_super}(\text{memory_read}(\text{pm_phy}'\text{mem})(d')))) \Rightarrow$ $(x \in ((\text{pm_phy}'\text{other_actions} \cup \text{memory_read_transformers}(\text{pm_phy}'\text{mem}, (\text{pm_phy}'\text{ro_addr} \cup \text{pm_phy}'\text{rw_addr}))))$

Using lemma linear_resolve_memory_address,

Expanding the definition of every,

Using lemma pm_linear_resolve_read_ok,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of linear_plain_unchanged_memory_invariant.2.1.2.3.2.3.

linear_plain_unchanged_memory_invariant.2.1.3:

{-1}	$(\text{extend} [\text{Address}, \text{Memory_Address_4G}, \text{bool}, \text{FALSE}](\text{virt_to_phys_range}(s', \text{restrict} [\text{Address}, \text{Memory_Address_4G}, \text{boolean}](\text{pm}'\text{ro_addr}) \cup \text{pm}'\text{rw_addr}))))$
{-2}	plain_memory?(pm_phy)
{-3}	unchanged_memory_invariant?(pm_phy'mem, pm_phy'states, $((\text{pm_phy}'\text{other_actions} \cup \text{memory_read_transformers}(\text{pm_phy}'\text{mem}, \text{pm_phy}'\text{ro_addr} \cup \text{pm_phy}'\text{rw_addr})))$
{-4}	is_linear_plain_memory?(pm')
{-5}	pm'states = pm_phy'states
{-6}	memory_read_side_effect_super_transformers(pm'mem, (pm'ro_addr \cup pm'rw_addr))(q')
{-7}	pm'states(s')
{1}	unchanged_memory_invariant?(pm_phy'mem, pm_phy'states, singleton(q'), $\text{extend}[\text{Address}, \text{Memory_Address_4G}, \text{bool}, \text{FALSE}]$ $(\text{virt_to_phys_range}(s',$ restrict $[\text{Address}, \text{Memory_Address_4G}, \text{boolean}]$ $(\text{pm}'\text{ro_addr} \cup \text{pm}'\text{rw_addr})))$ $(\text{singleton}(\text{Read}) \cup \text{singleton}(\text{Execute}))$

C.116 Proofs for Linear_Memory_Blessing_Properties (challenge-linear.pvs)

Expanding the definition of `memory_read_side_effect_super_transformers`,

Repeatedly Skolemizing and flattening,

Replacing using formula -8,

Hiding formulas: (-1 -8),

Rewriting using `pm_read_side_effect_linear`, matching in `*`,

Expanding the definition of `linear_read_side_effect`,

Case splitting on `Mem?(type_of(a!1))`,

we get 2 subgoals:

linear_plain_unchanged_memory_invariant.2.1.3.1:

<pre> {-1} Mem?(type_of(a')) {-2} (extend [Address, Memory_Address_4G, bool, FALSE](virt_to_phys_range(s', restrict [Address, {-3} plain_memory?(pm_phy) {-4} unchanged_memory_invariant?(pm_phy'mem, pm_phy'states, ((pm_phy'other_actions ∪ memory_read_transformers(pm_phy'mem) (pm_phy'ro_addr ∪ pm_phy'rw_addr)) {-5} is_linear_plain_memory?(pm') {-6} pm'states = pm_phy'states {-7} (address_block(a', length(bl')) ⊆ (pm'ro_addr ∪ pm'rw_addr)) {-8} pm'states(s') </pre>	<pre> singleton(expr_2_super(λ (s : Linear_memory [Physical_memory, pm_p IF null?(bl') ∨ (reg_base (min_linear)(type_of(a')) ≤ a' ∧ a' + length(bl') ≤ reg_size (max_linear)(type_of(a')), THEN IF Mem?(type_of(a')) THEN ap- (split(min_page, lin- (s) ELSE mem- (pm_phy'mem) (a', bl', cp')(s) ENDIF ELSE Fatal ENDIF)), extend[Address, Memory_Address_4G, bool, FALSE] (virt_to_phys_range(s', restrict [Address, Mem- ((pm'ro_addr ∪ pm'rw_addr) (singleton(Read) ∪ singleton(Exec </pre>
--	--

Case splitting on null?(bl!1),

we get 2 subgoals:

linear_plain_unchanged_memory_invariant.2.1.3.1.1:

<pre> {-1} null?(bl') {-2} Mem?(type_of(a')) {-3} (extend [Address, Memory_Address_4G, bool, FALSE](virt_to_phys_range(s', restrict [Address, Memory_A {-4} plain_memory?(pm_phy) {-5} unchanged_memory_invariant?(pm_phy'mem, pm_phy'states, ((pm_phy'other_actions ∪ memory_read_transformers(pm_phy'mem, (pm_ph (pm_phy'ro_addr ∪ pm_phy'rw_addr)) {-6} is_linear_plain_memory?(pm') {-7} pm'states = pm_phy'states {-8} (address_block(a', length(bl')) ⊆ (pm'ro_addr ∪ pm'rw_addr)) {-9} pm'states(s') </pre>	<pre> Linear_memory [Physical_memory, pm_phy]: IF null?(bl') ∨ (reg_base (min_linear)(type_of(a')) ≤ a' ∧ a' + length(bl') ≤ reg_size (max_linear)(type_of(a'))) THEN IF Mem?(type_of(a')) THEN ap- (split(min_page, a', bl'), lin- (s) ELSE mem- (pm_phy'mem) (a', bl', cp')(s) ENDIF ELSE Fatal ENDIF), extend[Address, Memory_Address_4G, bool, FALSE] (virt_to_phys_range(s', restrict [Address, Mem- ((pm'ro_addr ∪ pm'rw_addr), (singleton(Read) ∪ singleton(Execute)))))) </pre>
--	---

Simplifying, rewriting, and recording with decision procedures,
 Expanding the definition of apply_side_effects,
 Expanding the definition of ok_result,
 Expanding the definition of reduce,

C Proof scripts

Using lemma `split_null`,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 Keeping (1) and hiding *,
 Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of `linear_plain_unchanged_memory_invariant.2.1.3.1.1`.
`linear_plain_unchanged_memory_invariant.2.1.3.1.2`:

<pre> {-1} Mem?(type_of(a')) {-2} (extend [Address, Memory_Address_4G, bool, FALSE](virt_to_phys_range(s', restrict [Address, {-3} plain_memory?(pm_phy) {-4} unchanged_memory_invariant?(pm_phy'mem, pm_phy'states, ((pm_phy'other_actions ∪ memory_read_transformers(pm_phy'mem) (pm_phy'ro_addr ∪ pm_phy'rw_addr)) {-5} is_linear_plain_memory?(pm') {-6} pm'states = pm_phy'states {-7} (address_block(a', length(bl')) ⊆ (pm'ro_addr ∪ pm'rw_addr)) {-8} pm'states(s')</pre>	<pre> Linear_memory [Physical_memory, pm_p IF null?(bl') ∨ (reg_base (min_linear)(type_of(a')) ≤ a' ∧ a' + length(bl') ≤ reg_size (max_linear)(type_of(a')) THEN IF Mem?(type_of(a')) THEN ap- (split(min_page, lin- (s) ELSE mem- (pm_phy'mem) (a', bl', cp')(s) ENDIF ELSE Fatal ENDIF)), extend[Address, Memory_Address_4G, bool, FALSE] (virt_to_phys_range(s', restrict [Address, Mem- ((pm'ro_addr ∪ pm'rw_addr) (singleton(Read) ∪ singleton(Exe</pre>
---	---

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 3 subgoals:

linear_plain_unchanged_memory_invariant.2.1.3.1.2.1:

<pre> {-1} Mem?(type_of(a')) {-2} (extend [Address, Memory_Address_4G, bool, FALSE](virt_to_phys_range(s', restrict [Address, Memory_ {-3} plain_memory?(pm_phy) {-4} unchanged_memory_invariant?(pm_phy' mem, pm_phy' states, ((pm_phy'other_actions ∪ memory_read_transformers(pm_phy' mem, (pm_ (pm_phy'ro_addr ∪ pm_phy'rw_addr)) {-5} is_linear_plain_memory?(pm') {-6} pm' states = pm_phy' states {-7} (address_block(a', length(bl')) ⊆ (pm'ro_addr ∪ pm'rw_addr)) {-8} reg_base(min_linear)(type_of(a')) ≤ a' {-9} a' + length(bl') ≤ (#type_of := type_of(a'), offset := max_linear_offset#) {-10} pm' states(s') </pre>	<hr/> <pre> {1} null?(bl') {2} unchanged_memory_invariant?(pm_phy' mem, pm_phy' states, singleton(expr_2_super(λ (s: Linear_memory [Physical_memory, pm_phy]): apply_side_effects(split (min_page, a', bl'), lin- (s))), extend[Address, Memory_Address_4G, bool, FALSE] (virt_to_phys_range(s', restrict [Address, Mem- ((pm'ro_addr ∪ pm'rw_addr)), (singleton(Read) ∪ singleton(Execute)))))) </pre>
--	---

Using lemma address_block_split_type,

we get 2 subgoals:

linear_plain_unchanged_memory_invariant.2.1.3.1.2.1.1:

<pre> {-1} is_linear_plain_memory?(pm') ∧ (address_block(a', length(bl')) ⊆ (pm'ro_addr ∪ pm'rw_addr)) ⊃ every(λ (t: [Address, list[Byte]]): Mem?(t'1'type_of) ∧ 0 ≤ t'1'offset ∧ t'1'offset < max_linear_offset) (split(min_page, a', bl')) {-2} Mem?(type_of(a')) {-3} (extend [Address, Memory_Address_4G, bool, FALSE](virt_to_phys_range(s', restrict [Address, Mem- {-4} plain_memory?(pm_phy) {-5} unchanged_memory_invariant?(pm_phy'mem, pm_phy'states, ((pm_phy'other_actions ∪ memory_read_transformers(pm_phy'mem, (pm_phy'ro_addr ∪ pm_phy'rw_addr)) {-6} is_linear_plain_memory?(pm') {-7} pm'states = pm_phy'states {-8} (address_block(a', length(bl')) ⊆ (pm'ro_addr ∪ pm'rw_addr)) {-9} reg_base(min_linear)(type_of(a')) ≤ a' {-10} a' + length(bl') ≤ (#type_of := type_of(a'), offset := max_linear_offset#) {-11} pm'states(s') </pre>	<pre> {1} null?(bl') {2} unchanged_memory_invariant?(pm_phy'mem, pm_phy'states, singleton(expr_2_super(λ (s: Linear_memory [Physical_memory, pm- apply_side_effects(split (min_p a', bl'), lin- (s))), extend[Address, Memory_Address_4G, bool, FALSE] (virt_to_phys_range(s', restrict [Address, Mem- ory_Address_4G, boolean] ((pm'ro_addr ∪ pm'rw_ad (singleton(Read) ∪ singleton(Ex </pre>
--	--

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Using lemma `apply_side_effects_unchanged`,

Using lemma `unchanged_memory_invariant_mono`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 3 subgoals:

linear_plain_unchanged_memory_invariant.2.1.3.1.2.1.1.1:

{-1}	is_linear_plain_memory?(pm')	
{-2}	(address_block(a', length(bl')) \subseteq (pm'ro_addr \cup pm'rw_addr))	
{-3}	every(λ (t: [Address, list[Byte]]): Mem?(t'1'type_of) \wedge 0 \leq t'1'offset \wedge t'1'offset < max_linear_offset) (split(min_page, a', bl'))	
{-4}	Mem?(type_of(a'))	
{-5}	(extend [Address, Memory_Address_4G, bool, FALSE](virt_to_phys_range(s', restrict [Address, Memory_	
{-6}	plain_memory?(pm_phy)	
{-7}	unchanged_memory_invariant?(pm_phy'mem, pm_phy'states, (pm_phy'other_actions \cup memory_read_transformers(pm_phy'mem, (pm- (pm_phy'ro_addr \cup pm_phy'rw_addr))	
{-8}	pm'states = pm_phy'states	
{-9}	reg_base(min_linear)(type_of(a')) \leq a'	
{-10}	a' + length(bl') \leq (#type_of := type_of(a'), offset := max_linear_offset#)	
{-11}	pm'states(s')	
{1}	unchanged_memory_invariant?(pm_phy'mem, pm_phy'states, singleton(expr_2_super(apply_side_effects(split (min_page, a', bl') lin- ear_read_side_effect_in_page))), extend[Address, Memory_Address_4G, bool, FALSE] (restrict[Address, Mem- ory_Address_4G, boolean] (pm_phy'ro_addr \cup pm_phy'rw_addr))))	
{2}	unchanged_memory_invariant?(pm_phy'mem, pm_phy'states, singleton(expr_2_super(λ (s: Linear_memory [Physical_memory, pm_phy]): apply_side_effects(split (min_page, a', bl'), lin- ear_read_side_effect_in_page) (s))), extend[Address, Memory_Address_4G, bool, FALSE] (virt_to_phys_range(s', restrict [Address, Mem- ory_Address_4G, boolean] (pm'ro_addr \cup pm'rw_addr)), (singleton(Read) \cup singleton(Execute))))))	
{3}	every(λ (e: [Memory_Address_4G, list[Byte]]): unchanged_memory_invariant?(pm_phy'mem, pm_phy'states, singleton(expr_2_super(linear_read_side_effect_in_page (e))), extend[Address, Mem- ory_Address_4G, bool, FALSE] (restrict[Address, Mem- ory_Address_4G, boolean] (pm_phy'ro_addr \cup pm_phy'rw_addr))))	
{4}	(split(min_page, a', bl')) null?(bl')	

C Proof scripts

Hiding formulas: (-7 1 2),

Using lemma `split_linear_read_side_effects_unchanged_memory`,

we get 2 subgoals:

`linear_plain_unchanged_memory_invariant.2.1.3.1.2.1.1.1.1.1:`

{-1}	is_linear_plain_memory?(pm') \wedge (address_block(a', length(bl'))) \subseteq (pm'ro_addr \cup pm'rw_addr)) \supset every(λ (e: [Memory_Address_4G, list[Byte]]): unchanged_memory_invariant?(pm_phy'mem, pm'states, singleton(expr_2_super(linear_read_side_effect_in_pa (e))), (pm_phy'ro_addr \cup pm_phy'rw_addr))) (split(min_page, a', bl'))
{-2}	is_linear_plain_memory?(pm')
{-3}	(address_block(a', length(bl'))) \subseteq (pm'ro_addr \cup pm'rw_addr)
{-4}	every(λ (t: [Address, list[Byte]]): Mem?(t'1'type_of) \wedge 0 \leq t'1'offset \wedge t'1'offset $<$ max_linear_offset) (split(min_page, a', bl'))
{-5}	Mem?(type_of(a'))
{-6}	(extend [Address, Memory_Address_4G, bool, FALSE](virt_to_phys_range(s', restrict [Address,
{-7}	plain_memory?(pm_phy)
{-8}	pm'states = pm_phy'states
{-9}	reg_base(min_linear)(type_of(a')) \leq a'
{-10}	a' + length(bl') \leq (#type_of := type_of(a'), offset := max_linear_offset#)
{-11}	pm'states(s')
{1}	every(λ (e: [Memory_Address_4G, list[Byte]]): unchanged_memory_invariant?(pm_phy'mem, pm_phy'states, singleton(expr_2_super(linear_read_side_effect_in_pa (e))), extend[Address, Mem- ory_Address_4G, bool, FALSE] (restrict[Address, Mem- ory_Address_4G, boolean] ((pm_phy'ro_addr \cup pm_phy'rw_addr)))) (split(min_page, a', bl'))
{2}	null?(bl')

Simplifying, rewriting, and recording with decision procedures,

Using lemma `every_implied`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Hiding formulas: (-1 2),

Repeatedly Skolemizing and flattening,

Using lemma `unchanged_memory_invariant_mono`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Keeping (1) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `linear_plain_unchanged_memory_invariant.2.1.3.1.2.1.1.1.1.`

linear_plain_unchanged_memory_invariant.2.1.3.1.2.1.1.1.2:

{-1}	is_linear_plain_memory?(pm')
{-2}	(address_block(a', length(bl')) \subseteq (pm'ro_addr \cup pm'rw_addr))
{-3}	every(λ (t: [Address, list[Byte]]): Mem?(t'1'type_of) \wedge 0 \leq t'1'offset \wedge t'1'offset < max_linear_offset) (split(min_page, a', bl'))
{-4}	Mem?(type_of(a'))
{-5}	(extend [Address, Memory_Address_4G, bool, FALSE](virt_to_phys_range(s', restrict [Address, Memory_
{-6}	plain_memory?(pm_phy)
{-7}	pm'states = pm_phy'states
{-8}	reg_base(min_linear)(type_of(a')) \leq a'
{-9}	a' + length(bl') \leq (#type_of := type_of(a'), offset := max_linear_offset#)
{-10}	pm'states(s')
{1}	0 \leq a'offset \wedge a'offset < max_linear_offset
{2}	every(λ (e: [Memory_Address_4G, list[Byte]]): unchanged_memory_invariant?(pm_phy'mem, pm_phy'states, singleton(expr_2_super(linear_read_side_effect_in_page (e))), extend[Address, Mem- ory_Address_4G, bool, FALSE] (restrict[Address, Mem- ory_Address_4G, boolean] ((pm_phy'ro_addr \cup pm_phy'rw_addr)))) (split(min_page, a', bl'))
{3}	null?(bl')

Keeping (-8 -9 1 3) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of linear_plain_unchanged_memory_invariant.2.1.3.1.2.1.1.1.2.

linear_plain_unchanged_memory_invariant.2.1.3.1.2.1.1.2:

{-1}	unchanged_memory_invariant?(pm_phy'mem, pm_phy'states, singleton(expr_2_super(apply_side_effects(split ear_read_side_effect_in_page))), extend[Address, Memory_Address_4G, bool, FALSE] (restrict[Address, Mem- ory_Address_4G, boolean] ((pm_phy'ro_addr ∪ pm_phy'rw_addr))))	(min_pag lin-
{-2}	is_linear_plain_memory?(pm')	
{-3}	(address_block(a', length(bl')) ⊆ (pm'ro_addr ∪ pm'rw_addr))	
{-4}	every(λ (t: [Address, list[Byte]]): Mem?(t'1'type_of) ∧ 0 ≤ t'1'offset ∧ t'1'offset < max_linear_offset) (split(min_page, a', bl'))	
{-5}	Mem?(type_of(a'))	
{-6}	(extend [Address, Memory_Address_4G, bool, FALSE](virt_to_phys_range(s', restrict [Address,	
{-7}	plain_memory?(pm_phy)	
{-8}	unchanged_memory_invariant?(pm_phy'mem, pm_phy'states, ((pm_phy'other_actions ∪ memory_read_transformers(pm_phy'm (pm_phy'ro_addr ∪ pm_phy'rw_addr))	
{-9}	pm'states = pm_phy'states	
{-10}	reg_base(min_linear)(type_of(a')) ≤ a'	
{-11}	a' + length(bl') ≤ (#type_of := type_of(a'), offset := max_linear_offset#)	
{-12}	pm'states(s')	
{1}	(extend [Address, Memory_Address_4G, bool, FALSE](virt_to_phys_range(s', restrict [Address,	
{2}	unchanged_memory_invariant?(pm_phy'mem, pm_phy'states, singleton(expr_2_super(λ (s: Linear_memory [Physical_memory, pm_ apply_side_effects(split (min_p a', bl'), lin- (s))), extend[Address, Memory_Address_4G, bool, FALSE] (virt_to_phys_range(s', restrict [Address, Mem- ory_Address_4G, boolean] ((pm'ro_addr ∪ pm'rw_ad (singleton(Read) ∪ singleton(Ex	
{3}	null?(bl')	

Keeping (-6 1) and hiding *,

Expanding the definition of subset?,

Repeatedly Skolemizing and flattening,

Instantiating the top quantifier in -2 with the terms: (x!1),

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of linear_plain_unchanged_memory_invariant.2.1.3.1.2.1.1.2.

linear_plain_unchanged_memory_invariant.2.1.3.1.2.1.1.3:

{-1}	unchanged_memory_invariant?(pm_phy'mem, pm_phy'states, singleton(expr_2_super(apply_side_effects(split ear_read_side_effect_in_page))), extend[Address, Memory_Address_4G, bool, FALSE] (restrict[Address, Mem- ory_Address_4G, boolean] ((pm_phy'ro_addr ∪ pm_phy'rw_addr))))	(min_page, a', bl') lin-
{-2}	is_linear_plain_memory?(pm')	
{-3}	(address_block(a', length(bl')) ⊆ (pm'ro_addr ∪ pm'rw_addr))	
{-4}	every(λ (t: [Address, list[Byte]]): Mem?(t'1'type_of) ∧ 0 ≤ t'1'offset ∧ t'1'offset < max_linear_offset) (split(min_page, a', bl'))	
{-5}	Mem?(type_of(a'))	
{-6}	(extend [Address, Memory_Address_4G, bool, FALSE](virt_to_phys_range(s', restrict [Address, Memory_	
{-7}	plain_memory?(pm_phy)	
{-8}	unchanged_memory_invariant?(pm_phy'mem, pm_phy'states, (pm_phy'other_actions ∪ memory_read_transformers(pm_phy'mem, (pm_	
{-9}	pm'states = pm_phy'states	
{-10}	reg_base(min_linear)(type_of(a')) ≤ a'	
{-11}	a' + length(bl') ≤ (#type_of := type_of(a'), offset := max_linear_offset#)	
{-12}	pm'states(s')	
{1}	(singleton(expr_2_super(λ (s: Linear_memory[Physical_memory, pm_phy]): apply_side_effects(split(min_p	
{2}	unchanged_memory_invariant?(pm_phy'mem, pm_phy'states, singleton(expr_2_super(λ (s: Linear_memory [Physical_memory, pm_phy]): apply_side_effects(split (min_page, a', bl'), lin- (s))), extend[Address, Memory_Address_4G, bool, FALSE] (virt_to_phys_range(s', restrict [Address, Mem- ory_Address_4G, boolean] ((pm'ro_addr ∪ pm'rw_addr)), (singleton(Read) ∪ singleton(Execute))))))	
{3}	null?(bl')	

Keeping (1) and hiding *,

Expanding the definition of expr_2_super,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of linear_plain_unchanged_memory_invariant.2.1.3.1.2.1.1.3.

C Proof scripts

linear_plain_unchanged_memory_invariant.2.1.3.1.2.1.2:

{-1}	Mem?(type_of(a'))	
{-2}	(extend [Address, Memory_Address_4G, bool, FALSE](virt_to_phys_range(s', restrict [Address,	
{-3}	plain_memory?(pm_phy)	
{-4}	unchanged_memory_invariant?(pm_phy'mem, pm_phy'states,	
	((pm_phy'other_actions ∪ memory_read_transformers(pm_phy'mem,	
	(pm_phy'ro_addr ∪ pm_phy'rw_addr))	
{-5}	is_linear_plain_memory?(pm')	
{-6}	pm'states = pm_phy'states	
{-7}	(address_block(a', length(bl')) ⊆ (pm'ro_addr ∪ pm'rw_addr))	
{-8}	reg_base(min_linear)(type_of(a')) ≤ a'	
{-9}	a' + length(bl) ≤ (#type_of := type_of(a'), offset := max_linear_offset#)	
{-10}	pm'states(s')	
{1}	0 ≤ a'offset ∧ a'offset < max_linear_offset	
{2}	null?(bl')	
{3}	unchanged_memory_invariant?(pm_phy'mem, pm_phy'states,	
	singleton(expr_2_super(λ (s:	
		Linear_memory
		[Physical_memory, pm_
		apply_side_effects(split
		(min_p
		a',
		bl'),
		lin-
	ear_read_side_effect_in_page)	(s))),
		extend[Address, Memory_Address_4G, bool, FALSE]
		(virt_to_phys_range(s',
		restrict
		[Address, Mem-
	ory_Address_4G, boolean]	((pm'ro_addr ∪ pm'rw_ad
		(singleton(Read) ∪ singleton(Ex

Keeping (-8 -9 1 2) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of linear_plain_unchanged_memory_invariant.2.1.3.1.2.1.2.

linear_plain_unchanged_memory_invariant.2.1.3.1.2.2:

<pre> {-1} Mem?(type_of(a')) {-2} (extend [Address, Memory_Address_4G, bool, FALSE](virt_to_phys_range(s', restrict [Address, Memory_A {-3} plain_memory?(pm_phy) {-4} unchanged_memory_invariant?(pm_phy'mem, pm_phy'states, ((pm_phy'other_actions ∪ memory_read_transformers(pm_phy'mem, (pm_p (pm_phy'ro_addr ∪ pm_phy'rw_addr)) {-5} is_linear_plain_memory?(pm') {-6} pm'states = pm_phy'states {-7} (address_block(a', length(bl')) ⊆ (pm'ro_addr ∪ pm'rw_addr)) {-8} pm'states(s') </pre>	<pre> {1} null?(bl') {2} a' + length(bl') ≤ (#type_of := type_of(a'), offset := max_linear_offset#) {3} unchanged_memory_invariant?(pm_phy'mem, pm_phy'states, singleton(expr_2_super(λ (s: Linear_memory [Physical_memory, pm_phy]): Fatal)), extend[Address, Memory_Address_4G, bool, FALSE] (virt_to_phys_range(s', restrict [Address, Mem- ory_Address_4G, boolean] ((pm'ro_addr ∪ pm'rw_addr)), (singleton(Read) ∪ singleton(Execute)))))) </pre>
---	--

Keeping (3) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of linear_plain_unchanged_memory_invariant.2.1.3.1.2.2.

C Proof scripts

linear_plain_unchanged_memory_invariant.2.1.3.1.2.3:

<pre> {-1} Mem?(type_of(a')) {-2} (extend [Address, Memory_Address_4G, bool, FALSE](virt_to_phys_range(s', restrict [Address, {-3} plain_memory?(pm_phy) {-4} unchanged_memory_invariant?(pm_phy'mem, pm_phy'states, ((pm_phy'other_actions ∪ memory_read_transformers(pm_phy'mem, (pm_phy'ro_addr ∪ pm_phy'rw_addr)) {-5} is_linear_plain_memory?(pm') {-6} pm'states = pm_phy'states {-7} (address_block(a', length(bl')) ⊆ (pm'ro_addr ∪ pm'rw_addr)) {-8} pm'states(s') </pre>	<pre> Linear_memory [Physical_memory, pm_p Fatal)), extend[Address, Memory_Address_4G, bool, FALSE] (virt_to_phys_range(s', restrict [Address, Mem- ory_Address_4G, boolean] ((pm'ro_addr ∪ pm'rw_addr (singleton(Read) ∪ singleton(Exec </pre>
--	---

Using lemma pm_memory_addr,

Simplifying, rewriting, and recording with decision procedures,

Keeping (-7 1 2) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of linear_plain_unchanged_memory_invariant.2.1.3.1.2.3.

linear_plain_unchanged_memory_invariant.2.1.3.2:

<pre> {-1} (extend [Address, Memory_Address_4G, bool, FALSE](virt_to_phys_range(s', restrict [Address, Memory_A {-2} plain_memory?(pm_phy) {-3} unchanged_memory_invariant?(pm_phy' mem, pm_phy' states, ((pm_phy'other_actions ∪ memory_read_transformers(pm_phy'mem, (pm_p (pm_phy'ro_addr ∪ pm_phy'rw_addr)) {-4} is_linear_plain_memory?(pm') {-5} pm' states = pm_phy' states {-6} (address_block(a', length(bl')) ⊆ (pm'ro_addr ∪ pm'rw_addr)) {-7} pm' states(s') </pre>	<pre> {1} Mem?(type_of(a')) {2} unchanged_memory_invariant?(pm_phy' mem, pm_phy' states, singleton(expr_2_super(λ (s: Linear_memory [Physical_memory, pm_phy]): IF null?(bl') ∨ (reg_base (min_linear)(type_of(a')) ≤ a' ∧ a' + length(bl') ≤ reg_size (max_linear)(type_of(a'))) THEN IF Mem?(type_of(a')) THEN ap- (split(min_page, a', bl'), lin- (s) ELSE mem- (pm_phy' mem) (a', bl', cp')(s) ENDIF ELSE Fatal ENDIF)), extend[Address, Memory_Address_4G, bool, FALSE] (virt_to_phys_range(s', restrict [Address, Mem- ory_Address_4G, boolean] ((pm'ro_addr ∪ pm'rw_addr)), (singleton(Read) ∪ singleton(Execute)))))) </pre>
---	--

Using lemma unchanged_memory_invariant_mono,

Case splitting on subset?(singleton(expr_2_super(LAMBDA (s: Linear_memory [Physical_memory, pm_phy]): memory_read_side_effect(pm_phy'mem) (a!1, bl!1, cp!1)(s))), union(union(pm_phy'other_actions, memory_read_transformers(pm_phy'mem, union (pm_phy'ro_addr, pm_phy'rw_addr))), union(memory_read_side_effect_super.L union (pm_phy'ro_addr, pm_phy'rw_addr)), memory_write_side_effect_super_transformers(pm_phy'mem,

C Proof scripts

`pm_phy'rw_addr))))),`

we get 2 subgoals:

1442

linear_plain_unchanged_memory_invariant.2.1.3.2.1:

```

{-1} (singleton(expr_2_super( $\lambda$  ( $s$  : Linear_memory [Physical_memory, pm_phy]) : memory_read_side_effect(pm_phy'
{-2} (singleton(expr_2_super( $\lambda$  ( $s$  : Linear_memory [Physical_memory, pm_phy]) : memory_read_side_effect(pm_phy'
    ^
    (extend [Address, Memory_Address_4G, bool, FALSE](virt_to_phys_range( $s'$ , restrict [Address, Memory_Address_4G, boolean]
    ^
    unchanged_memory_invariant?(pm_phy' mem, pm_phy' states,
        ((pm_phy' other_actions  $\cup$  memory_read_transformers(pm_phy' mem, (pm_phy' ro_addr  $\cup$  pm_phy' rw_addr))
     $\supset$ 
    unchanged_memory_invariant?(pm_phy' mem, pm_phy' states,
        singleton(expr_2_super( $\lambda$  ( $s$  :
            Linear_memory
            [Physical_memory, pm_phy]) :
            mem-
            ory_read_side_effect(pm_phy' mem)
            ( $a'$ ,  $bl'$ ,  $cp'$ )( $s$ )),
            extend[Address, Memory_Address_4G, bool, FALSE]
            (virt_to_phys_range( $s'$ ,
            restrict
            [Address, Mem-
            ory_Address_4G, boolean]
            ((pm' ro_addr  $\cup$  pm' rw_addr)),
            (singleton(Read)  $\cup$  singleton(Execute))))))
{-3} (extend [Address, Memory_Address_4G, bool, FALSE](virt_to_phys_range( $s'$ , restrict [Address, Memory_Address_4G, boolean]
{-4} plain_memory?(pm_phy)
{-5} unchanged_memory_invariant?(pm_phy' mem, pm_phy' states,
    ((pm_phy' other_actions  $\cup$  memory_read_transformers(pm_phy' mem, (pm_phy' ro_addr  $\cup$  pm_phy' rw_addr))
{-6} is_linear_plain_memory?(pm')
{-7} pm' states = pm_phy' states
{-8} (address_block( $a'$ , length( $bl'$ ))  $\subseteq$  (pm' ro_addr  $\cup$  pm' rw_addr))
{-9} pm' states( $s'$ )
-----
{1} Mem?(type_of( $a'$ ))
{2} unchanged_memory_invariant?(pm_phy' mem, pm_phy' states,
    singleton(expr_2_super( $\lambda$  ( $s$  :
        Linear_memory
        [Physical_memory, pm_phy]) :
        IF null?( $bl'$ )  $\vee$ 
        (reg_base
        (min_linear)(type_of( $a'$ ))
         $\leq$ 
         $a'$ 
         $\wedge$ 
         $a' + \text{length}(bl')$ 
         $\leq$ 
        reg_size
        (max_linear)(type_of( $a'$ )))
        THEN IF Mem?(type_of( $a'$ ))
        THEN ap-
        ply_side_effects
        (split(min_page,  $a'$ ,  $bl'$ ),
        lin-
        ear_read_side_effect_in_page)
        ( $s$ )
        ELSE mem-
        ory_read_side_effect
        (pm_phy' mem)
        ( $a'$ ,  $bl'$ ,  $cp'$ )( $s$ )
        ENDIF
        ELSE Fatal
        ENDIF)),
    extend[Address, Memory_Address_4G, bool, FALSE]
    (virt_to_phys_range( $s'$ ,

```

C Proof scripts

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 2 subgoals:

`linear_plain_unchanged_memory_invariant.2.1.3.2.1.1:`

{-1}	(singleton(expr_2_super(λ (s : Linear_memory[Physical_memory, pm_phy]) : memory_read_side_	
{-2}	(extend [Address, Memory_Address_4G, bool, FALSE](virt_to_phys_range(s' , restrict [Address,	
{-3}	unchanged_memory_invariant?(pm_phy'mem, pm_phy'states,	
	((pm_phy'other_actions \cup memory_read_transformers(pm_phy'mem	
	(pm_phy'ro_addr \cup pm_phy'rw_addr))	
{-4}	unchanged_memory_invariant?(pm_phy'mem, pm_phy'states,	
	singleton(expr_2_super(λ (s :	
	Linear_memory	[Physical_memory, pm_p
	mem-	mem-
	ory_read_side_effect(pm_phy'mem)	(a', bl', cp')(s))),
	extend[Address, Memory_Address_4G, bool, FALSE]	(virt_to_phys_range(s' ,
	restrict	[Address, Mem-
	ory_Address_4G, boolean]	((pm'ro_addr \cup pm'rw_addr
	(singleton(Read) \cup singleton(Exec	
{-5}	plain_memory?(pm_phy)	
{-6}	is_linear_plain_memory?(pm')	
{-7}	pm'states = pm_phy'states	
{-8}	(address_block(a', length(bl')) \subseteq (pm'ro_addr \cup pm'rw_addr))	
{-9}	pm'states(s')	
{1}	Mem?(type_of(a'))	
{2}	null?(bl')	
{3}	a' + length(bl') \leq	
	reg_size((#type_of := Mem_, offset := max_linear_offset#))(type_of(a'))	
{4}	unchanged_memory_invariant?(pm_phy'mem, pm_phy'states,	
	singleton(expr_2_super(λ (s :	
	Linear_memory	[Physical_memory, pm_p
	Fatal)),	Fatal)),
	extend[Address, Memory_Address_4G, bool, FALSE]	(virt_to_phys_range(s' ,
	restrict	[Address, Mem-
	ory_Address_4G, boolean]	((pm'ro_addr \cup pm'rw_addr
	(singleton(Read) \cup singleton(Exec	

Keeping (4) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `linear_plain_unchanged_memory_invariant.2.1.3.2.1.1`.

linear_plain_unchanged_memory_invariant.2.1.3.2.1.2:

{-1}	(singleton(expr_2_super(λ (s : Linear_memory[Physical_memory, pm_phy]) : memory_read_side_effect(pm_	
{-2}	extend [Address, Memory_Address_4G, bool, FALSE](virt_to_phys_range(s' , restrict [Address, Memory_A	
{-3}	unchanged_memory_invariant?(pm_phy'mem, pm_phy'states, ((pm_phy'other_actions \cup memory_read_transformers(pm_phy'mem, (pm_ph	
{-4}	unchanged_memory_invariant?(pm_phy'mem, pm_phy'states, singleton(expr_2_super(λ (s :	Linear_memory [Physical_memory, pm_phy]): mem-
	ory_read_side_effect(pm_phy'mem)	(a', bl', cp')(s)),
		extend[Address, Memory_Address_4G, bool, FALSE] (virt_to_phys_range(s' ,
		restrict [Address, Mem-
	ory_Address_4G, boolean]	((pm'ro_addr \cup pm'rw_addr)), (singleton(Read) \cup singleton(Execute))))
{-5}	plain_memory?(pm_phy)	
{-6}	is_linear_plain_memory?(pm')	
{-7}	pm'states = pm_phy'states	
{-8}	(address_block(a' , length(bl')) \subseteq (pm'ro_addr \cup pm'rw_addr))	
{-9}	pm'states(s')	
{1}	Mem?(type_of(a'))	
{2}	null?(bl')	
{3}	reg_base(min_linear)(type_of(a')) \leq a'	
{4}	unchanged_memory_invariant?(pm_phy'mem, pm_phy'states, singleton(expr_2_super(λ (s :	Linear_memory [Physical_memory, pm_phy]): Fatal)),
		extend[Address, Memory_Address_4G, bool, FALSE] (virt_to_phys_range(s' ,
		restrict [Address, Mem-
	ory_Address_4G, boolean]	((pm'ro_addr \cup pm'rw_addr)), (singleton(Read) \cup singleton(Execute))))

Using lemma pm_memory_addr,

Simplifying, rewriting, and recording with decision procedures,

Keeping (-8 1 3) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of linear_plain_unchanged_memory_invariant.2.1.3.2.1.2.

linear_plain_unchanged_memory_invariant.2.1.3.2.2:

```

{-1} (singleton(expr_2_super( $\lambda$  ( $s$  : Linear_memory[Physical_memory, pm_phy]) : memory_read_side_
    ^
    (extend [Address, Memory_Address_4G, bool, FALSE](virt_to_phys_range( $s'$ , restrict [Address,
    ^
    unchanged_memory_invariant?(pm_phy'mem, pm_phy'states,
        ((pm_phy'other_actions  $\cup$  memory_read_transformers(pm_phy'
        (pm_phy'ro_addr  $\cup$  pm_phy'rw_addr))
     $\supset$ 
    unchanged_memory_invariant?(pm_phy'mem, pm_phy'states,
        singleton(expr_2_super( $\lambda$  ( $s$  :
            Linear_memory
            [Physical_memory, pm_ph
            mem-
    ory_read_side_effect(pm_phy'mem)
            ( $a'$ ,  $bl'$ ,  $cp'$ )( $s$ ))),
        extend [Address, Memory_Address_4G, bool, FALSE]
        (virt_to_phys_range( $s'$ ,
            restrict
            [Address, Mem-
    ory_Address_4G, boolean]
            ((pm'ro_addr  $\cup$  pm'rw_ad
            (singleton(Read)  $\cup$  singleton(Ex
{-2} (extend [Address, Memory_Address_4G, bool, FALSE](virt_to_phys_range( $s'$ , restrict [Address,
{-3} plain_memory?(pm_phy)
{-4} unchanged_memory_invariant?(pm_phy'mem, pm_phy'states,
        ((pm_phy'other_actions  $\cup$  memory_read_transformers(pm_phy'me
        (pm_phy'ro_addr  $\cup$  pm_phy'rw_addr))
{-5} is_linear_plain_memory?(pm')
{-6} pm'states = pm_phy'states
{-7} (address_block( $a'$ , length( $bl'$ ))  $\subseteq$  (pm'ro_addr  $\cup$  pm'rw_addr))
{-8} pm'states( $s'$ )
-----
{1} (singleton(expr_2_super( $\lambda$  ( $s$  : Linear_memory[Physical_memory, pm_phy]) : memory_read_side_
{2} Mem?(type_of( $a'$ ))
{3} unchanged_memory_invariant?(pm_phy'mem, pm_phy'states,
        singleton(expr_2_super( $\lambda$  ( $s$  :
            Linear_memory
            [Physical_memory, pm-p
            IF null?( $bl'$ )  $\vee$ 
            (reg_base
            (min_linear)(type_of( $a'$ ))
             $\leq$ 
             $a'$ 
             $\wedge$ 
             $a' + \text{length}(bl')$ 
             $\leq$ 
            reg_size
            (max_linear)(type_of( $a'$ ))
            THEN IF Mem?(type_of( $a'$ ))
            THEN ap-
    ply_side_effects
            (split(min_page,
            lin-
    ear_read_side_effect_in_page)
            ( $s$ )
            ELSE mem-
    ory_read_side_effect
            (pm_phy'mem)
            ( $a'$ ,  $bl'$ ,  $cp'$ )( $s$ )
            ENDIF
            ELSE Fatal
            ENDIF)),
        extend [Address, Memory_Address_4G, bool, FALSE]
        (virt_to_phys_range( $s'$ ,

```

Hiding formulas: (-1 -4 3),

Case splitting on `memory_read_side_effect_super_transformers(pm_phy'mem, union (pm_phy'ro_addr, pm_phy'rw_addr)) (expr_2_super (LAMBDA (s: Linear_memory [Physical_memory, pm_phy]): memory_read_side_effect (pm_phy'mem) (a!1, bl!1, cp!1)(s)))`,

we get 2 subgoals:

`linear_plain_unchanged_memory_invariant.2.1.3.2.2.1:`

{-1}	<code>memory_read_side_effect_super_transformers(pm_phy'mem, (pm_phy'ro_addr ∪ pm_phy'rw_addr))</code> <code>(expr_2_super(λ (s: Linear_memory [Physical_memory, pm_phy]): mem- <code>ory_read_side_effect(pm_phy'mem)</code> <code>(a', bl', cp')(s)))</code> </code>
{-2}	<code>(extend [Address, Memory_Address_4G, bool, FALSE](virt_to_phys_range(s', restrict [Address, Memory_A</code>
{-3}	<code>plain_memory?(pm_phy)</code>
{-4}	<code>is_linear_plain_memory?(pm')</code>
{-5}	<code>pm' states = pm_phy states</code>
{-6}	<code>(address_block(a', length(bl')) ⊆ (pm'ro_addr ∪ pm'rw_addr))</code>
{-7}	<code>pm' states(s')</code>
{1}	<code>(singleton(expr_2_super(λ (s: Linear_memory [Physical_memory, pm_phy]): memory_read_side_effect(pm_p</code>
{2}	<code>Mem?(type_of(a'))</code>

Keeping (-1 1) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `linear_plain_unchanged_memory_invariant.2.1.3.2.2.1`.

`linear_plain_unchanged_memory_invariant.2.1.3.2.2.2:`

{-1}	<code>(extend [Address, Memory_Address_4G, bool, FALSE](virt_to_phys_range(s', restrict [Address, Memory_A</code>
{-2}	<code>plain_memory?(pm_phy)</code>
{-3}	<code>is_linear_plain_memory?(pm')</code>
{-4}	<code>pm' states = pm_phy states</code>
{-5}	<code>(address_block(a', length(bl')) ⊆ (pm'ro_addr ∪ pm'rw_addr))</code>
{-6}	<code>pm' states(s')</code>
{1}	<code>memory_read_side_effect_super_transformers(pm_phy'mem, (pm_phy'ro_addr ∪ pm_phy'rw_addr))</code> <code>(expr_2_super(λ (s: Linear_memory [Physical_memory, pm_phy]): mem- <code>ory_read_side_effect(pm_phy'mem)</code> <code>(a', bl', cp')(s)))</code> </code>
{2}	<code>(singleton(expr_2_super(λ (s: Linear_memory [Physical_memory, pm_phy]): memory_read_side_effect(pm_p</code>
{3}	<code>Mem?(type_of(a'))</code>

Hiding formulas: 2,

Expanding the definition of `memory_read_side_effect_super_transformers`,

Instantiating quantified variables,

Expanding the definition of `expr_2_super`,

Expanding the definition of `subset?`,

Repeatedly Skolemizing and flattening,

Instantiating quantified variables,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Expanding the definition of `member`,

Expanding the definition of `address_block`,

C Proof scripts

Using lemma `pm_memory_addr`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `linear_plain_unchanged_memory_invariant.2.1.3.2.2.2`.

`linear_plain_unchanged_memory_invariant.2.1.4`:

<pre> {-1} (extend [Address, Memory_Address_4G, bool, FALSE](virt_to_phys_range(s', restrict [Address, {-2} plain_memory?(pm_phy) {-3} unchanged_memory_invariant?(pm_phy'mem, pm_phy'states, ((pm_phy'other_actions ∪ memory_read_transformers(pm_phy'mem, (pm_phy'ro_addr ∪ pm_phy'rw_addr)) {-4} is_linear_plain_memory?(pm') {-5} pm'states = pm_phy'states {-6} memory_write_side_effect_super_transformers(pm'mem, pm'rw_addr)(q') {-7} pm'states(s') </pre>	<pre> singleton(q'), extend[Address, Memory_Address_4G, bool, FALSE] (virt_to_phys_range(s', restrict [Address, Mem- ory_Address_4G, boolean] ((pm'ro_addr ∪ pm'rw_addr) (singleton(Read) ∪ singleton(Exec </pre>
--	---

Expanding the definition of `memory_write_side_effect_super_transformers`,

Repeatedly Skolemizing and flattening,

Replacing using formula -8,

Hiding formulas: (-1 -8),

Rewriting using `pm_write_side_effect_linear`, matching in *,

Expanding the definition of `linear_write_side_effect`,

Installing automatic rewrites from: `union_right`

Case splitting on `Mem?(type_of(a!1))`,

we get 2 subgoals:

linear_plain_unchanged_memory_invariant.2.1.4.1:

<pre> {-1} Mem?(type_of(a')) {-2} (extend [Address, Memory_Address_4G, bool, FALSE](virt_to_phys_range(s', restrict [Address, Memory_A {-3} plain_memory?(pm_phy) {-4} unchanged_memory_invariant?(pm_phy'mem, pm_phy'states, ((pm_phy'other_actions ∪ memory_read_transformers(pm_phy'mem, (pm_p (pm_phy'ro_addr ∪ pm_phy'rw_addr)) {-5} is_linear_plain_memory?(pm') {-6} pm'states = pm_phy'states {-7} (address_block(a', length(bl')) ⊆ pm'rw_addr) {-8} pm'states(s') </pre>	<pre> Linear_memory [Physical_memory, pm_phy]): IF null?(bl') ∨ (reg_base (min_linear)(type_of(a')) ≤ a' ∧ a' + length(bl') ≤ reg_size (max_linear)(type_of(a'))) THEN IF Mem?(type_of(a')) THEN ap- (split(min_page, a', bl'), lin- (s) ELSE mem- (pm_phy'mem) (a', bl', cp')(s) ENDIF ELSE Fatal ENDIF), extend[Address, Memory_Address_4G, bool, FALSE] (virt_to_phys_range(s', restrict [Address, Mem- ((pm'ro_addr ∪ pm'rw_addr)), (singleton(Read) ∪ singleton(Execute)))] </pre>
--	---

Case splitting on null?(bl1),

we get 2 subgoals:

linear_plain_unchanged_memory_invariant.2.1.4.1.1:

<pre> {-1} null?(bl') {-2} Mem?(type_of(a')) {-3} (extend [Address, Memory_Address_4G, bool, FALSE](virt_to_phys_range(s', restrict [Address, {-4} plain_memory?(pm_phy) {-5} unchanged_memory_invariant?(pm_phy'mem, pm_phy'states, ((pm_phy'other_actions ∪ memory_read_transformers(pm_phy'mem) (pm_phy'ro_addr ∪ pm_phy'rw_addr)) {-6} is_linear_plain_memory?(pm') {-7} pm'states = pm_phy'states {-8} (address_block(a', length(bl')) ⊆ pm'rw_addr) {-9} pm'states(s') </pre>	<pre> Linear_memory [Physical_memory, pm_p IF null?(bl') ∨ (reg_base (min_linear)(type_of(a')) ≤ a' ∧ a' + length(bl') ≤ reg_size (max_linear)(type_of(a')) THEN IF Mem?(type_of(a')) THEN ap- (split(min_page, lin- (s) ELSE mem- (pm_phy'mem) (a', bl', cp')(s) ENDIF ELSE Fatal ENDIF)), extend[Address, Memory_Address_4G, bool, FALSE] (virt_to_phys_range(s', restrict [Address, Mem- ((pm'ro_addr ∪ pm'rw_addr) (singleton(Read) ∪ singleton(Exec </pre>
---	---

Simplifying, rewriting, and recording with decision procedures,
Expanding the definition of apply_side_effects,
Expanding the definition of ok_result,
Expanding the definition of reduce,

Using lemma split_null,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Keeping (1) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of linear_plain_unchanged_memory_invariant.2.1.4.1.1.

linear_plain_unchanged_memory_invariant.2.1.4.1.2:

<pre> {-1} Mem?(type_of(a')) {-2} (extend [Address, Memory_Address_4G, bool, FALSE](virt_to_phys_range(s', restrict [Address, Memory_A {-3} plain_memory?(pm_phy) {-4} unchanged_memory_invariant?(pm_phy'mem, pm_phy'states, ((pm_phy'other_actions ∪ memory_read_transformers(pm_phy'mem, (pm_p (pm_phy'ro_addr ∪ pm_phy'rw_addr)) {-5} is_linear_plain_memory?(pm') {-6} pm'states = pm_phy'states {-7} (address_block(a', length(bl')) ⊆ pm'rw_addr) {-8} pm'states(s') </pre>	<pre> {1} null?(bl') {2} unchanged_memory_invariant?(pm_phy'mem, pm_phy'states, singleton(expr_2_super(λ (s: Linear_memory [Physical_memory, pm_phy]): IF null?(bl') ∨ (reg_base (min_linear)(type_of(a')) ≤ a' ∧ a' + length(bl') ≤ reg_size (max_linear)(type_of(a'))) THEN IF Mem?(type_of(a')) THEN ap- (split(min_page, a', bl'), lin- (s) ELSE mem- (pm_phy'mem) (a', bl', cp')(s) ENDIF ELSE Fatal ENDIF)), extend[Address, Memory_Address_4G, bool, FALSE] (virt_to_phys_range(s', restrict [Address, Mem- (pm'ro_addr ∪ pm'rw_addr)), (singleton(Read) ∪ singleton(Execute)))))) </pre>
--	--

C Proof scripts

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 3 subgoals:

`linear_plain_unchanged_memory_invariant.2.1.4.1.2.1:`

<pre> {-1} Mem?(type_of(a')) {-2} (extend [Address, Memory_Address_4G, bool, FALSE](virt_to_phys_range(s', restrict [Address, Mem- {-3} plain_memory?(pm_phy) {-4} unchanged_memory_invariant?(pm_phy'mem, pm_phy'states, ((pm_phy'other_actions ∪ memory_read_transformers(pm_phy'mem) (pm_phy'ro_addr ∪ pm_phy'rw_addr)) {-5} is_linear_plain_memory?(pm') {-6} pm'states = pm_phy'states {-7} (address_block(a', length(bl')) ⊆ pm'rw_addr) {-8} reg_base(min_linear)(type_of(a')) ≤ a' {-9} a' + length(bl') ≤ (#type_of := type_of(a'), offset := max_linear_offset#) {-10} pm'states(s') </pre>	<pre> {1} null?(bl') {2} unchanged_memory_invariant?(pm_phy'mem, pm_phy'states, singleton(expr_2_super(λ (s: Linear_memory [Physical_memory, pm_ apply_side_effects(split (min_p a', bl'), lin- (s))), extend[Address, Memory_Address_4G, bool, FALSE] (virt_to_phys_range(s', restrict [Address, Mem- (pm'ro_addr ∪ pm'rw_ad (singleton(Read) ∪ singleton(Ex </pre>
---	--

Using lemma `address_block_split_type`,

we get 2 subgoals:

linear_plain_unchanged_memory_invariant.2.1.4.1.2.1.1:

<pre> {-1} is_linear_plain_memory?(pm') ∧ (address_block(a', length(bl')) ⊆ (pm'ro_addr ∪ pm'rw_addr)) ⊃ every(λ (t: [Address, list[Byte]]): Mem?(t'1'type_of) ∧ 0 ≤ t'1'offset ∧ t'1'offset < max_linear_offset) (split(min_page, a', bl')) {-2} Mem?(type_of(a')) {-3} (extend [Address, Memory_Address_4G, bool, FALSE](virt_to_phys_range(s', restrict [Address, Memory_ {-4} plain_memory?(pm_phy) {-5} unchanged_memory_invariant?(pm_phy'mem, pm_phy'states, ((pm_phy'other_actions ∪ memory_read_transformers(pm_phy'mem, (pm_p (pm_phy'ro_addr ∪ pm_phy'rw_addr)) {-6} is_linear_plain_memory?(pm') {-7} pm'states = pm_phy'states {-8} (address_block(a', length(bl')) ⊆ pm'rw_addr) {-9} reg_base(min_linear)(type_of(a')) ≤ a' {-10} a' + length(bl') ≤ (#type_of := type_of(a'), offset := max_linear_offset#) {-11} pm'states(s') </pre>	<pre> {1} null?(bl') {2} unchanged_memory_invariant?(pm_phy'mem, pm_phy'states, singleton(expr_2_super(λ (s: Linear_memory [Physical_memory, pm_phy]): apply_side_effects(split (min_page, a', bl'), lin- (s))), extend[Address, Memory_Address_4G, bool, FALSE] (virt_to_phys_range(s', restrict [Address, Mem- ory_Address_4G, boolean] ((pm'ro_addr ∪ pm'rw_addr)), (singleton(Read) ∪ singleton(Execute)))))) </pre>
--	--

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 2 subgoals:

linear_plain_unchanged_memory_invariant.2.1.4.1.2.1.1.1:

<pre> {-1} is_linear_plain_memory?(pm') {-2} every(λ (t: [Address, list[Byte]]): Mem?(t'1'type_of) ∧ 0 ≤ t'1'offset ∧ t'1'offset < max_linear_offset) (split(min_page, a', bl')) {-3} Mem?(type_of(a')) {-4} (extend [Address, Memory_Address_4G, bool, FALSE](virt_to_phys_range(s', restrict [Address, {-5} plain_memory?(pm_phy) {-6} unchanged_memory_invariant?(pm_phy'mem, pm_phy'states, ((pm_phy'other_actions ∪ memory_read_transformers(pm_phy'mem, (pm_phy'ro_addr ∪ pm_phy'rw_addr)) {-7} pm'states = pm_phy'states {-8} (address_block(a', length(bl')) ⊆ pm'rw_addr) {-9} reg_base(min_linear)(type_of(a')) ≤ a' {-10} a' + length(bl') ≤ (#type_of := type_of(a'), offset := max_linear_offset#) {-11} pm'states(s') </pre>	<pre> {1} null?(bl') {2} unchanged_memory_invariant?(pm_phy'mem, pm_phy'states, singleton(expr_2_super(λ (s: Linear_memory [Physical_memory, pm_ apply_side_effects(split (min_p a', bl'), lin- (s))), extend[Address, Memory_Address_4G, bool, FALSE] (virt_to_phys_range(s', restrict [Address, Mem- ory_Address_4G, boolean] ((pm'ro_addr ∪ pm'rw_ad (singleton(Read) ∪ singleton(Ex </pre>
--	--

Using lemma `apply_side_effects_unchanged`,

Using lemma `unchanged_memory_invariant_mono`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 3 subgoals:

linear_plain_unchanged_memory_invariant.2.1.4.1.2.1.1.1.1:

{-1}	is_linear_plain_memory?(pm')	
{-2}	every(λ (t: [Address, list[Byte]]): Mem?(t'1'type_of) \wedge 0 \leq t'1'offset \wedge t'1'offset < max_linear_offset) (split(min_page, a', bl'))	
{-3}	Mem?(type_of(a'))	
{-4}	(extend [Address, Memory_Address_4G, bool, FALSE](virt_to_phys_range(s', restrict [Address, Memory_	
{-5}	plain_memory?(pm_phy)	
{-6}	unchanged_memory_invariant?(pm_phy'mem, pm_phy'states, (pm_phy'other_actions \cup memory_read_transformers(pm_phy'mem, (pm_phy'ro_addr \cup pm_phy'rw_addr))	
{-7}	pm' states = pm_phy'states	
{-8}	(address_block(a', length(bl')) \subseteq pm'rw_addr)	
{-9}	reg_base(min_linear)(type_of(a')) \leq a'	
{-10}	a' + length(bl') \leq (#type_of := type_of(a'), offset := max_linear_offset#)	
{-11}	pm' states(s')	
{1}	unchanged_memory_invariant?(pm_phy'mem, pm_phy'states, singleton(expr_2_super(apply_side_effects(split (min_page, a', bl') lin- ear_write_side_effect_in_page))), extend[Address, Memory_Address_4G, bool, FALSE] (restrict[Address, Mem- ory_Address_4G, boolean] (pm_phy'ro_addr \cup pm_phy'rw_addr))))	
{2}	unchanged_memory_invariant?(pm_phy'mem, pm_phy'states, singleton(expr_2_super(λ (s: Linear_memory [Physical_memory, pm_phy]): apply_side_effects(split (min_page, a', bl'), lin- ear_write_side_effect_in_page) (s))), extend[Address, Memory_Address_4G, bool, FALSE] (virt_to_phys_range(s', restrict [Address, Mem- ory_Address_4G, boolean] (pm'ro_addr \cup pm'rw_addr)), (singleton(Read) \cup singleton(Execute))))))	
{3}	every(λ (e: [Memory_Address_4G, list[Byte]]): unchanged_memory_invariant?(pm_phy'mem, pm_phy'states, singleton(expr_2_super(linear_write_side_effect_in_page (e))), extend[Address, Mem- ory_Address_4G, bool, FALSE] (restrict[Address, Mem- ory_Address_4G, boolean] (pm_phy'ro_addr \cup pm_phy'rw_addr))))	
{4}	(split(min_page, a', bl')) null?(bl')	

C Proof scripts

Hiding formulas: (-7 1 2),

Using lemma `split_linear_write_side_effects_unchanged_memory`,

we get 2 subgoals:

`linear_plain_unchanged_memory_invariant.2.1.4.1.2.1.1.1.1.1:`

{-1}	$\text{is_linear_plain_memory?}(pm') \wedge (\text{address_block}(a', \text{length}(bl')) \subseteq pm'\text{'rw_addr}) \supset$ $\text{every}(\lambda (e: [\text{Memory_Address_4G}, \text{list}[\text{Byte}]]):$ $\text{unchanged_memory_invariant?}(pm_phy'\text{'mem}, pm'\text{'states},$ $\text{singleton}(\text{expr_2_super}(\text{linear_write_side_effect_in_pa}$ $(e))),$ $(pm_phy'\text{'ro_addr} \cup pm_phy'\text{'rw_addr}))$ $(\text{split}(\text{min_page}, a', bl'))$
{-2}	$\text{is_linear_plain_memory?}(pm')$
{-3}	$\text{every}(\lambda (t: [\text{Address}, \text{list}[\text{Byte}]]):$ $\text{Mem?}(t'\text{'1' type_of}) \wedge 0 \leq t'\text{'1' offset} \wedge t'\text{'1' offset} < \text{max_linear_offset})$ $(\text{split}(\text{min_page}, a', bl'))$
{-4}	$\text{Mem?}(\text{type_of}(a'))$
{-5}	$(\text{extend} [\text{Address}, \text{Memory_Address_4G}, \text{bool}, \text{FALSE}](\text{virt_to_phys_range}(s', \text{restrict} [\text{Address},$
{-6}	$\text{plain_memory?}(pm_phy)$
{-7}	$\text{unchanged_memory_invariant?}(pm_phy'\text{'mem}, pm_phy'\text{'states},$ $((pm_phy'\text{'other_actions} \cup \text{memory_read_transformers}(pm_phy'\text{'m}$ $(pm_phy'\text{'ro_addr} \cup pm_phy'\text{'rw_addr}))$
{-8}	$(\text{address_block}(a', \text{length}(bl')) \subseteq pm'\text{'rw_addr})$
{-9}	$\text{reg_base}(\text{min_linear})(\text{type_of}(a')) \leq a'$
{-10}	$a' + \text{length}(bl') \leq (\#\text{type_of} := \text{type_of}(a'), \text{offset} := \text{max_linear_offset}\#)$
{-11}	$pm'\text{'states}(s')$
{1}	$\text{every}(\lambda (e: [\text{Memory_Address_4G}, \text{list}[\text{Byte}]]):$ $\text{unchanged_memory_invariant?}(pm_phy'\text{'mem}, pm_phy'\text{'states},$ $\text{singleton}(\text{expr_2_super}(\text{linear_write_side_effect_in_pa}$ $(e))),$ $\text{extend}[\text{Address}, \text{Mem-}$ $\text{ory_Address_4G}, \text{bool}, \text{FALSE}]$ $(\text{restrict}[\text{Address}, \text{Mem-}$ $\text{ory_Address_4G}, \text{boolean}]$ $((pm_phy'\text{'ro_addr} \cup pm_phy'\text{'rw_addr})))$ $(\text{split}(\text{min_page}, a', bl'))$
{2}	$\text{null?}(bl')$

Simplifying, rewriting, and recording with decision procedures,

Using lemma `every_implied`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Hiding formulas: (-1 2),

Repeatedly Skolemizing and flattening,

Using lemma `unchanged_memory_invariant_mono`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Keeping (1) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `linear_plain_unchanged_memory_invariant.2.1.4.1.2.1.1.1.1.1.`

linear_plain_unchanged_memory_invariant.2.1.4.1.2.1.1.1.1.2:

{-1}	is_linear_plain_memory?(pm')
{-2}	every(λ (t : [Address, list[Byte]]): $\text{Mem?}(t'1'\text{type_of}) \wedge 0 \leq t'1'\text{offset} \wedge t'1'\text{offset} < \text{max_linear_offset}$ $(\text{split}(\text{min_page}, a', \text{bl}'))$)
{-3}	Mem?(type_of(a'))
{-4}	(extend [Address, Memory_Address_4G, bool, FALSE](virt_to_phys_range(s', restrict [Address, Memory_
{-5}	plain_memory?(pm_phy
{-6}	unchanged_memory_invariant?(pm_phy'mem, pm_phy'states, $((\text{pm_phy'other_actions} \cup \text{memory_read_transformers}(\text{pm_phy'mem}, (\text{pm_p$ $(\text{pm_phy'ro_addr} \cup \text{pm_phy'rw_addr}))$)
{-7}	(address_block(a', length(bl')) \subseteq pm'rw_addr)
{-8}	reg_base(min_linear)(type_of(a')) $\leq a'$
{-9}	$a' + \text{length}(\text{bl}') \leq (\#\text{type_of} := \text{type_of}(a'), \text{offset} := \text{max_linear_offset}\#)$
{-10}	pm'states(s')
{1}	$0 \leq a'\text{offset} \wedge a'\text{offset} < \text{max_linear_offset}$
{2}	every(λ (e : [Memory_Address_4G, list[Byte]]): $\text{unchanged_memory_invariant?}(\text{pm_phy'mem}, \text{pm_phy'states},$ $\text{singleton}(\text{expr_2_super}(\text{linear_write_side_effect_in_page}$ $(e)),$ $\text{extend}[\text{Address}, \text{Mem-}$ $\text{ory_Address_4G}, \text{bool}, \text{FALSE}]$ $(\text{restrict}[\text{Address}, \text{Mem-}$ $\text{ory_Address_4G}, \text{boolean}]$ $((\text{pm_phy'ro_addr} \cup \text{pm_phy'rw_addr}))))$ $(\text{split}(\text{min_page}, a', \text{bl}'))$)
{3}	null?(bl')

Keeping (-8 -9 1 3) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of linear_plain_unchanged_memory_invariant.2.1.4.1.2.1.1.1.1.2.

linear_plain_unchanged_memory_invariant.2.1.4.1.2.1.1.1.2:

{-1}	unchanged_memory_invariant?(pm_phy'mem, pm_phy'states, singleton(expr_2_super(apply_side_effects(split ear_write_side_effect_in_page))), extend[Address, Memory_Address_4G, bool, FALSE] (restrict[Address, Mem- ory_Address_4G, boolean] ((pm_phy'ro_addr ∪ pm_phy'rw_addr))))	(min_pag lin-
{-2}	is_linear_plain_memory?(pm')	
{-3}	every(λ (t: [Address, list[Byte]]): Mem?(t'1'type_of) ∧ 0 ≤ t'1'offset ∧ t'1'offset < max_linear_offset) (split(min_page, a', bl'))	
{-4}	Mem?(type_of(a'))	
{-5}	(extend [Address, Memory_Address_4G, bool, FALSE](virt_to_phys_range(s', restrict [Address,	
{-6}	plain_memory?(pm_phy)	
{-7}	unchanged_memory_invariant?(pm_phy'mem, pm_phy'states, ((pm_phy'other_actions ∪ memory_read_transformers(pm_phy'm (pm_phy'ro_addr ∪ pm_phy'rw_addr))	
{-8}	pm'states = pm_phy'states	
{-9}	(address_block(a', length(bl')) ⊆ pm'rw_addr)	
{-10}	reg_base(min_linear)(type_of(a')) ≤ a'	
{-11}	a' + length(bl') ≤ (#type_of := type_of(a'), offset := max_linear_offset#)	
{-12}	pm'states(s')	
{1}	(extend [Address, Memory_Address_4G, bool, FALSE](virt_to_phys_range(s', restrict [Address,	
{2}	unchanged_memory_invariant?(pm_phy'mem, pm_phy'states, singleton(expr_2_super(λ (s: Linear_memory [Physical_memory, pm_ apply_side_effects(split (min_p a', bl'), lin- (s))), extend[Address, Memory_Address_4G, bool, FALSE] (virt_to_phys_range(s', restrict [Address, Mem- ory_Address_4G, boolean] ((pm'ro_addr ∪ pm'rw_ad (singleton(Read) ∪ singleton(Ex	(min_p a', bl'), lin- (s))), restrict [Address, Mem- ory_Address_4G, boolean] ((pm'ro_addr ∪ pm'rw_ad (singleton(Read) ∪ singleton(Ex
{3}	null?(bl')	

Keeping (-5 1) and hiding *,

Expanding the definition of subset?,

Repeatedly Skolemizing and flattening,

Instantiating the top quantifier in -2 with the terms: (x!1),

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of linear_plain_unchanged_memory_invariant.2.1.4.1.2.1.1.1.2.

linear_plain_unchanged_memory_invariant.2.1.4.1.2.1.1.1.3:

{-1}	unchanged_memory_invariant?(pm_phy'mem, pm_phy'states, singleton(expr_2_super(apply_side_effects(split (min_page, a', bl') lin- ear_write_side_effect_in_page))), extend[Address, Memory_Address_4G, bool, FALSE] (restrict[Address, Mem- ory_Address_4G, boolean] ((pm_phy'ro_addr ∪ pm_phy'rw_addr))))
{-2}	is_linear_plain_memory?(pm')
{-3}	every(λ (t: [Address, list[Byte]]): Mem?(t'1'type_of) ∧ 0 ≤ t'1'offset ∧ t'1'offset < max_linear_offset) (split(min_page, a', bl'))
{-4}	Mem?(type_of(a'))
{-5}	(extend [Address, Memory_Address_4G, bool, FALSE](virt_to_phys_range(s', restrict [Address, Memory_
{-6}	plain_memory?(pm_phy)
{-7}	unchanged_memory_invariant?(pm_phy'mem, pm_phy'states, ((pm_phy'other_actions ∪ memory_read_transformers(pm_phy'mem, (pm- (pm_phy'ro_addr ∪ pm_phy'rw_addr)))
{-8}	pm'states = pm_phy'states
{-9}	(address_block(a', length(bl')) ⊆ pm'rw_addr)
{-10}	reg_base(min_linear)(type_of(a')) ≤ a'
{-11}	a' + length(bl') ≤ (#type_of := type_of(a'), offset := max_linear_offset#)
{-12}	pm'states(s')
{1}	(singleton(expr_2_super(λ (s: Linear_memory[Physical_memory, pm_phy]): apply_side_effects(split(min- {2} unchanged_memory_invariant?(pm_phy'mem, pm_phy'states, singleton(expr_2_super(λ (s: Linear_memory [Physical_memory, pm_phy]): apply_side_effects(split (min_page, a', bl'), lin- ear_write_side_effect_in_page) (s))), extend[Address, Memory_Address_4G, bool, FALSE] (virt_to_phys_range(s', restrict [Address, Mem- ory_Address_4G, boolean] ((pm'ro_addr ∪ pm'rw_addr)), (singleton(Read) ∪ singleton(Execute))))))
{3}	null?(bl')

Keeping (1) and hiding *,

Expanding the definition of expr_2_super,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of linear_plain_unchanged_memory_invariant.2.1.4.1.2.1.1.1.3.

C Proof scripts

linear_plain_unchanged_memory_invariant.2.1.4.1.2.1.1.2:

{-1}	is_linear_plain_memory?(pm')	
{-2}	Mem?(type_of(a'))	
{-3}	(extend [Address, Memory_Address_4G, bool, FALSE](virt_to_phys_range(s', restrict [Address,	
{-4}	plain_memory?(pm_phy)	
{-5}	unchanged_memory_invariant?(pm_phy'mem, pm_phy'states,	
	((pm_phy'other_actions ∪ memory_read_transformers(pm_phy'mem,	
	(pm_phy'ro_addr ∪ pm_phy'rw_addr))	
{-6}	pm'states = pm_phy'states	
{-7}	(address_block(a', length(bl')) ⊆ pm'rw_addr)	
{-8}	reg_base(min_linear)(type_of(a')) ≤ a'	
{-9}	a' + length(bl) ≤ (#type_of := type_of(a'), offset := max_linear_offset#)	
{-10}	pm'states(s')	
{1}	(address_block(a', length(bl')) ⊆ (pm'ro_addr ∪ pm'rw_addr))	
{2}	null?(bl')	
{3}	unchanged_memory_invariant?(pm_phy'mem, pm_phy'states,	
	singleton(expr_2_super(λ (s:	
		Linear_memory
		[Physical_memory, pm_
		apply_side_effects(split
		(min_p
		a',
		bl'),
		lin-
	ear_write_side_effect_in_page)	(s)),
		extend[Address, Memory_Address_4G, bool, FALSE]
		(virt_to_phys_range(s',
		restrict
		[Address, Mem-
	ory_Address_4G, boolean]	((pm'ro_addr ∪ pm'rw_ad
		(singleton(Read) ∪ singleton(Ex

Keeping (-7 1) and hiding *,

Rewriting using subset_transitive, matching in * where a gets address_block(a!1, length(bl!1)), b gets pm!1'rw_addr, c gets union(pm!1'ro_addr, pm!1'rw_addr),

Rewriting using union_subset3, matching in *,

This completes the proof of linear_plain_unchanged_memory_invariant.2.1.4.1.2.1.1.2.

linear_plain_unchanged_memory_invariant.2.1.4.1.2.1.2:

{-1}	Mem?(type_of(a'))	
{-2}	(extend [Address, Memory_Address_4G, bool, FALSE](virt_to_phys_range(s', restrict [Address, Memory_	
{-3}	plain_memory?(pm_phy)	
{-4}	unchanged_memory_invariant?(pm_phy' mem, pm_phy' states,	
	(pm_phy'other_actions ∪ memory_read_transformers(pm_phy' mem, (pm_	
	(pm_phy'ro_addr ∪ pm_phy'rw_addr))	
{-5}	is_linear_plain_memory?(pm')	
{-6}	pm' states = pm_phy' states	
{-7}	(address_block(a', length(bl')) ⊆ pm'rw_addr)	
{-8}	reg_base(min_linear)(type_of(a')) ≤ a'	
{-9}	a' + length(bl') ≤ (#type_of := type_of(a'), offset := max_linear_offset#)	
{-10}	pm' states(s')	
{1}	0 ≤ a'offset ∧ a'offset < max_linear_offset	
{2}	null?(bl')	
{3}	unchanged_memory_invariant?(pm_phy' mem, pm_phy' states,	
	singleton(expr_2_super(λ (s:	
		Linear_memory
		[Physical_memory, pm_phy]):
		apply_side_effects(split
		(min_page,
		a',
		bl'),
		lin-
	ear_write_side_effect_in_page)	(s)),
		extend[Address, Memory_Address_4G, bool, FALSE]
		(virt_to_phys_range(s',
		restrict
		[Address, Mem-
	ory_Address_4G, boolean]	((pm'ro_addr ∪ pm'rw_addr)),
		(singleton(Read) ∪ singleton(Execute))))

Keeping (-8 -9 1 2) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of linear_plain_unchanged_memory_invariant.2.1.4.1.2.1.2.

C Proof scripts

linear_plain_unchanged_memory_invariant.2.1.4.1.2.2:

{-1}	Mem?(type_of(a'))	
{-2}	(extend [Address, Memory_Address_4G, bool, FALSE](virt_to_phys_range(s', restrict [Address,	
{-3}	plain_memory?(pm_phy)	
{-4}	unchanged_memory_invariant?(pm_phy'mem, pm_phy'states,	((pm_phy'other_actions ∪ memory_read_transformers(pm_phy'mem,
		(pm_phy'ro_addr ∪ pm_phy'rw_addr))
{-5}	is_linear_plain_memory?(pm')	
{-6}	pm'states = pm_phy'states	
{-7}	(address_block(a', length(bl')) ⊆ pm'rw_addr)	
{-8}	pm'states(s')	
{1}	null?(bl')	
{2}	a' + length(bl') ≤ (#type_of := type_of(a'), offset := max_linear_offset#)	
{3}	unchanged_memory_invariant?(pm_phy'mem, pm_phy'states,	singleton(expr_2_super(λ (s:
		Linear_memory
		[Physical_memory, pm_p
		Fatal)),
	extend[Address, Memory_Address_4G, bool, FALSE]	
	(virt_to_phys_range(s',	
		restrict
		[Address, Mem-
	ory_Address_4G, boolean]	
		((pm'ro_addr ∪ pm'rw_addr
		(singleton(Read) ∪ singleton(Exec

Keeping (3) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of linear_plain_unchanged_memory_invariant.2.1.4.1.2.2.

linear_plain_unchanged_memory_invariant.2.1.4.1.2.3:

<pre> {-1} Mem?(type_of(a')) {-2} (extend [Address, Memory_Address_4G, bool, FALSE](virt_to_phys_range(s', restrict [Address, Memory_A {-3} plain_memory?(pm_phy) {-4} unchanged_memory_invariant?(pm_phy'mem, pm_phy'states, ((pm_phy'other_actions ∪ memory_read_transformers(pm_phy'mem, (pm_pl (pm_phy'ro_addr ∪ pm_phy'rw_addr)) {-5} is_linear_plain_memory?(pm') {-6} pm'states = pm_phy'states {-7} (address_block(a', length(bl')) ⊆ pm'rw_addr) {-8} pm'states(s') </pre>	<pre> {1} null?(bl') {2} reg_base(min_linear)(type_of(a')) ≤ a' {3} unchanged_memory_invariant?(pm_phy'mem, pm_phy'states, singleton(expr_2_super(λ (s: Linear_memory [Physical_memory, pm_phy]): Fatal)), extend[Address, Memory_Address_4G, bool, FALSE] (virt_to_phys_range(s', restrict [Address, Mem- ory_Address_4G, boolean] ((pm'ro_addr ∪ pm'rw_addr)), (singleton(Read) ∪ singleton(Execute)))))) </pre>
---	--

Using lemma pm_memory_addr,

Simplifying, rewriting, and recording with decision procedures,

Keeping (-7 1 2) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of linear_plain_unchanged_memory_invariant.2.1.4.1.2.3.

linear_plain_unchanged_memory_invariant.2.1.4.2:

<pre> {-1} (extend [Address, Memory_Address_4G, bool, FALSE](virt_to_phys_range(s', restrict [Address, {-2} plain_memory?(pm_phy) {-3} unchanged_memory_invariant?(pm_phy'mem, pm_phy'states, ((pm_phy'other_actions ∪ memory_read_transformers(pm_phy'mem, (pm_phy'ro_addr ∪ pm_phy'rw_addr)) {-4} is_linear_plain_memory?(pm') {-5} pm'states = pm_phy'states {-6} (address_block(a', length(bl')) ⊆ pm'rw_addr) {-7} pm'states(s') </pre>	<pre> {1} Mem?(type_of(a')) {2} unchanged_memory_invariant?(pm_phy'mem, pm_phy'states, singleton(expr_2_super(λ (s : Linear_memory [Physical_memory, pm_p IF null?(bl') ∨ (reg_base (min_linear)(type_of(a')) ≤ a' ∧ a' + length(bl') ≤ reg_size (max_linear)(type_of(a')), THEN IF Mem?(type_of(a')) THEN ap- (split(min_page, lin- (s) ELSE mem- (pm_phy'mem) (a', bl', cp')(s) ENDIF ELSE Fatal ENDIF)), extend[Address, Memory_Address_4G, bool, FALSE] (virt_to_phys_range(s', restrict [Address, Mem- ((pm'ro_addr ∪ pm'rw_addr) (singleton(Read) ∪ singleton(Exe </pre>
--	--

Using lemma unchanged_memory_invariant_mono,

Case splitting on subset?(singleton(expr_2_super(LAMBDA (s: Linear_memory [Physical_memory, pm_phy]): memory_write_side_effect(pm_phy'mem) (a!1, bl!1, cp!1)(s))), union(union(pm_phy'other_actions, memory_read_transformers(pm_phy'mem, union (pm_phy'ro_addr, pm_phy'rw_addr))), union(memory_read_side_effect(pm_phy'ro_addr, pm_phy'rw_addr), memory_write_side_effect_super_transformers(pm_phy'mem,

`pm_phy'rw_addr))))),`

we get 2 subgoals:

linear_plain_unchanged_memory_invariant.2.1.4.2.1:

```

{-1} (singleton(expr_2_super( $\lambda$  ( $s$  : Linear_memory[Physical_memory, pm_phy]) : memory_write_side.
{-2} (singleton(expr_2_super( $\lambda$  ( $s$  : Linear_memory[Physical_memory, pm_phy]) : memory_write_side.
    ^
    (extend [Address, Memory_Address_4G, bool, FALSE](virt_to_phys_range( $s'$ , restrict [Address, Mem-
    ^
    unchanged_memory_invariant?(pm_phy'mem, pm_phy'states,
        ((pm_phy'other_actions  $\cup$  memory_read_transformers(pm_phy'mem,
        (pm_phy'ro_addr  $\cup$  pm_phy'rw_addr))
     $\supset$ 
    unchanged_memory_invariant?(pm_phy'mem, pm_phy'states,
        singleton(expr_2_super( $\lambda$  ( $s$  :
            Linear_memory
            [Physical_memory, pm_phy'mem-
            mem-
            ory_write_side_effect(pm_phy'mem)
            ( $a'$ ,  $bl'$ ,  $cp'$ )( $s$ ))),
            extend [Address, Memory_Address_4G, bool, FALSE]
            (virt_to_phys_range( $s'$ ,
            restrict
            [Address, Mem-
            ory_Address_4G, boolean]
            ((pm'ro_addr  $\cup$  pm'rw_addr)
            (singleton(Read)  $\cup$  singleton(Ex-
{-3} (extend [Address, Memory_Address_4G, bool, FALSE](virt_to_phys_range( $s'$ , restrict [Address, Mem-
{-4} plain_memory?(pm_phy)
{-5} unchanged_memory_invariant?(pm_phy'mem, pm_phy'states,
        ((pm_phy'other_actions  $\cup$  memory_read_transformers(pm_phy'mem,
        (pm_phy'ro_addr  $\cup$  pm_phy'rw_addr))
{-6} is_linear_plain_memory?(pm')
{-7} pm'states = pm_phy'states
{-8} (address_block( $a'$ , length( $bl'$ ))  $\subseteq$  pm'rw_addr)
{-9} pm'states( $s'$ )
-----
{1} Mem?(type_of( $a'$ ))
{2} unchanged_memory_invariant?(pm_phy'mem, pm_phy'states,
    singleton(expr_2_super( $\lambda$  ( $s$  :
        Linear_memory
        [Physical_memory, pm_p-
        IF null?( $bl'$ )  $\vee$ 
        (reg_base
        (min_linear)(type_of( $a'$ ))
         $\leq$ 
         $a'$ 
         $\wedge$ 
         $a' + \text{length}(bl')$ 
         $\leq$ 
        reg_size
        (max_linear)(type_of( $a'$ )),
        THEN IF Mem?(type_of( $a'$ ))
        THEN ap-
        ply_side_effects
        (split(min_page,
        lin-
        ear_write_side_effect_in_page)
        ( $s$ )
        ELSE mem-
        ory_write_side_effect
        (pm_phy'mem)
        ( $a'$ ,  $bl'$ ,  $cp'$ )( $s$ )
        ENDIF
        ELSE Fatal
        ENDIF)),
    extend [Address, Memory_Address_4G, bool, FALSE]
    (virt_to_phys_range( $s'$ ,

```

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 2 subgoals:

`linear_plain_unchanged_memory_invariant.2.1.4.2.1.1:`

{-1}	<code>(singleton(expr_2_super(λ (s : Linear_memory[Physical_memory, pm_phy]) : memory_write_side_effect(pm_</code>
{-2}	<code>extend [Address, Memory_Address_4G, bool, FALSE](virt_to_phys_range(s', restrict [Address, Memory_A</code>
{-3}	<code>unchanged_memory_invariant?(pm_phy'mem, pm_phy'states,</code> <code>((pm_phy'other_actions \cup memory_read_transformers(pm_phy'mem, (pm_ph</code> <code>(pm_phy'ro_addr \cup pm_phy'rw_addr))</code>
{-4}	<code>unchanged_memory_invariant?(pm_phy'mem, pm_phy'states,</code> <code>singleton(expr_2_super(λ (s :</code> <code>Linear_memory</code> <code>[Physical_memory, pm_phy]):</code> <code>mem-</code> <code>ory_write_side_effect(pm_phy'mem)</code> <code>(a', bl', cp')(s))),</code> <code>extend[Address, Memory_Address_4G, bool, FALSE]</code> <code>(virt_to_phys_range(s',</code> <code>restrict</code> <code>[Address, Mem-</code> <code>ory_Address_4G, boolean]</code> <code>((pm'ro_addr \cup pm'rw_addr)),</code> <code>(singleton(Read) \cup singleton(Execute))))))</code>
{-5}	<code>plain_memory?(pm_phy)</code>
{-6}	<code>is_linear_plain_memory?(pm')</code>
{-7}	<code>pm'states = pm_phy'states</code>
{-8}	<code>(address_block(a', length(bl')) \subseteq pm'rw_addr)</code>
{-9}	<code>pm'states(s')</code>
{1}	<code>Mem?(type_of(a'))</code>
{2}	<code>null?(bl')</code>
{3}	<code>a' + length(bl') \leq</code> <code>reg_size((#type_of := Mem_, offset := max_linear_offset#))(type_of(a'))</code>
{4}	<code>unchanged_memory_invariant?(pm_phy'mem, pm_phy'states,</code> <code>singleton(expr_2_super(λ (s :</code> <code>Linear_memory</code> <code>[Physical_memory, pm_phy]):</code> <code>Fatal)),</code> <code>extend[Address, Memory_Address_4G, bool, FALSE]</code> <code>(virt_to_phys_range(s',</code> <code>restrict</code> <code>[Address, Mem-</code> <code>ory_Address_4G, boolean]</code> <code>((pm'ro_addr \cup pm'rw_addr)),</code> <code>(singleton(Read) \cup singleton(Execute))))))</code>

Keeping (4) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `linear_plain_unchanged_memory_invariant.2.1.4.2.1.1`.

linear_plain_unchanged_memory_invariant.2.1.4.2.1.2:

{-1}	(singleton(expr_2_super($\lambda (s : \text{Linear_memory}[\text{Physical_memory}, \text{pm_phy}]) : \text{memory_write_side}$	
{-2}	(extend [Address, Memory_Address_4G, bool, FALSE](virt_to_phys_range(s' , restrict [Address,	
{-3}	unchanged_memory_invariant?(pm_phy' mem, pm_phy' states,	
	((pm_phy'other_actions \cup memory_read_transformers(pm_phy'mem	
	(pm_phy'ro_addr \cup pm_phy'rw_addr))	
{-4}	unchanged_memory_invariant?(pm_phy' mem, pm_phy' states,	
	singleton(expr_2_super($\lambda (s :$	
		Linear_memory
		[Physical_memory, pm_p
		mem-
	ory_write_side_effect(pm_phy' mem)	
		(a' , bl' , cp')(s))),
	extend [Address, Memory_Address_4G, bool, FALSE]	
	(virt_to_phys_range(s' ,	
		restrict
		[Address, Mem-
	ory_Address_4G, boolean]	
		((pm'ro_addr \cup pm'rw_addr
		(singleton(Read) \cup singleton(Exe
{-5}	plain_memory?(pm_phy)	
{-6}	is_linear_plain_memory?(pm')	
{-7}	pm' states = pm_phy' states	
{-8}	(address_block(a' , length(bl')) \subseteq pm'rw_addr)	
{-9}	pm' states(s')	
{1}	Mem?(type_of(a'))	
{2}	null?(bl')	
{3}	reg_base(min_linear)(type_of(a')) $\leq a'$	
{4}	unchanged_memory_invariant?(pm_phy' mem, pm_phy' states,	
	singleton(expr_2_super($\lambda (s :$	
		Linear_memory
		[Physical_memory, pm_p
		Fatal)),
	extend [Address, Memory_Address_4G, bool, FALSE]	
	(virt_to_phys_range(s' ,	
		restrict
		[Address, Mem-
	ory_Address_4G, boolean]	
		((pm'ro_addr \cup pm'rw_addr
		(singleton(Read) \cup singleton(Exe

Using lemma pm_memory_addr,

Simplifying, rewriting, and recording with decision procedures,

Keeping (-8 1 3) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of linear_plain_unchanged_memory_invariant.2.1.4.2.1.2.

linear_plain_unchanged_memory_invariant.2.1.4.2.2:

```

{-1} (singleton(expr_2_super( $\lambda$  (s : Linear_memory[Physical_memory, pm_phy]) : memory_write_side_effect(pm_phy'
    ^
    (extend [Address, Memory_Address_4G, bool, FALSE](virt_to_phys_range(s', restrict [Address, Memory_Address_4G, boolean]
    ^
    unchanged_memory_invariant?(pm_phy' mem, pm_phy' states,
        ((pm_phy' other_actions  $\cup$  memory_read_transformers(pm_phy' mem, (pm_phy' ro_addr  $\cup$  pm_phy' rw_addr))
     $\supset$ 
    unchanged_memory_invariant?(pm_phy' mem, pm_phy' states,
        singleton(expr_2_super( $\lambda$  (s :
            Linear_memory
            [Physical_memory, pm_phy]):
            mem-
            ory_write_side_effect(pm_phy' mem)
            (a', bl', cp')(s))),
            extend[Address, Memory_Address_4G, bool, FALSE]
            (virt_to_phys_range(s',
            restrict
            [Address, Mem-
            ory_Address_4G, boolean]
            ((pm' ro_addr  $\cup$  pm' rw_addr)),
            (singleton(Read)  $\cup$  singleton(Execute))))))
{-2} (extend [Address, Memory_Address_4G, bool, FALSE](virt_to_phys_range(s', restrict [Address, Memory_Address_4G, boolean]
{-3} plain_memory?(pm_phy)
{-4} unchanged_memory_invariant?(pm_phy' mem, pm_phy' states,
    ((pm_phy' other_actions  $\cup$  memory_read_transformers(pm_phy' mem, (pm_phy' ro_addr  $\cup$  pm_phy' rw_addr))
{-5} is_linear_plain_memory?(pm')
{-6} pm' states = pm_phy' states
{-7} (address_block(a', length(bl'))  $\subseteq$  pm' rw_addr)
{-8} pm' states(s')
-----
{1} (singleton(expr_2_super( $\lambda$  (s : Linear_memory[Physical_memory, pm_phy]) : memory_write_side_effect(pm_phy'
{2} Mem?(type_of(a'))
{3} unchanged_memory_invariant?(pm_phy' mem, pm_phy' states,
    singleton(expr_2_super( $\lambda$  (s :
        Linear_memory
        [Physical_memory, pm_phy]):
        IF null?(bl')  $\vee$ 
        (reg_base
            (min_linear)(type_of(a'))
             $\leq$ 
            a'
             $\wedge$ 
            a' + length(bl')
             $\leq$ 
            reg_size
            (max_linear)(type_of(a'))))
        THEN IF Mem?(type_of(a'))
            THEN ap-
            (split(min_page, a', bl'),
            lin-
            (s)
            ELSE mem-
            ory_write_side_effect
            (pm_phy' mem)
            (a', bl', cp')(s)
            ENDIF
        ELSE Fatal
        ENDIF)),
    extend[Address, Memory_Address_4G, bool, FALSE]
    (virt_to_phys_range(s',

```

C Proof scripts

Hiding formulas: (-1 -4 3),

Case splitting on `memory_write_side_effect_super_transformers(pm_phy'mem, pm_phy'rw_addr)`
`(expr_2_super (LAMBDA (s: Linear_memory [Physical_memory, pm_phy]): memory_write_side_effect`
`(pm_phy'mem) (a!1, bl!1, cp!1)(s)))`,

we get 2 subgoals:

`linear_plain_unchanged_memory_invariant.2.1.4.2.2.1:`

{-1}	<code>memory_write_side_effect_super_transformers(pm_phy'mem, pm_phy'rw_addr)</code>	<code>(expr_2_super(λ (s:</code>	<code>Lin-</code>
	<code>ear_memory</code>	<code>[Physical_memory, pm-</code>	<code>mem-</code>
	<code>ory_write_side_effect(pm_phy'mem)</code>	<code>(a', bl', cp')(s)))</code>	
{-2}	<code>(extend [Address, Memory_Address_4G, bool, FALSE](virt_to_phys_range(s', restrict [Address,</code>		
{-3}	<code>plain_memory?(pm_phy)</code>		
{-4}	<code>is_linear_plain_memory?(pm')</code>		
{-5}	<code>pm' states = pm_phy' states</code>		
{-6}	<code>(address_block(a', length(bl')) ⊆ pm'rw_addr)</code>		
{-7}	<code>pm' states(s')</code>		
{1}	<code>(singleton(expr_2_super(λ (s: Linear_memory [Physical_memory, pm_phy]): memory_write_side.</code>		
{2}	<code>Mem?(type_of(a'))</code>		

Keeping (-1 -6 1) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `linear_plain_unchanged_memory_invariant.2.1.4.2.2.1`.

`linear_plain_unchanged_memory_invariant.2.1.4.2.2.2:`

{-1}	<code>(extend [Address, Memory_Address_4G, bool, FALSE](virt_to_phys_range(s', restrict [Address,</code>		
{-2}	<code>plain_memory?(pm_phy)</code>		
{-3}	<code>is_linear_plain_memory?(pm')</code>		
{-4}	<code>pm' states = pm_phy' states</code>		
{-5}	<code>(address_block(a', length(bl')) ⊆ pm'rw_addr)</code>		
{-6}	<code>pm' states(s')</code>		
{1}	<code>memory_write_side_effect_super_transformers(pm_phy'mem, pm_phy'rw_addr)</code>	<code>(expr_2_super(λ (s:</code>	<code>Lin-</code>
	<code>ear_memory</code>	<code>[Physical_memory, pm-</code>	<code>mem-</code>
	<code>ory_write_side_effect(pm_phy'mem)</code>	<code>(a', bl', cp')(s)))</code>	
{2}	<code>(singleton(expr_2_super(λ (s: Linear_memory [Physical_memory, pm_phy]): memory_write_side.</code>		
{3}	<code>Mem?(type_of(a'))</code>		

Hiding formulas: 2,

Expanding the definition of `memory_write_side_effect_super_transformers`,

Instantiating quantified variables,

Expanding the definition of `expr_2_super`,

Expanding the definition of `subset?`,

Repeatedly Skolemizing and flattening,

Instantiating quantified variables,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Expanding the definition of member,
 Expanding the definition of address_block,
 Using lemma pm_memory_addr,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 This completes the proof of `linear_plain_unchanged_memory_invariant.2.1.4.2.2.2`.
`linear_plain_unchanged_memory_invariant.2.2:`

{-1}	$\text{plain_memory?}(\text{pm_phy}) \supset$ $\text{unchanged_memory_invariant?}(\text{pm_phy}'\text{mem}, \text{pm_phy}'\text{states},$ $\quad ((\text{pm_phy}'\text{other_actions} \cup \text{memory_read_transformers}(\text{pm_phy}'\text{mem}, (\text{pm_phy}'\text{ro_addr} \cup \text{pm_phy}'\text{rw_addr})))$
{-2}	$\text{is_linear_plain_memory?}(\text{pm}') \supset \text{pm}'\text{states} = \text{pm_phy}'\text{states}$
{-3}	$\text{is_linear_plain_memory?}(\text{pm}') \supset \text{plain_memory?}(\text{pm_phy})$
{-4}	$(\text{pm}'\text{other_actions}(q') \vee$ $\quad \text{memory_read_transformers}(\text{pm}'\text{mem}, (\text{pm}'\text{ro_addr} \cup \text{pm}'\text{rw_addr}))(q')$ \vee $\quad \text{memory_read_side_effect_super_transformers}(\text{pm}'\text{mem}, (\text{pm}'\text{ro_addr} \cup \text{pm}'\text{rw_addr}))(q')$ $\vee \text{memory_write_side_effect_super_transformers}(\text{pm}'\text{mem}, \text{pm}'\text{rw_addr})(q')$
{-5}	$\text{pm}'\text{states}(s')$
{-6}	$\text{is_linear_plain_memory?}(\text{pm}')$
{1}	$(\text{extend} [\text{Address}, \text{Memory_Address_4G}, \text{bool}, \text{FALSE}] (\text{virt_to_phys_range}(s', \text{restrict} [\text{Address}, \text{Memory_Address_4G}, \text{boolean}] (\text{pm}'\text{ro_addr} \cup \text{pm}'\text{rw_addr})),$
{2}	$\text{unchanged_memory_invariant?}(\text{pm_phy}'\text{mem}, \text{pm_phy}'\text{states}, \text{singleton}(q'),$ $\quad \text{extend} [\text{Address}, \text{Memory_Address_4G}, \text{bool}, \text{FALSE}]$ $\quad (\text{virt_to_phys_range}(s',$ $\quad \quad \text{restrict}$ $\quad \quad [\text{Address}, \text{Memory_Address_4G}, \text{boolean}]$ $\quad \quad (\text{pm}'\text{ro_addr} \cup \text{pm}'\text{rw_addr})),$ $\quad \quad (\text{singleton}(\text{Read}) \cup \text{singleton}(\text{Execute}))))$

Hiding formulas: (-1 -2 -3 -4 2),
 Using lemma pm_linear_blessed,
 Expanding the definition of linear_blessed?,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 Keeping (-3 1) and hiding *,
 Case splitting on $\text{union}(\text{restrict}[\text{Address}, \text{Memory_Address_4G}, \text{boolean}](\text{pm}'\text{ro_addr}), \text{restrict}[\text{Address}, \text{Memory_Address_4G}, \text{boolean}](\text{pm}'\text{rw_addr})) = \text{restrict}[\text{Address}, \text{Memory_Address_4G}, \text{boolean}](\text{union}(\text{pm}'\text{ro_addr}, \text{pm}'\text{rw_addr}))$,
 we get 2 subgoals:

`linear_plain_unchanged_memory_invariant.2.2.1:`

{-1}	$(\text{restrict}[\text{Address}, \text{Memory_Address_4G}, \text{boolean}](\text{pm}'\text{ro_addr}) \cup \text{restrict} [\text{Address}, \text{Memory_Address_4G}, \text{boolean}] (\text{pm}'\text{rw_addr}))$ $= \text{restrict}[\text{Address}, \text{Memory_Address_4G}, \text{boolean}]((\text{pm}'\text{ro_addr} \cup \text{pm}'\text{rw_addr}))$
{-2}	$(\text{virt_to_phys_range}(s', (\text{restrict} [\text{Address}, \text{Memory_Address_4G}, \text{boolean}](\text{pm}'\text{ro_addr}) \cup \text{restrict} [\text{Address}, \text{Memory_Address_4G}, \text{boolean}] (\text{pm}'\text{rw_addr}))))$
{1}	$(\text{extend} [\text{Address}, \text{Memory_Address_4G}, \text{bool}, \text{FALSE}] (\text{virt_to_phys_range}(s', \text{restrict} [\text{Address}, \text{Memory_Address_4G}, \text{boolean}] (\text{pm}'\text{ro_addr}) \cup \text{restrict} [\text{Address}, \text{Memory_Address_4G}, \text{boolean}] (\text{pm}'\text{rw_addr}))))$

Replacing using formula -1,
 Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of `linear_plain_unchanged_memory_invariant.2.2.1`.
`linear_plain_unchanged_memory_invariant.2.2.2:`

{-1}	$(\text{virt_to_phys_range}(s', (\text{restrict} [\text{Address}, \text{Memory_Address_4G}, \text{boolean}](\text{pm}'\text{ro_addr}) \cup \text{restrict} [\text{Address}, \text{Memory_Address_4G}, \text{boolean}] (\text{pm}'\text{rw_addr}))))$
{1}	$(\text{restrict}[\text{Address}, \text{Memory_Address_4G}, \text{boolean}](\text{pm}'\text{ro_addr}) \cup \text{restrict} [\text{Address}, \text{Memory_Address_4G}, \text{boolean}] (\text{pm}'\text{rw_addr}))$ $= \text{restrict}[\text{Address}, \text{Memory_Address_4G}, \text{boolean}]((\text{pm}'\text{ro_addr} \cup \text{pm}'\text{rw_addr}))$
{2}	$(\text{extend} [\text{Address}, \text{Memory_Address_4G}, \text{bool}, \text{FALSE}] (\text{virt_to_phys_range}(s', \text{restrict} [\text{Address}, \text{Memory_Address_4G}, \text{boolean}] (\text{pm}'\text{ro_addr}) \cup \text{restrict} [\text{Address}, \text{Memory_Address_4G}, \text{boolean}] (\text{pm}'\text{rw_addr}))))$

C Proof scripts

Keeping (1) and hiding *,

Applying decompose-equality,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `linear_plain_unchanged_memory_invariant.2.2.2`.

Q.E.D.

C.116.3 Linear_Memory_Blessing_Properties.linear_plain_unchanged_memory_invariant_w

Terse proof for `linear_plain_unchanged_memory_invariant_write`.

`linear_plain_unchanged_memory_invariant_write`:

$\{1\} \quad \forall (pm: \text{Plain_Memory}[\text{Physical_memory}]):$ $\text{is_linear_plain_memory?}(pm) \supset$ $\text{unchanged_memory_invariant?}(pm' \text{mem}, pm' \text{states}, \text{mem-}$ $\text{ory_write_transformers}(pm' \text{mem}, pm' \text{rw_addr},$ $pm' \text{ro_addr})$
--

Repeatedly Skolemizing and flattening,

Installing automatic rewrites from: `has_next_state singleton subset_reflexive pm_write_linear linear_write Mem in_memory Mem reg_size union_left max_linear < min_linear Address_Helpers.<= reg_base`

Using lemma `linear_unchanged_invariant`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 3 subgoals:

`linear_plain_unchanged_memory_invariant_write.1`:

$\{-1\} \quad \text{is_linear_plain_memory?}(pm')$ $\{-2\} \quad \text{unchanged_memory_invariant?}(pm' \text{mem}, pm' \text{states},$ $\text{memory_write_transformers}(pm' \text{mem}, pm' \text{rw_addr}),$ $\text{extend}[\text{Address}, \text{Memory_Address_4G}, \text{bool}, \text{FALSE}]$ $(\text{restrict}[\text{Address}, \text{Mem-}$ $\text{ory_Address_4G}, \text{boolean}]$ $\text{(pm' ro_addr)}))$
$\{1\} \quad \text{unchanged_memory_invariant?}(pm' \text{mem}, pm' \text{states},$ $\text{memory_write_transformers}(pm' \text{mem}, pm' \text{rw_addr}), pm' \text{ro.}$

Case splitting on `extend[Address, Memory_Address_4G, bool, FALSE] (restrict[Address, Memory_Address_4G, boolean](pm!1'ro_addr)) = pm!1'ro_addr`,

we get 2 subgoals:

linear_plain_unchanged_memory_invariant_write.1.1:

<pre>{-1} extend[Address, Memory_Address_4G, bool, FALSE] (restrict[Address, Memory_Address_4G, boolean](pm'ro_addr)) = pm'ro_addr {-2} is_linear_plain_memory?(pm') {-3} unchanged_memory_invariant?(pm'mem, pm'states, memory_write_transformers(pm'mem, pm'rw_addr), extend[Address, Memory_Address_4G, bool, FALSE] (restrict[Address, Mem- ory_Address_4G, boolean] (pm'ro_addr)))</pre>	<pre>{1} unchanged_memory_invariant?(pm'mem, pm'states, memory_write_transformers(pm'mem, pm'rw_addr), pm'ro_addr)</pre>
---	--

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of linear_plain_unchanged_memory_invariant_write.1.1.

linear_plain_unchanged_memory_invariant_write.1.2:

<pre>{-1} is_linear_plain_memory?(pm') {-2} unchanged_memory_invariant?(pm'mem, pm'states, memory_write_transformers(pm'mem, pm'rw_addr), extend[Address, Memory_Address_4G, bool, FALSE] (restrict[Address, Mem- ory_Address_4G, boolean] (pm'ro_addr)))</pre>	<pre>{1} extend[Address, Memory_Address_4G, bool, FALSE] (restrict[Address, Memory_Address_4G, boolean](pm'ro_addr)) = pm'ro_addr {2} unchanged_memory_invariant?(pm'mem, pm'states, memory_write_transformers(pm'mem, pm'rw_addr), pm'ro_addr)</pre>
---	---

Hiding formulas: (-2 2),

Expanding the definition of extend,

Expanding the definition of restrict,

Applying decompose-equality,

Using lemma pm_memory_addr,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of linear_plain_unchanged_memory_invariant_write.1.2.

linear_plain_unchanged_memory_invariant_write.2:

<pre>{-1} is_linear_plain_memory?(pm')</pre>	<pre>{1} $\forall (s: (pm'states)):$ unchanged_memory_invariant?(pm_phy'mem, pm_phy'states, memory_write_transformers(pm'mem, pm'rw_addr), extend[Address, Memory_Address_4G, bool, FALSE] (virt_to_phys_range(s, restrict [Address, Mem- ory_Address_4G, boolean] (pm'ro_addr), (singleton(Read) \cup singleton(Execute))))))</pre>
<pre>{2} unchanged_memory_invariant?(pm'mem, pm'states, memory_write_transformers(pm'mem, pm'rw_addr), pm'ro_addr)</pre>	

C Proof scripts

Hiding formulas: 2,
 Repeatedly Skolemizing and flattening,
 Rewriting using `unchanged_memory_invariant_all_transformers`, matching in * where pm gets `pm_phy'mem`,

Hiding formulas: 2,
 Repeatedly Skolemizing and flattening,
 Expanding the definition of `memory_write_transformers`,
 Repeatedly Skolemizing and flattening,
 Replacing using formula -3,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 Expanding the definition of `linear_write`,
 Hiding formulas: (-3),
 Using lemma `pm_memory_addr`,
 Installing automatic rewrites from: `union_right`
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 Using lemma `pm_plain_phy`,

Rewriting using `plain_memory_unchanged_composition`[`Physical_memory`, `Address`, `Unit`], matching in * where pm gets `pm_phy`, addresses gets `extend`[`Address`, `Memory_Address_4G`, `bool`, `FALSE`] (`virt_to_phys_range`(`s!1`, `restrict`[`Address`, `Memory_Address_4G`, `boolean`] (`pm!1'ro_addr`), `union`(`singleton`(`Read`), `singleton`(`Execute`))))), P gets LAMBDA (a: `Address`): FORALL (s: (`pm_phy'states`)): OK?(`linear_resolve`(`a!1`, `Write`)(`s`)) IMPLIES `data`(`linear_resolve`(`a!1`, `Write`)(`s`)) = a,

we get 4 subgoals:

`linear_plain_unchanged_memory_invariant_write.2.1:`

<pre> {-1} plain_memory?(pm_phy) {-2} is_linear_plain_memory?(pm') {-3} Mem?(type_of(a')) {-4} 0 ≤ a'offset {-5} a'type_of = a'type_of {-6} a'offset < max_linear_offset {-7} b' < max_byte {-8} pm'rw_addr(a') {-9} pm'states(s') </pre>	<pre> {1} (extend [Address, Memory_Address_4G, bool, FALSE](virt_to_phys_range(s', restrict [Address, {2} unchanged_memory_invariant?(pm_phy'mem, pm_phy'states, singleton(expr_2_super(linear_resolve(a', Write) ## (λ (pa: Address): mem- ory_write(pm_phy'mem) (pa, b')))), extend [Address, Memory_Address_4G, bool, FALSE] (virt_to_phys_range(s', restrict [Address, Mem- ory_Address_4G, boolean] (pm'ro_addr), (singleton(Read) ∪ singleton(Exe </pre>
---	--

Hiding formulas: 2,
 Using lemma `pm_linear_blessed`,
 Expanding the definition of `linear_blessed?`,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Keeping (-3 1) and hiding *,

Installing automatic rewrites from: subset? member extend restrict

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Repeatedly Skolemizing and flattening,

Expanding the definition of extend,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Instantiating the top quantifier in -1 with the terms: (x!1),

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Expanding the definition of virt_to_phys_range,

Repeatedly Skolemizing and flattening,

Instantiating quantified variables,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `linear_plain_unchanged_memory_invariant_write.2.1`.

`linear_plain_unchanged_memory_invariant_write.2.2`:

{-1}	<code>plain_memory?(pm_phy)</code>
{-2}	<code>is_linear_plain_memory?(pm')</code>
{-3}	<code>Mem?(type_of(a'))</code>
{-4}	<code>0 ≤ a'offset</code>
{-5}	<code>a'type_of = a'type_of</code>
{-6}	<code>a'offset < max_linear_offset</code>
{-7}	<code>b' < max_byte</code>
{-8}	<code>pm'rw_addr(a')</code>
{-9}	<code>pm'states(s')</code>
{1}	$\forall (s_1: (\text{pm_phy}'\text{states})): \text{OK}?(linear_resolve(a', \text{Write})(s_1)) \supset$ $(\forall (s: (\text{pm_phy}'\text{states})): \text{OK}?(linear_resolve(a', \text{Write})(s)) \supset$ $\text{data}(linear_resolve(a', \text{Write})(s)) = \text{data}(linear_resolve(a', \text{Write})(s_1)))$
{2}	$\text{unchanged_memory_invariant}?(pm_phy'\text{mem}, pm_phy'\text{states},$ $\text{singleton}(\text{expr_2_super}(linear_resolve(a', \text{Write}) \#\#$ $(\lambda (pa: \text{Address}): \text{mem-}$ $\text{ory_write}(pm_phy'\text{mem})$ $(\text{pa}, b'))),$ $\text{extend}[\text{Address}, \text{Memory_Address_4G}, \text{bool}, \text{FALSE}]$ $(\text{virt_to_phys_range}(s',$ restrict $[\text{Address}, \text{Mem-}$ $\text{ory_Address_4G}, \text{boolean}]$ $(\text{pm}'\text{ro_addr},$ $(\text{singleton}(\text{Read}) \cup \text{singleton}(\text{Execute}))))$

Hiding formulas: 2,

Repeatedly Skolemizing and flattening,

Using lemma `linear_resolve_same_result`,

Using lemma `pm_states`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `linear_plain_unchanged_memory_invariant_write.2.2`.

linear_plain_unchanged_memory_invariant_write.2.3:

<pre> {-1} plain_memory?(pm_phy) {-2} is_linear_plain_memory?(pm') {-3} Mem?(type_of(a')) {-4} 0 ≤ a'offset {-5} a'type_of = a'type_of {-6} a'offset < max_linear_offset {-7} b' < max_byte {-8} pm'rw_addr(a') {-9} pm'states(s') </pre>	<pre> {1} unchanged_memory_invariant?(pm_phy'mem, pm_phy'states, singleton(expr_2_super(linear_resolve(a', Write))), extend[Address, Memory_Address_4G, bool, FALSE] (virt_to_phys_range(s', restrict [Address, Mem- ory_Address_4G, boolean] (pm'ro_addr), (singleton(Read) ∪ singleton(Execu- {2} unchanged_memory_invariant?(pm_phy'mem, pm_phy'states, singleton(expr_2_super(linear_resolve(a', Write) ## (λ (pa: Address): mem- (pa, b')))), extend[Address, Memory_Address_4G, bool, FALSE] (virt_to_phys_range(s', restrict [Address, Mem- ory_Address_4G, boolean] (pm'ro_addr), (singleton(Read) ∪ singleton(Execu- </pre>
---	--

Hiding formulas: 2,

Using lemma linear_resolve_unchanged_pm_phy,

Using lemma pm_states,

Using lemma unchanged_memory_invariant_mono[Physical_memory],

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Hiding formulas: (-1 2),

Expanding the definition of subset?,

Repeatedly Skolemizing and flattening,

Expanding the definition of member,

Expanding the definition of extend,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Using lemma pm_resolve_address,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Keeping (-6 1) and hiding *,

Expanding the definition of virt_to_phys_range,

Repeatedly Skolemizing and flattening,

Instantiating quantified variables,

we get 2 subgoals:

linear_plain_unchanged_memory_invariant_write.2.3.1:

{-1}	Mem?(lin_a' 'type_of)
{-2}	$0 \leq \text{lin_a}'\text{'offset}$
{-3}	$\text{lin_a}'\text{'offset} < \text{max_linear_offset}$
{-4}	restrict[Address, Memory_Address_4G, boolean](pm' 'ro_addr)(lin_a')
{-5}	union singleton(Read), singleton(Execute))(ac')
{-6}	OK?(linear_resolve(lin_a', ac')(s'))
{-7}	data(linear_resolve(lin_a', ac')(s')) = x'
{1}	OK?(linear_resolve(lin_a', ac')(s')) \wedge data(linear_resolve(lin_a', ac')(s')) = x'

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of linear_plain_unchanged_memory_invariant_write.2.3.1.

linear_plain_unchanged_memory_invariant_write.2.3.2:

{-1}	Mem?(lin_a' 'type_of)
{-2}	$0 \leq \text{lin_a}'\text{'offset}$
{-3}	$\text{lin_a}'\text{'offset} < \text{max_linear_offset}$
{-4}	restrict[Address, Memory_Address_4G, boolean](pm' 'ro_addr)(lin_a')
{-5}	union singleton(Read), singleton(Execute))(ac')
{-6}	OK?(linear_resolve(lin_a', ac')(s'))
{-7}	data(linear_resolve(lin_a', ac')(s')) = x'
{1}	restrict[Address, Memory_Address_4G, boolean]((pm' 'ro_addr \cup pm' 'rw_addr))(lin_a')

Keeping (-4 1) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of linear_plain_unchanged_memory_invariant_write.2.3.2.

linear_plain_unchanged_memory_invariant_write.2.4:

<pre> {-1} plain_memory?(pm_phy) {-2} is_linear_plain_memory?(pm') {-3} Mem?(type_of(a')) {-4} 0 ≤ a'offset {-5} a'type_of = a'type_of {-6} a'offset < max_linear_offset {-7} b' < max_byte {-8} pm'rw_addr(a') {-9} pm'states(s') </pre>	<pre> {1} ∀ (d: (λ (a: Address): ∀ (s: (pm_phy'states)): OK?(linear_resolve(a', Write)(s)) ⊃ data(linear_resolve(a', Write)(s) = a)): unchanged_memory_invariant?(pm_phy'mem, pm_phy'states, singleton(expr_2_super(memory_write(pm_phy'mem)(d, b'))) extend[Address, Memory_Address_4G, bool, FALSE] (virt_to_phys_range(s', restrict [Address, Mem- ory_Address_4G, boolean] (pm'ro_addr), (singleton(Read) ∪ singleton(Exe- </pre>
<pre> {2} unchanged_memory_invariant?(pm_phy'mem, pm_phy'states, singleton(expr_2_super(linear_resolve(a', Write) ## (λ (pa: Address): mem- ory_write(pm_phy'mem) (pa, b')))), extend[Address, Memory_Address_4G, bool, FALSE] (virt_to_phys_range(s', restrict [Address, Mem- ory_Address_4G, boolean] (pm'ro_addr), (singleton(Read) ∪ singleton(Exe- </pre>	

Hiding formulas: 2,
Repeatedly Skolemizing and flattening,
Using lemma pm_states,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Instantiating quantified variables,
Installing automatic rewrites from: pm_linear_resolve_write_ok
Using lemma pm_plain_phy,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Using lemma plain_memory_unchanged_memory_invariant_write,
Using lemma plain_memory_unchanged_memory_write_invariant,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Expanding the definition of unchanged_memory_write_invariant?,
Instantiating quantified variables,
Case splitting on pm_phy'rw_addr(d!1),

we get 2 subgoals:

linear_plain_unchanged_memory_invariant_write.2.4.1:

{-1}	pm_phy'rw_addr(d')	
{-2}	plain_memory?(pm_phy)	
{-3}	pm_phy'rw_addr(d') \supset unchanged_memory_invariant?(pm_phy'mem, pm_phy'states, singleton(expr_2_super(memory_write(pm_phy'mem)(d' , b'))), (pm_phy'rw_addr \ { d' }))	
{-4}	unchanged_memory_invariant?(pm_phy'mem, pm_phy'states, memory_write_transformers(pm_phy'mem, pm_phy'rw_addr), pm_phy'ro_addr)	
{-5}	is_linear_plain_memory?(pm')	
{-6}	pm' states = pm_phy' states	
{-7}	data(linear_resolve(a' , Write)(s')) = d'	
{-8}	Mem?(type_of(a'))	
{-9}	$0 \leq a'$ 'offset	
{-10}	a' 'type_of = a' 'type_of	
{-11}	a' 'offset < max_linear_offset	
{-12}	$b' < \text{max_byte}$	
{-13}	pm'rw_addr(a')	
{-14}	pm' states(s')	
{1}	unchanged_memory_invariant?(pm_phy'mem, pm_phy'states, singleton(expr_2_super(memory_write(pm_phy'mem)(d' , b'))), extend[Address, Memory_Address_4G, bool, FALSE] (virt_to_phys_range(s' , restrict [Address, Mem- ory_Address_4G, boolean] (pm'ro_addr), (singleton(Read) \cup singleton(Execute))))))	

Simplifying, rewriting, and recording with decision procedures,

Using lemma unchanged_memory_invariant_mono,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 2 subgoals:

linear_plain_unchanged_memory_invariant_write.2.4.1.1:

{-1}	unchanged_memory_invariant?(pm_phy'mem, pm_phy'states,	memory_write_transformers(pm_phy'mem, pm_phy'rw_addr)	
		pm_phy'ro_addr)	
{-2}	unchanged_memory_invariant?(pm_phy'mem, pm_phy'states,	singleton(expr_2_super(memory_write(pm_phy'mem)(d', b'))),	
		pm_phy'ro_addr)	
{-3}	pm_phy'rw_addr(d')		
{-4}	plain_memory?(pm_phy)		
{-5}	unchanged_memory_invariant?(pm_phy'mem, pm_phy'states,	singleton(expr_2_super(memory_write(pm_phy'mem)(d', b'))),	
		(pm_phy'rw_addr \ {d'}))	
{-6}	is_linear_plain_memory?(pm')		
{-7}	pm' 'states = pm_phy'states		
{-8}	data(linear_resolve(a', Write)(s')) = d'		
{-9}	Mem?(type_of(a'))		
{-10}	0 ≤ a' 'offset		
{-11}	a' 'type_of = a' 'type_of		
{-12}	a' 'offset < max_linear_offset		
{-13}	b' < max_byte		
{-14}	pm' 'rw_addr(a')		
{-15}	pm' 'states(s')		
{1}	unchanged_memory_invariant?(pm_phy'mem, pm_phy'states,	singleton(expr_2_super(memory_write(pm_phy'mem)(d', b'))),	
		extend[Address, Memory_Address_4G, bool, FALSE]	
		(virt_to_phys_range(s',	
		restrict	
		[Address, Mem-	
	ory_Address_4G, boolean]	(pm' 'ro_addr),	
		(singleton(Read) ∪ singleton(Ex	

Hiding formulas: -1,

Using lemma unchanged_memory_invariant_union_addresses,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Hiding formulas: (-1 -2),

Using lemma unchanged_memory_invariant_mono,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Hiding formulas: (-1 2),

Using lemma pm_linear_blessed,

Expanding the definition of linear_blessed?,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Hiding formulas: (-1 -2 -4 -5 -6),

Installing automatic rewrites from: subset? member union remove

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Repeatedly Skolemizing and flattening,

Expanding the definition of extend,

Instantiating quantified variables,

we get 2 subgoals:

linear_plain_unchanged_memory_invariant_write.2.4.1.1.1:

<pre> {-1} virt_to_phys_range(s', (restrict[Address, Memory_Address_4G, boolean](pm'ro_addr) ∪ restrict [Address, (singleton(Read) ∪ singleton(Execute))) (x')) ⇒ restrict[Address, Memory_Address_4G, boolean]((pm_phy'ro_addr ∪ pm_phy'rw_addr))(x') {-2} ∀ (a₁, a₂: Memory_Address_4G): restrict[Address, Memory_Address_4G, boolean](pm'rw_addr)(a₁) ∧ (restrict[Address, Memory_Address_4G, boolean](pm'ro_addr)(a₂) ∨ restrict[Address, Memory_Address_4G, boolean](pm'rw_addr)(a₂)) ⊃ ¬ linear_shared?(s')(a₁, a₂) {-3} IF Mem?(x'type_of) ∧ 0 ≤ x'offset ∧ x'offset < max_linear_offset THEN virt_to_phys_range(s', restrict[Address, Memory_Address_4G, boolean](pm'ro_addr), (singleton(Read) ∪ singleton(Execute))) (x') ELSE FALSE ENDIF {-4} pm_phy'rw_addr(d') {-5} plain_memory?(pm_phy) {-6} is_linear_plain_memory?(pm') {-7} pm'states = pm_phy'states {-8} data(linear_resolve(a', Write)(s')) = d' {-9} Mem?(type_of(a')) {-10} 0 ≤ a'offset {-11} a'type_of = a'type_of {-12} a'offset < max_linear_offset {-13} b' < max_byte {-14} pm'rw_addr(a') {-15} pm'states(s') </pre>	<pre> {1} pm_phy'ro_addr(x') {2} d' ≠ x' ∧ pm_phy'rw_addr(x') </pre>
---	--

Expanding the definition of restrict,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 3 subgoals:

C Proof scripts

linear_plain_unchanged_memory_invariant_write.2.4.1.1.1.1:

{-1}	pm_phy'rw_addr(x')
{-2}	$\forall (a_1, a_2: \text{Memory_Address_4G}):$ $\text{pm}'\text{rw_addr}(a_1) \wedge (\text{pm}'\text{ro_addr}(a_2) \vee \text{pm}'\text{rw_addr}(a_2)) \supset$ $\neg \text{linear_shared?}(s')(a_1, a_2)$
{-3}	Mem?(x' type_of)
{-4}	$0 \leq x'\text{offset}$
{-5}	$x'\text{offset} < \text{max_linear_offset}$
{-6}	$\text{virt_to_phys_range}(s', \lambda (s: \text{Memory_Address_4G}): \text{pm}'\text{ro_addr}(s),$ $(\text{singleton}(\text{Read}) \cup \text{singleton}(\text{Execute})))$ (x')
{-7}	pm_phy'rw_addr(d')
{-8}	plain_memory?(pm_phy)
{-9}	is_linear_plain_memory?(pm')
{-10}	pm' states = pm_phy' states
{-11}	$\text{data}(\text{linear_resolve}(a', \text{Write})(s')) = d'$
{-12}	Mem?(type_of(a'))
{-13}	$0 \leq a'\text{offset}$
{-14}	$a'\text{type_of} = a'\text{type_of}$
{-15}	$a'\text{offset} < \text{max_linear_offset}$
{-16}	$b' < \text{max_byte}$
{-17}	pm' rw_addr(a')
{-18}	pm' states(s')
{1}	pm_phy'ro_addr(x')
{2}	$d' \neq x'$

Expanding the definition of virt_to_phys_range,

Repeatedly Skolemizing and flattening,

Instantiating the top quantifier in -7 with the terms: (a!1 lin_a!1),

Expanding the definition of linear_shared?,

Instantiating the top quantifier in 2 with the terms: (Write ac!1),

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of linear_plain_unchanged_memory_invariant_write.2.4.1.1.1.1.

linear_plain_unchanged_memory_invariant_write.2.4.1.1.1.2:

<p>{-1} $\forall (a_1, a_2: \text{Memory_Address_4G}):$ $\text{pm}'\text{rw_addr}(a_1) \wedge (\text{pm}'\text{ro_addr}(a_2) \vee \text{pm}'\text{rw_addr}(a_2)) \supset$ $\neg \text{linear_shared?}(s')(a_1, a_2)$</p> <p>{-2} $\text{Mem?}(x'\text{type_of})$</p> <p>{-3} $0 \leq x'\text{offset}$</p> <p>{-4} $x'\text{offset} < \text{max_linear_offset}$</p> <p>{-5} $\text{virt_to_phys_range}(s', \lambda (s: \text{Memory_Address_4G}): \text{pm}'\text{ro_addr}(s),$ $(\text{singleton}(\text{Read}) \cup \text{singleton}(\text{Execute})))$ (x')</p> <p>{-6} $\text{pm_phy}'\text{rw_addr}(d')$</p> <p>{-7} $\text{plain_memory?}(\text{pm_phy})$</p> <p>{-8} $\text{is_linear_plain_memory?}(\text{pm}')$</p> <p>{-9} $\text{pm}'\text{states} = \text{pm_phy}'\text{states}$</p> <p>{-10} $\text{data}(\text{linear_resolve}(a', \text{Write})(s')) = d'$</p> <p>{-11} $\text{Mem?}(\text{type_of}(a'))$</p> <p>{-12} $0 \leq a'\text{offset}$</p> <p>{-13} $a'\text{type_of} = a'\text{type_of}$</p> <p>{-14} $a'\text{offset} < \text{max_linear_offset}$</p> <p>{-15} $b' < \text{max_byte}$</p> <p>{-16} $\text{pm}'\text{rw_addr}(a')$</p> <p>{-17} $\text{pm}'\text{states}(s')$</p>	<hr style="border: 0.5px solid black;"/> <p>{1} $\text{virt_to_phys_range}(s',$ $(\lambda (s: \text{Memory_Address_4G}): \text{pm}'\text{ro_addr}(s) \cup \lambda (s: \text{Memory_Address_4G}): \text{pm}$ $(\text{singleton}(\text{Read}) \cup \text{singleton}(\text{Execute})))$ (x')</p> <p>{2} $\text{pm_phy}'\text{ro_addr}(x')$</p> <p>{3} $\text{pm_phy}'\text{rw_addr}(x')$</p>
---	--

Keeping (-5 1) and hiding *,

Expanding the definition of virt_to_phys_range,

Repeatedly Skolemizing and flattening,

Instantiating quantified variables,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of linear_plain_unchanged_memory_invariant_write.2.4.1.1.1.2.

linear_plain_unchanged_memory_invariant_write.2.4.1.1.1.3:

<p>{-1} $\forall (a_1, a_2: \text{Memory_Address_4G}):$ $\text{pm}'\text{rw_addr}(a_1) \wedge (\text{pm}'\text{ro_addr}(a_2) \vee \text{pm}'\text{rw_addr}(a_2)) \supset$ $\neg \text{linear_shared?}(s')(a_1, a_2)$</p> <p>{-2} $\text{Mem?}(x'\text{type_of})$</p> <p>{-3} $0 \leq x'\text{offset}$</p> <p>{-4} $x'\text{offset} < \text{max_linear_offset}$</p> <p>{-5} $\text{virt_to_phys_range}(s', \lambda (s: \text{Memory_Address_4G}): \text{pm}'\text{ro_addr}(s),$ $(\text{singleton}(\text{Read}) \cup \text{singleton}(\text{Execute})))$ (x')</p> <p>{-6} $\text{pm_phy}'\text{rw_addr}(d')$</p> <p>{-7} $\text{plain_memory?}(\text{pm_phy})$</p> <p>{-8} $\text{is_linear_plain_memory?}(\text{pm}')$</p> <p>{-9} $\text{pm}'\text{states} = \text{pm_phy}'\text{states}$</p> <p>{-10} $\text{data}(\text{linear_resolve}(a', \text{Write})(s')) = d'$</p> <p>{-11} $\text{Mem?}(\text{type_of}(a'))$</p> <p>{-12} $0 \leq a'\text{offset}$</p> <p>{-13} $a'\text{type_of} = a'\text{type_of}$</p> <p>{-14} $a'\text{offset} < \text{max_linear_offset}$</p> <p>{-15} $b' < \text{max_byte}$</p> <p>{-16} $\text{pm}'\text{rw_addr}(a')$</p> <p>{-17} $\text{pm}'\text{states}(s')$</p>	<hr style="border: 0.5px solid black;"/> <p>{1} $\text{virt_to_phys_range}(s',$ $(\lambda (s: \text{Memory_Address_4G}): \text{pm}'\text{ro_addr}(s) \cup \lambda (s: \text{Memory_Address_4G}):$ $(\text{singleton}(\text{Read}) \cup \text{singleton}(\text{Execute})))$ (x')</p> <p>{2} $\text{pm_phy}'\text{ro_addr}(x')$</p> <p>{3} $d' \neq x'$</p>
---	--

Keeping (-5 1) and hiding *,

Expanding the definition of virt_to_phys_range,

Repeatedly Skolemizing and flattening,

Instantiating quantified variables,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of linear_plain_unchanged_memory_invariant_write.2.4.1.1.1.3.

linear_plain_unchanged_memory_invariant_write.2.4.1.1.2:

{-1}	$\forall (a_1, a_2: \text{Memory_Address_4G}):$
	$\text{restrict}[\text{Address}, \text{Memory_Address_4G}, \text{boolean}](\text{pm}'\text{'rw_addr})(a_1) \wedge$ $(\text{restrict}[\text{Address}, \text{Memory_Address_4G}, \text{boolean}](\text{pm}'\text{'ro_addr})(a_2) \vee$ $\text{restrict}[\text{Address}, \text{Memory_Address_4G}, \text{boolean}](\text{pm}'\text{'rw_addr})(a_2))$ $\supset \neg \text{linear_shared?}(s')(a_1, a_2)$
{-2}	$\text{IF Mem?}(x'\text{'type_of}) \wedge 0 \leq x'\text{'offset} \wedge x'\text{'offset} < \text{max_linear_offset}$ $\text{THEN virt_to_phys_range}(s', \text{restrict}[\text{Address}, \text{Memory_Address_4G}, \text{boolean}](\text{pm}'\text{'ro_addr}),$ $(\text{singleton}(\text{Read}) \cup \text{singleton}(\text{Execute})))$ (x') ELSE FALSE ENDIF
{-3}	$\text{pm_phy}'\text{'rw_addr}(d')$
{-4}	$\text{plain_memory?}(\text{pm_phy})$
{-5}	$\text{is_linear_plain_memory?}(\text{pm}')$
{-6}	$\text{pm}'\text{'states} = \text{pm_phy}'\text{'states}$
{-7}	$\text{data}(\text{linear_resolve}(a', \text{Write})(s')) = d'$
{-8}	$\text{Mem?}(\text{type_of}(a'))$
{-9}	$0 \leq a'\text{'offset}$
{-10}	$a'\text{'type_of} = a'\text{'type_of}$
{-11}	$a'\text{'offset} < \text{max_linear_offset}$
{-12}	$b' < \text{max_byte}$
{-13}	$\text{pm}'\text{'rw_addr}(a')$
{-14}	$\text{pm}'\text{'states}(s')$
<hr/>	
{1}	$\text{Mem?}(x'\text{'type_of}) \wedge 0 \leq x'\text{'offset} \wedge x'\text{'offset} < \text{max_linear_offset}$
{2}	$\text{pm_phy}'\text{'ro_addr}(x')$
{3}	$d' \neq x' \wedge \text{pm_phy}'\text{'rw_addr}(x')$

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of linear_plain_unchanged_memory_invariant_write.2.4.1.1.2.

C Proof scripts

linear_plain_unchanged_memory_invariant_write.2.4.1.2:

{-1}	unchanged_memory_invariant?(pm_phy'mem, pm_phy'states, memory_write_transformers(pm_phy'mem, pm_phy'rw_addr), pm_phy'ro_addr)
{-2}	pm_phy'rw_addr(d')
{-3}	plain_memory?(pm_phy)
{-4}	unchanged_memory_invariant?(pm_phy'mem, pm_phy'states, singleton(expr_2_super(memory_write(pm_phy'mem)(d' , b'))), (pm_phy'rw_addr \ { d' }))
{-5}	is_linear_plain_memory?(pm')
{-6}	pm' 'states = pm_phy'states
{-7}	data(linear_resolve(a' , Write)(s')) = d'
{-8}	Mem?(type_of(a'))
{-9}	$0 \leq a'$ 'offset
{-10}	a' 'type_of = a' 'type_of
{-11}	a' 'offset < max_linear_offset
{-12}	$b' < \text{max_byte}$
{-13}	pm' 'rw_addr(a')
{-14}	pm' 'states(s')
{1}	(singleton(expr_2_super(memory_write(pm_phy'mem)(d' , b'))) \subseteq memory_write_transformers(pm...
{2}	unchanged_memory_invariant?(pm_phy'mem, pm_phy'states, singleton(expr_2_super(memory_write(pm_phy'mem)(d' , b'))), extend[Address, Memory_Address_4G, bool, FALSE] (virt_to_phys_range(s' , restrict [Address, Mem- ory_Address_4G, boolean] (pm' 'ro_addr), (singleton(Read) \cup singleton(Ex

Keeping (-2 1) and hiding *,

Expanding the definition of subset?,

Repeatedly Skolemizing and flattening,

Expanding the definition of member,

Expanding the definition of singleton,

Replacing using formula -1,

Expanding the definition of memory_write_transformers,

Instantiating quantified variables,

This completes the proof of linear_plain_unchanged_memory_invariant_write.2.4.1.2.

linear_plain_unchanged_memory_invariant_write.2.4.2:

{-1}	plain_memory?(pm_phy)	
{-2}	pm_phy'rw_addr(d') \supset unchanged_memory_invariant?(pm_phy'mem, pm_phy'states, singleton(expr_2_super(memory_write(pm_phy'mem)(d', b'))), (pm_phy'rw_addr \ {d'}))	
{-3}	unchanged_memory_invariant?(pm_phy'mem, pm_phy'states, memory_write_transformers(pm_phy'mem, pm_phy'rw_addr), pm_phy'ro_addr)	
{-4}	is_linear_plain_memory?(pm')	
{-5}	pm' states = pm_phy' states	
{-6}	data(linear_resolve(a', Write)(s')) = d'	
{-7}	Mem?(type_of(a'))	
{-8}	0 \leq a'offset	
{-9}	a'type_of = a'type_of	
{-10}	a'offset < max_linear_offset	
{-11}	b' < max_byte	
{-12}	pm'rw_addr(a')	
{-13}	pm' states(s')	
{1}	pm_phy'rw_addr(d')	
{2}	unchanged_memory_invariant?(pm_phy'mem, pm_phy'states, singleton(expr_2_super(memory_write(pm_phy'mem)(d', b'))), extend[Address, Memory_Address_4G, bool, FALSE] (virt_to_phys_range(s', restrict [Address, Mem- (pm'ro_addr), (singleton(Read) \cup singleton(Execute)))))	

Hiding formulas: (-2 -3 2),

Using lemma pm_linear_blessed,

Expanding the definition of linear_blessed?,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Keeping (-2 -4 -11 -17 1) and hiding *,

Expanding the definition of subset?,

Instantiating quantified variables,

Instantiating the top quantifier in -2 with the terms: d',

we get 2 subgoals:

linear_plain_unchanged_memory_invariant_write.2.4.2.1:

{-1}	restrict[Address, Memory_Address_4G, boolean](pm'rw_addr(a')) \supset OK?(linear_resolve(a', Write)(s'))
{-2}	(d' \in virt_to_phys_range(s', restrict [Address, Memory_Address_4G, boolean](pm'rw_addr), singleton(Write))) \Rightarrow (d' \in restrict[Address, Memory_Address_4G, boolean](pm_phy'rw_addr))
{-3}	data(linear_resolve(a', Write)(s')) = d'
{-4}	pm'rw_addr(a')
{1}	pm_phy'rw_addr(d')

Expanding the definition of member,

Expanding the definition of restrict,

Expanding the definition of virt_to_phys_range,

Instantiating quantified variables,

C Proof scripts

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `linear_plain_unchanged_memory_invariant_write.2.4.2.1`.
`linear_plain_unchanged_memory_invariant_write.2.4.2.2`:

{-1}	<code>restrict[Address, Memory_Address_4G, boolean](pm'rw_addr)(a') \supset</code>
	<code>OK?(linear_resolve(a', Write)(s'))</code>
{-2}	<code>data(linear_resolve(a', Write)(s')) = d'</code>
{-3}	<code>pm'rw_addr(a')</code>
{1}	<code>Mem?(d' type_of) \wedge 0 \leq d'offset \wedge d'offset < max_linear_offset</code>
{2}	<code>pm_phy'rw_addr(d')</code>

Using lemma `linear_resolve_memory_address`,
Expanding the definition of every,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `linear_plain_unchanged_memory_invariant_write.2.4.2.2`.
`linear_plain_unchanged_memory_invariant_write.3`:

{-1}	<code>is_linear_plain_memory?(pm')</code>
{1}	<code>(restrict[Address, Memory_Address_4G, boolean](pm'ro_addr) \subseteq restrict [Address, Memory_Ad</code>
{2}	<code>unchanged_memory_invariant?(pm'mem, pm'states,</code> <code>memory_write_transformers(pm'mem, pm'rw_addr), pm'ro</code>

Keeping (1) and hiding *,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `linear_plain_unchanged_memory_invariant_write.3`.
Q.E.D.

C.116.4 Linear_Memory_Blessing_Properties.linear_plain_unchanged_memory_write_invar

Terse proof for `linear_plain_unchanged_memory_write_invariant`.

`linear_plain_unchanged_memory_write_invariant`:

{1}	\forall (pm: Plain_Memory[Physical_memory]): <code>is_linear_plain_memory?(pm) \supset</code> <code>unchanged_memory_write_invariant?(pm'mem, pm'states, pm'rw_addr)</code>
-----	--

Repeatedly Skolemizing and flattening,
Installing automatic rewrites from: (`expr_2_super!` `expr_2_super_res!` `##!` `has_next_state!`
`singleton!` `max_linear!` `min_linear!` `Address_Helpers.<=!` `reg_size!` `reg_base!` `Mem! <!`
`pm_phy_read_after_resolve_ok!` `pm_resolve_address!` `union_right!` `in_memory!` `subset_reflexive!`)
Expanding the definition of `unchanged_memory_write_invariant?`,
Repeatedly Skolemizing and flattening,
Using lemma `linear_unchanged_invariant`,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
we get 3 subgoals:

linear_plain_unchanged_memory_write_invariant.1:

{-1}	is_linear_plain_memory?(pm')
{-2}	unchanged_memory_invariant?(pm' mem, pm' states, singleton(expr_2_super(memory_write(pm' mem)(waddr', b'))), extend[Address, Memory_Address_4G, bool, FALSE] (restrict[Address, Mem- ory_Address_4G, boolean] ((pm'rw_addr \ {waddr'}))))
{-3}	b' < max_byte
{-4}	pm'rw_addr(waddr')
{1}	unchanged_memory_invariant?(pm' mem, pm' states, singleton(expr_2_super(memory_write(pm' mem)(waddr', b'))), (pm'rw_addr \ {waddr'}))

Case splitting on extend[Address, Memory_Address_4G, bool, FALSE] (restrict[Address, Memory_Address_4G, boolean] (remove(waddr!1, pm!1'rw_addr))) = remove(waddr!1, pm!1'rw_addr), we get 2 subgoals:

linear_plain_unchanged_memory_write_invariant.1.1:

{-1}	extend[Address, Memory_Address_4G, bool, FALSE] (restrict[Address, Memory_Address_4G, boolean]((pm'rw_addr \ {waddr'}))) = (pm'rw_addr \ {waddr'})
{-2}	is_linear_plain_memory?(pm')
{-3}	unchanged_memory_invariant?(pm' mem, pm' states, singleton(expr_2_super(memory_write(pm' mem)(waddr', b'))), extend[Address, Memory_Address_4G, bool, FALSE] (restrict[Address, Mem- ory_Address_4G, boolean] ((pm'rw_addr \ {waddr'}))))
{-4}	b' < max_byte
{-5}	pm'rw_addr(waddr')
{1}	unchanged_memory_invariant?(pm' mem, pm' states, singleton(expr_2_super(memory_write(pm' mem)(waddr', b'))), (pm'rw_addr \ {waddr'}))

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting, This completes the proof of linear_plain_unchanged_memory_write_invariant.1.1.

linear_plain_unchanged_memory_write_invariant.1.2:

{-1}	is_linear_plain_memory?(pm')
{-2}	unchanged_memory_invariant?(pm' mem, pm' states, singleton(expr_2_super(memory_write(pm' mem)(waddr', b'))), extend[Address, Memory_Address_4G, bool, FALSE] (restrict[Address, Mem- ory_Address_4G, boolean] ((pm'rw_addr \ {waddr'}))))
{-3}	b' < max_byte
{-4}	pm'rw_addr(waddr')
{1}	extend[Address, Memory_Address_4G, bool, FALSE] (restrict[Address, Memory_Address_4G, boolean]((pm'rw_addr \ {waddr'}))) = (pm'rw_addr \ {waddr'})
{2}	unchanged_memory_invariant?(pm' mem, pm' states, singleton(expr_2_super(memory_write(pm' mem)(waddr', b'))), (pm'rw_addr \ {waddr'}))

C Proof scripts

Applying decompose-equality,

Using lemma pm_memory_addr,

Simplifying, rewriting, and recording with decision procedures,

Keeping (-1 1) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `linear_plain_unchanged_memory_write_invariant.1.2`.

`linear_plain_unchanged_memory_write_invariant.2`:

<pre>{-1} is_linear_plain_memory?(pm') {-2} b' < max_byte {-3} pm'rw_addr(waddr')</pre>	<hr style="border: 0.5px solid black;"/> <pre>{1} $\forall (s: (pm'states))$: unchanged_memory_invariant?(pm_phy'mem, pm_phy'states, singleton(expr_2_super(memory_write(pm'mem) (waddr', b'))), extend[Address, Memory_Address_4G, bool, FALSE] (virt_to_phys_range(s, restrict [Address, Mem- ory_Address_4G, boolean] ((pm'rw_addr \ {waddr'}), (singleton(Read) \cup singleton(E {2} unchanged_memory_invariant?(pm'mem, pm'states, singleton(expr_2_super(memory_write(pm'mem)(waddr', b'))), (pm'rw_addr \ {waddr'}))</pre>
--	---

Hiding formulas: 2,

Repeatedly Skolemizing and flattening,

Rewriting using pm_write_linear, matching in *,

Using lemma pm_plain_phy,

Using lemma pm_states,

Expanding the definition of linear_write,

Using lemma pm_memory_addr,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Rewriting using `plain_memory_unchanged_composition[Physical_memory, Address, Unit]`, matching in * where pm gets pm_phy, addresses gets `extend[Address, Memory_Address_4G, bool, FALSE] (virt_to_phys_range(s!1, restrict[Address, Memory_Address_4G, boolean] (remove(waddr!1, pm!1'rw_addr)), union(singleton(Read), singleton(Execute))))`, P gets `LAMBDA (a: Address): FORALL (s: (pm_phy'states)): OK?(linear_resolve(waddr!1, Write)(s)) IMPLIES data(linear_resolve(waddr!1, Write)(s)) = a`,

we get 4 subgoals:

linear_plain_unchanged_memory_write_invariant.2.1:

<pre> {-1} is_linear_plain_memory?(pm') {-2} Mem?(type_of(waddr')) {-3} 0 ≤ waddr'offset {-4} waddr'type_of = waddr'type_of {-5} waddr'offset < max_linear_offset {-6} pm'states = pm_phy'states {-7} plain_memory?(pm_phy) {-8} pm'states(s') {-9} b' < max_byte {-10} pm'rw_addr(waddr') </pre>	<pre> {1} (extend [Address, Memory_Address_4G, bool, FALSE](virt_to_phys_range(s', restrict [Address, Memory_ {2} unchanged_memory_invariant?(pm_phy'mem, pm_phy'states, singleton(expr_2_super(linear_resolve(waddr', Write) ## (λ (pa: Address): mem- ory_write(pm_phy'mem) (pa, b')))), extend[Address, Memory_Address_4G, bool, FALSE] (virt_to_phys_range(s', restrict [Address, Mem- ory_Address_4G, boolean] ((pm'rw_addr \ {waddr'})), (singleton(Read) ∪ singleton(Execute)))))) </pre>
---	--

Hiding formulas: 2,

Using lemma pm_linear_blessed,

Expanding the definition of linear_blessed?,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Keeping (-3 -8 1) and hiding *,

Expanding the definition of subset?,

Repeatedly Skolemizing and flattening,

Expanding the definition of member,

Expanding the definition of extend,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Instantiating quantified variables,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 2 subgoals:

linear_plain_unchanged_memory_write_invariant.2.1.1:

<pre> {-1} restrict[Address, Memory_Address_4G, boolean]((pm_phy'ro_addr ∪ pm_phy'rw_addr)(x') {-2} Mem?(x'type_of) {-3} 0 ≤ x'offset {-4} x'offset < max_linear_offset {-5} virt_to_phys_range(s', restrict[Address, Memory_Address_4G, boolean] ((pm'rw_addr \ {waddr'})), (singleton(Read) ∪ singleton(Execute))) (x') {-6} is_linear_plain_memory?(pm') {1} union(pm_phy'ro_addr, pm_phy'rw_addr)(x') </pre>

C Proof scripts

Expanding the definition of `restrict`,
which is trivially true.

This completes the proof of `linear_plain_unchanged_memory_write_invariant.2.1.1`.

`linear_plain_unchanged_memory_write_invariant.2.1.2`:

<pre>{-1} Mem?(x' 'type_of) {-2} 0 ≤ x' 'offset {-3} x' 'offset < max_linear_offset {-4} virt_to_phys_range(s',</pre>	<pre>restrict[Address, Memory_Address_4G, boolean] ((pm' 'rw_addr \ {waddr'})), (singleton(Read) ∪ singleton(Execute))) (x')</pre>
<pre>{-5} is_linear_plain_memory?(pm')</pre>	<pre>(restrict[Address, Memory_Address_4G, boolean](pm' 'ro_addr) ∪ restrict (singleton(Read) ∪ singleton(Execute))) (x')</pre>
<pre>{1} virt_to_phys_range(s',</pre>	<pre>(restrict[Address, Memory_Address_4G, boolean](pm' 'ro_addr) ∪ restrict (singleton(Read) ∪ singleton(Execute))) (x')</pre>
<pre>{2} union(pm_phy' 'ro_addr, pm_phy' 'rw_addr)(x')</pre>	

Keeping (-4 1) and hiding *,

Expanding the definition of `virt_to_phys_range`,

Repeatedly Skolemizing and flattening,

Instantiating quantified variables,

we get 2 subgoals:

`linear_plain_unchanged_memory_write_invariant.2.1.2.1`:

<pre>{-1} Mem?(lin_a' 'type_of) {-2} 0 ≤ lin_a' 'offset {-3} lin_a' 'offset < max_linear_offset {-4} restrict[Address, Memory_Address_4G, boolean]((pm' 'rw_addr \ {waddr'}))(lin_a')</pre>	<pre>union(singleton(Read), singleton(Execute))(ac')</pre>
<pre>{-6} OK?(linear_resolve(lin_a', ac')(s'))</pre>	<pre>data(linear_resolve(lin_a', ac')(s')) = x'</pre>
<pre>{-7} data(linear_resolve(lin_a', ac')(s')) = x'</pre>	<pre>OK?(linear_resolve(lin_a', ac')(s')) ∧ data(linear_resolve(lin_a', ac')(s')) = x'</pre>

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `linear_plain_unchanged_memory_write_invariant.2.1.2.1`.

`linear_plain_unchanged_memory_write_invariant.2.1.2.2`:

<pre>{-1} Mem?(lin_a' 'type_of) {-2} 0 ≤ lin_a' 'offset {-3} lin_a' 'offset < max_linear_offset {-4} restrict[Address, Memory_Address_4G, boolean]((pm' 'rw_addr \ {waddr'}))(lin_a')</pre>	<pre>union(singleton(Read), singleton(Execute))(ac')</pre>
<pre>{-6} OK?(linear_resolve(lin_a', ac')(s'))</pre>	<pre>data(linear_resolve(lin_a', ac')(s')) = x'</pre>
<pre>{-7} data(linear_resolve(lin_a', ac')(s')) = x'</pre>	<pre>union[Memory_Address_4G] (restrict[Address, Memory_Address_4G, boolean](pm' 'ro_addr), restrict[Address, Memory_Address_4G, boolean](pm' 'rw_addr)) (lin_a')</pre>

Keeping (-4 1) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `linear_plain_unchanged_memory_write_invariant.2.1.2.2`.

`linear_plain_unchanged_memory_write_invariant.2.2:`

<pre> {-1} is_linear_plain_memory?(pm') {-2} Mem?(type_of(waddr')) {-3} 0 ≤ waddr'.offset {-4} waddr'.type_of = waddr'.type_of {-5} waddr'.offset < max_linear_offset {-6} pm'.states = pm_phy'.states {-7} plain_memory?(pm_phy) {-8} pm'.states(s') {-9} b' < max_byte {-10} pm'.rw_addr(waddr') </pre>	<hr/> <pre> {1} ∀ (s_1: (pm_phy'.states)): OK?(linear_resolve(waddr', Write)(s_1)) ⊃ (∀ (s: (pm_phy'.states)): OK?(linear_resolve(waddr', Write)(s)) ⊃ data(linear_resolve(waddr', Write)(s)) = data(linear_resolve(waddr', Write)(s_1))) {2} unchanged_memory_invariant?(pm_phy'.mem, pm_phy'.states, singleton(expr_2_super(linear_resolve(waddr', Write) ## (λ (pa: Address): mem- (pa, b')))), extend[Address, Memory_Address_4G, bool, FALSE] (virt_to_phys_range(s', restrict [Address, Mem- ((pm'.rw_addr \ {waddr'})), (singleton(Read) ∪ singleton(Execute))))) </pre>
---	---

Hiding formulas: 2,

Repeatedly Skolemizing and flattening,

Using lemma `linear_resolve_same_result`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `linear_plain_unchanged_memory_write_invariant.2.2`.

linear_plain_unchanged_memory_write_invariant.2.3:

{-1}	is_linear_plain_memory?(pm')	
{-2}	Mem?(type_of(waddr'))	
{-3}	$0 \leq \text{waddr}'.\text{offset}$	
{-4}	waddr'.type_of = waddr'.type_of	
{-5}	waddr'.offset < max_linear_offset	
{-6}	pm'.states = pm_phy.states	
{-7}	plain_memory?(pm_phy)	
{-8}	pm'.states(s')	
{-9}	$b' < \text{max_byte}$	
{-10}	pm'.rw_addr(waddr')	
{1}	unchanged_memory_invariant?(pm_phy.mem, pm_phy.states, singleton(expr_2_super(linear_resolve(waddr', Write))), extend[Address, Memory_Address_4G, bool, FALSE] (virt_to_phys_range(s', restrict [Address, Mem- ory_Address_4G, boolean] ((pm'.rw_addr \ {waddr'}))) (singleton(Read) ∪ singleton(Ex	
{2}	unchanged_memory_invariant?(pm_phy.mem, pm_phy.states, singleton(expr_2_super(linear_resolve(waddr', Write) ## (λ (pa: Address): mem- (pa, b')))), extend[Address, Memory_Address_4G, bool, FALSE] (virt_to_phys_range(s', restrict [Address, Mem- ory_Address_4G, boolean] ((pm'.rw_addr \ {waddr'}))) (singleton(Read) ∪ singleton(Ex	

Hiding formulas: 2,

Using lemma linear_resolve_unchanged_pm_phy,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Replacing using formula -7,

Using lemma unchanged_memory_invariant_mono,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Hiding formulas: (-1 2),

Using lemma pm_linear_blessed,

Expanding the definition of linear_blessed?,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Keeping (-3 -6 1) and hiding *,

Installing automatic rewrites from: (empty?! member! subset?! extend! difference! disjoint?! intersection!)

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Repeatedly Skolemizing and flattening,

Instantiating quantified variables,

we get 2 subgoals:

linear_plain_unchanged_memory_write_invariant.2.3.1:

{-1}	$\text{virt_to_phys_range}(s',$ $\quad (\text{restrict}[\text{Address}, \text{Memory_Address_4G}, \text{boolean}](\text{pm}'\text{ro_addr}) \cup \text{restrict} [\text{Address},$ $\quad (\text{singleton}(\text{Read}) \cup \text{singleton}(\text{Execute})))$ $\quad (x')$ $\Rightarrow \text{restrict}[\text{Address}, \text{Memory_Address_4G}, \text{boolean}]((\text{pm_phy}'\text{ro_addr} \cup \text{pm_phy}'\text{rw_addr}))(x')$
{-2}	$\forall (x: \text{Memory_Address_4G}):$ $\quad \neg (\text{virt_to_phys_range}(s',$ $\quad (\text{restrict} [\text{Address}, \text{Memory_Address_4G}, \text{boolean}](\text{pm}'\text{ro_addr}) \cup \text{restrict} [\text{A}$ $\quad (x)$ $\quad \wedge$ $\quad \text{address_in_pt_range?}(s',$ $\quad (\text{restrict} [\text{Address}, \text{Memory_Address_4G}, \text{boolean}](\text{pm}'\text{ro_addr}) \cup \text{restrict}$ $\quad (x))$
{-3}	$\text{IF Mem?}(x'\text{'type_of}) \wedge 0 \leq x'\text{'offset} \wedge x'\text{'offset} < \text{max_linear_offset}$ $\quad \text{THEN virt_to_phys_range}(s',$ $\quad \text{restrict}[\text{Address}, \text{Memory_Address_4G}, \text{boolean}]$ $\quad ((\text{pm}'\text{rw_addr} \setminus \{\text{waddr}'\})),$ $\quad (\text{singleton}(\text{Read}) \cup \text{singleton}(\text{Execute})))$ $\quad (x')$ $\quad \text{ELSE FALSE}$ $\quad \text{ENDIF}$
{1}	<hr/> $\text{union}(\text{pm_phy}'\text{ro_addr}, \text{pm_phy}'\text{rw_addr})(x') \wedge$ $\quad \neg \text{IF Mem?}(x'\text{'type_of}) \wedge 0 \leq x'\text{'offset} \wedge x'\text{'offset} < \text{max_linear_offset}$ $\quad \text{THEN address_in_pt_range?}(s',$ $\quad \text{restrict}[\text{Address}, \text{Memory_Address_4G}, \text{boolean}]$ $\quad ((\text{pm}'\text{ro_addr} \cup \text{pm}'\text{rw_addr}))$ $\quad (x')$ $\quad \text{ELSE FALSE}$ $\quad \text{ENDIF}$

Expanding the definition of restrict,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 we get 2 subgoals:

linear_plain_unchanged_memory_write_invariant.2.3.1.1:

{-1}	$\text{union}(\text{pm_phy}'\text{ro_addr}, \text{pm_phy}'\text{rw_addr})(x')$
{-2}	$\forall (x: \text{Memory_Address_4G}):$ $\quad \neg (\text{virt_to_phys_range}(s',$ $\quad (\lambda (s: \text{Memory_Address_4G}): \text{pm}'\text{ro_addr}(s) \cup \lambda (s: \text{Memory_Address_4G})$ $\quad (x)$ $\quad \wedge$ $\quad \text{address_in_pt_range?}(s',$ $\quad (\lambda (s: \text{Memory_Address_4G}): \text{pm}'\text{ro_addr}(s) \cup \lambda (s: \text{Memory_Address}$ $\quad (x))$
{-3}	$\text{Mem?}(x'\text{'type_of})$
{-4}	$0 \leq x'\text{'offset}$
{-5}	$x'\text{'offset} < \text{max_linear_offset}$
{-6}	$\text{virt_to_phys_range}(s', \lambda (s: \text{Memory_Address_4G}): \text{remove}(\text{waddr}', \text{pm}'\text{rw_addr})(s),$ $\quad (\text{singleton}(\text{Read}) \cup \text{singleton}(\text{Execute})))$ $\quad (x')$
{-7}	$\text{address_in_pt_range?}(s', \lambda (s: \text{Memory_Address_4G}): \text{union}(\text{pm}'\text{ro_addr}, \text{pm}'\text{rw_addr})(s))$ $\quad (x')$ <hr/>

C Proof scripts

Instantiating quantified variables,

Applying propositional simplification,

we get 2 subgoals:

`linear_plain_unchanged_memory_write_invariant.2.3.1.1.1:`

<pre> {-1} union(pm_phy'ro_addr, pm_phy'rw_addr)(x') {-2} Mem?(x' type_of) {-3} 0 ≤ x' offset {-4} x' offset < max_linear_offset {-5} virt_to_phys_range(s', λ (s : Memory_Address_4G): remove(waddr', pm'rw_addr)(s), (singleton(Read) ∪ singleton(Execute))) (x') {-6} address_in_pt_range?(s', λ (s : Memory_Address_4G): union(pm'ro_addr, pm'rw_addr)(s)) (x') </pre>	<pre> {1} virt_to_phys_range(s', (λ (s : Memory_Address_4G) : pm'ro_addr(s) ∪ λ (s : Memory_Address_4G) : pm'rw_addr(s))) (x') </pre>
---	---

Expanding the definition of `virt_to_phys_range`,

Expanding the definition of `virt_to_phys_range`,

Repeatedly Skolemizing and flattening,

Instantiating quantified variables,

we get 3 subgoals:

`linear_plain_unchanged_memory_write_invariant.2.3.1.1.1.1:`

<pre> {-1} Mem?(lin_a' type_of) {-2} 0 ≤ lin_a' offset {-3} lin_a' offset < max_linear_offset {-4} remove(waddr', pm'rw_addr)(lin_a') {-5} union(singleton(Read), singleton(Execute))(ac') {-6} union(pm_phy'ro_addr, pm_phy'rw_addr)(x') {-7} Mem?(x' type_of) {-8} 0 ≤ x' offset {-9} x' offset < max_linear_offset {-10} OK?(linear_resolve(lin_a', ac')(s')) {-11} data(linear_resolve(lin_a', ac')(s')) = x' {-12} address_in_pt_range?(s', λ (s : Memory_Address_4G): union(pm'ro_addr, pm'rw_addr)(s)) (x') </pre>	<pre> {1} OK?(linear_resolve(lin_a', ac')(s')) ∧ data(linear_resolve(lin_a', ac')(s')) = x' </pre>
---	--

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `linear_plain_unchanged_memory_write_invariant.2.3.1.1.1.1`.

linear_plain_unchanged_memory_write_invariant.2.3.1.1.1.2:

{-1}	Mem?(lin_a' type_of)
{-2}	0 ≤ lin_a' offset
{-3}	lin_a' offset < max_linear_offset
{-4}	remove(waddr', pm' rw_addr)(lin_a')
{-5}	union singleton(Read), singleton(Execute))(ac')
{-6}	union(pm_phy' ro_addr, pm_phy' rw_addr)(x')
{-7}	Mem?(x' type_of)
{-8}	0 ≤ x' offset
{-9}	x' offset < max_linear_offset
{-10}	OK?(linear_resolve(lin_a', ac')(s'))
{-11}	data(linear_resolve(lin_a', ac')(s')) = x'
{-12}	address_in_pt_range?(s', λ (s: Memory_Address_4G): union(pm' ro_addr, pm' rw_addr)(s)) (x')
{1}	fullset[Memory_access](ac')

Expanding the definition of fullset,

which is trivially true.

This completes the proof of linear_plain_unchanged_memory_write_invariant.2.3.1.1.1.2.

linear_plain_unchanged_memory_write_invariant.2.3.1.1.1.3:

{-1}	Mem?(lin_a' type_of)
{-2}	0 ≤ lin_a' offset
{-3}	lin_a' offset < max_linear_offset
{-4}	remove(waddr', pm' rw_addr)(lin_a')
{-5}	union singleton(Read), singleton(Execute))(ac')
{-6}	union(pm_phy' ro_addr, pm_phy' rw_addr)(x')
{-7}	Mem?(x' type_of)
{-8}	0 ≤ x' offset
{-9}	x' offset < max_linear_offset
{-10}	OK?(linear_resolve(lin_a', ac')(s'))
{-11}	data(linear_resolve(lin_a', ac')(s')) = x'
{-12}	address_in_pt_range?(s', λ (s: Memory_Address_4G): union(pm' ro_addr, pm' rw_addr)(s)) (x')
{1}	union[Memory_Address_4G] (λ (s: Memory_Address_4G): pm' ro_addr(s), λ (s: Memory_Address_4G): pm' rw_addr(s)) (lin_a')

Keeping (-4 1) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of linear_plain_unchanged_memory_write_invariant.2.3.1.1.1.3.

C Proof scripts

`linear_plain_unchanged_memory_write_invariant.2.3.1.1.2:`

<pre> {-1} union(pm_phy'ro_addr, pm_phy'rw_addr)(x') {-2} Mem?(x''type_of) {-3} 0 ≤ x''offset {-4} x''offset < max_linear_offset {-5} virt_to_phys_range(s', λ (s: Memory_Address_4G): remove(waddr', pm''rw_addr)(s), (singleton(Read) ∪ singleton(Execute))) (x') {-6} address_in_pt_range?(s', λ (s: Memory_Address_4G): union(pm''ro_addr, pm''rw_addr)(s)) (x') </pre>	<pre> {1} address_in_pt_range?(s', (λ (s: Memory_Address_4G): pm''ro_addr(s) ∪ λ (s: Memory_Address_4G): pm''rw_addr(s))) (x') </pre>
--	---

Keeping (-6 1) and hiding *,

Case splitting on $(\text{LAMBDA } (s: \text{Memory_Address_4G}): \text{union}(pm!1'ro_addr, pm!1'rw_addr)(s)) = \text{union}(\text{LAMBDA } (s: \text{Memory_Address_4G}): pm!1'ro_addr(s), \text{LAMBDA } (s: \text{Memory_Address_4G}): pm!1'rw_addr(s))$,

we get 2 subgoals:

`linear_plain_unchanged_memory_write_invariant.2.3.1.1.2.1:`

<pre> {-1} (λ (s: Memory_Address_4G): union(pm''ro_addr, pm''rw_addr)(s)) = (λ (s: Memory_Address_4G): pm''ro_addr(s) ∪ λ (s: Memory_Address_4G): pm''rw_addr(s)) {-2} address_in_pt_range?(s', λ (s: Memory_Address_4G): union(pm''ro_addr, pm''rw_addr)(s)) (x') </pre>	<pre> {1} address_in_pt_range?(s', (λ (s: Memory_Address_4G): pm''ro_addr(s) ∪ λ (s: Memory_Address_4G): pm''rw_addr(s))) (x') </pre>
---	---

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `linear_plain_unchanged_memory_write_invariant.2.3.1.1.2.1`.

`linear_plain_unchanged_memory_write_invariant.2.3.1.1.2.2:`

<pre> {-1} address_in_pt_range?(s', λ (s: Memory_Address_4G): union(pm''ro_addr, pm''rw_addr)(s)) (x') </pre>	<pre> {1} (λ (s: Memory_Address_4G): union(pm''ro_addr, pm''rw_addr)(s)) = (λ (s: Memory_Address_4G): pm''ro_addr(s) ∪ λ (s: Memory_Address_4G): pm''rw_addr(s)) {2} address_in_pt_range?(s', (λ (s: Memory_Address_4G): pm''ro_addr(s) ∪ λ (s: Memory_Address_4G): pm''rw_addr(s))) (x') </pre>
---	--

Applying decompose-equality,

Hiding formulas: (-1 2),

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `linear_plain_unchanged_memory_write_invariant.2.3.1.1.2.2`.

linear_plain_unchanged_memory_write_invariant.2.3.1.2:

<p>{-1} $\forall (x: \text{Memory_Address_4G}):$ $\neg (\text{virt_to_phys_range}(s',$ $\quad (\lambda (s: \text{Memory_Address_4G}): \text{pm}'\text{ro_addr}(s) \cup \lambda (s: \text{Memory_Address_4G}):$ $\quad \quad (x)$ $\quad \quad \wedge$ $\quad \quad \text{address_in_pt_range?}(s',$ $\quad \quad \quad (\lambda (s: \text{Memory_Address_4G}): \text{pm}'\text{ro_addr}(s) \cup \lambda (s: \text{Memory_Address_4G}):$ $\quad \quad \quad \quad (x))$</p> <p>{-2} $\text{Mem?}(x'\text{'type_of})$ {-3} $0 \leq x'\text{'offset}$ {-4} $x'\text{'offset} < \text{max_linear_offset}$ {-5} $\text{virt_to_phys_range}(s', \lambda (s: \text{Memory_Address_4G}): \text{remove}(\text{waddr}', \text{pm}'\text{rw_addr})(s),$ $\quad \quad \quad (\text{singleton}(\text{Read}) \cup \text{singleton}(\text{Execute})))$ $\quad \quad \quad (x')$</p>	<p>{1} $\text{virt_to_phys_range}(s',$ $\quad (\lambda (s: \text{Memory_Address_4G}): \text{pm}'\text{ro_addr}(s) \cup \lambda (s: \text{Memory_Address_4G}): \text{pm}'\text{rw_addr}(s),$ $\quad \quad (\text{singleton}(\text{Read}) \cup \text{singleton}(\text{Execute})))$ $\quad \quad (x')$</p> <p>{2} $\text{union}(\text{pm_phy}'\text{ro_addr}, \text{pm_phy}'\text{rw_addr})(x')$</p>
--	--

Keeping (-5 1) and hiding *,

Expanding the definition of virt_to_phys_range,

Repeatedly Skolemizing and flattening,

Instantiating quantified variables,

we get 2 subgoals:

linear_plain_unchanged_memory_write_invariant.2.3.1.2.1:

<p>{-1} $\text{Mem?}(\text{lin_a}'\text{'type_of})$ {-2} $0 \leq \text{lin_a}'\text{'offset}$ {-3} $\text{lin_a}'\text{'offset} < \text{max_linear_offset}$ {-4} $\text{remove}(\text{waddr}', \text{pm}'\text{rw_addr})(\text{lin_a}')$ {-5} $\text{union}(\text{singleton}(\text{Read}), \text{singleton}(\text{Execute}))(\text{ac}')$ {-6} $\text{OK?}(\text{linear_resolve}(\text{lin_a}', \text{ac}')(s'))$ {-7} $\text{data}(\text{linear_resolve}(\text{lin_a}', \text{ac}')(s')) = x'$</p>	<p>{1} $\text{OK?}(\text{linear_resolve}(\text{lin_a}', \text{ac}')(s')) \wedge$ $\text{data}(\text{linear_resolve}(\text{lin_a}', \text{ac}')(s')) = x'$</p>
---	--

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of linear_plain_unchanged_memory_write_invariant.2.3.1.2.1.

linear_plain_unchanged_memory_write_invariant.2.3.1.2.2:

<p>{-1} $\text{Mem?}(\text{lin_a}'\text{'type_of})$ {-2} $0 \leq \text{lin_a}'\text{'offset}$ {-3} $\text{lin_a}'\text{'offset} < \text{max_linear_offset}$ {-4} $\text{remove}(\text{waddr}', \text{pm}'\text{rw_addr})(\text{lin_a}')$ {-5} $\text{union}(\text{singleton}(\text{Read}), \text{singleton}(\text{Execute}))(\text{ac}')$ {-6} $\text{OK?}(\text{linear_resolve}(\text{lin_a}', \text{ac}')(s'))$ {-7} $\text{data}(\text{linear_resolve}(\text{lin_a}', \text{ac}')(s')) = x'$</p>	<p>{1} $\text{union}[\text{Memory_Address_4G}]$ $\quad (\lambda (s: \text{Memory_Address_4G}): \text{pm}'\text{ro_addr}(s),$ $\quad \quad \lambda (s: \text{Memory_Address_4G}): \text{pm}'\text{rw_addr}(s))$ $\quad (\text{lin_a}')$</p>
---	--

C Proof scripts

Keeping (-4 1) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `linear_plain_unchanged_memory_write_invariant.2.3.1.2.2`.

`linear_plain_unchanged_memory_write_invariant.2.3.2`:

```

{-1}  ∀ (x: Memory_Address_4G):
      ¬ (virt_to_phys_range(s',
                           (restrict [Address, Memory_Address_4G, boolean](pm'ro_addr)
                                      (x))
                           ∧
                           address_in_pt_range?(s',
                                                  (restrict [Address, Memory_Address_4G, boolean](pm'ro_addr)
                                                          (x))
                                                  ))
      {2}  IF Mem?(x'type_of) ∧ 0 ≤ x'offset ∧ x'offset < max_linear_offset
          THEN virt_to_phys_range(s',
                                  restrict[Address, Memory_Address_4G, boolean]
                                      ((pm'rw_addr \ {waddr'})),
                                  (singleton(Read) ∪ singleton(Execute)))
          (x')
          ELSE FALSE
      ENDIF
-----
{1}  Mem?(x'type_of) ∧ 0 ≤ x'offset ∧ x'offset < max_linear_offset
{2}  union(pm_phy'ro_addr, pm_phy'rw_addr)(x') ∧
      ¬ IF Mem?(x'type_of) ∧ 0 ≤ x'offset ∧ x'offset < max_linear_offset
          THEN address_in_pt_range?(s',
                                      restrict[Address, Memory_Address_4G, boolean]
                                          ((pm'ro_addr ∪ pm'rw_addr)))
          (x')
          ELSE FALSE
      ENDIF

```

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `linear_plain_unchanged_memory_write_invariant.2.3.2`.

linear_plain_unchanged_memory_write_invariant.2.4:

{-1}	is_linear_plain_memory?(pm')
{-2}	Mem?(type_of(waddr'))
{-3}	0 ≤ waddr'offset
{-4}	waddr'type_of = waddr'type_of
{-5}	waddr'offset < max_linear_offset
{-6}	pm'states = pm_phy'states
{-7}	plain_memory?(pm_phy)
{-8}	pm'states(s')
{-9}	b' < max_byte
{-10}	pm'rw_addr(waddr')
{1}	$\forall (d: (\lambda (a: \text{Address}): \forall (s: (\text{pm_phy}'\text{states}): \text{OK}?(\text{linear_resolve}(\text{waddr}', \text{Write})(s) \supset \text{data}(\text{linear_resolve}(\text{waddr}', \text{Write})(s) = a))): \text{unchanged_memory_invariant}?(\text{pm_phy}'\text{mem}, \text{pm_phy}'\text{states}, \text{singleton}(\text{expr_2_super}(\text{memory_write}(\text{pm_phy}'\text{mem})(d, b'))), \text{extend}[\text{Address}, \text{Memory_Address_4G}, \text{bool}, \text{FALSE}] (\text{virt_to_phys_range}(s', \text{restrict} [\text{Address}, \text{Memory_Address_4G}, \text{boolean}] ((\text{pm}'\text{rw_addr} \setminus \{\text{waddr}'\})), (\text{singleton}(\text{Read}) \cup \text{singleton}(\text{Execute}))))))$
{2}	$\text{unchanged_memory_invariant}?(\text{pm_phy}'\text{mem}, \text{pm_phy}'\text{states}, \text{singleton}(\text{expr_2_super}(\text{linear_resolve}(\text{waddr}', \text{Write}) \#\# (\lambda (pa: \text{Address}): \text{mem-}(\text{pa}, b'))), \text{extend}[\text{Address}, \text{Memory_Address_4G}, \text{bool}, \text{FALSE}] (\text{virt_to_phys_range}(s', \text{restrict} [\text{Address}, \text{Memory_Address_4G}, \text{boolean}] ((\text{pm}'\text{rw_addr} \setminus \{\text{waddr}'\})), (\text{singleton}(\text{Read}) \cup \text{singleton}(\text{Execute}))))))$

Hiding formulas: 2,
Repeatedly Skolemizing and flattening,
Instantiating the top quantifier in -1 with the terms: s',
Using lemma pm_linear_resolve_write_ok,
Simplifying, rewriting, and recording with decision procedures,
Using lemma plain_memory_unchanged_memory_invariant_write,
Using lemma plain_memory_unchanged_memory_write_invariant,
Simplifying, rewriting, and recording with decision procedures,
Expanding the definition of unchanged_memory_write_invariant?,
Instantiating quantified variables,
Using lemma unchanged_memory_invariant_union_addresses,
Case splitting on pm_phy'rw_addr(d!1),
we get 2 subgoals:

linear_plain_unchanged_memory_write_invariant.2.4.1:

{-1}	pm_phy'rw_addr(d')	
{-2}	unchanged_memory_invariant?(pm_phy'mem, pm_phy'states, singleton(expr_2_super(memory_write(pm_phy'mem)(d' , b')), (pm_phy'rw_addr \ { d' })))	
	^	
	unchanged_memory_invariant?(pm_phy'mem, pm_phy'states, singleton(expr_2_super(memory_write(pm_phy'mem)(d' , b')), pm_phy'ro_addr)	
	⊃	
	unchanged_memory_invariant?(pm_phy'mem, pm_phy'states, singleton(expr_2_super(memory_write(pm_phy'mem)(d' , b')), ((pm_phy'rw_addr \ { d' }) ∪ pm_phy'ro_addr))	
{-3}	pm_phy'rw_addr(d') ⊃ unchanged_memory_invariant?(pm_phy'mem, pm_phy'states, singleton(expr_2_super(memory_write(pm_phy'mem)(d' , b')), (pm_phy'rw_addr \ { d' })))	
{-4}	unchanged_memory_invariant?(pm_phy'mem, pm_phy'states, memory_write_transformers(pm_phy'mem, pm_phy'rw_addr), pm_phy'ro_addr)	
{-5}	OK?(linear_resolve(waddr', Write)(s'))	
{-6}	data(linear_resolve(waddr', Write)(s')) = d'	
{-7}	is_linear_plain_memory?(pm')	
{-8}	Mem?(type_of(waddr'))	
{-9}	$0 \leq$ waddr'offset	
{-10}	waddr'type_of = waddr'type_of	
{-11}	waddr'offset < max_linear_offset	
{-12}	pm'states = pm_phy'states	
{-13}	plain_memory?(pm_phy)	
{-14}	pm'states(s')	
{-15}	$b' <$ max_byte	
{-16}	pm'rw_addr(waddr')	
{1}	unchanged_memory_invariant?(pm_phy'mem, pm_phy'states, singleton(expr_2_super(memory_write(pm_phy'mem)(d' , b')), extend[Address, Memory_Address_4G, bool, FALSE] (virt_to_phys_range(s' , restrict [Address, Mem- ory_Address_4G, boolean] ((pm'rw_addr \ {waddr'})) (singleton(Read) ∪ singleton(Ex	

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 2 subgoals:

linear_plain_unchanged_memory_write_invariant.2.4.1.1:

{-1}	pm_phy'rw_addr(d')	
{-2}	unchanged_memory_invariant?(pm_phy' mem, pm_phy' states, singleton(expr_2_super(memory_write(pm_phy' mem)(d' , b'))), (pm_phy'rw_addr \ { d' }))	
{-3}	unchanged_memory_invariant?(pm_phy' mem, pm_phy' states, singleton(expr_2_super(memory_write(pm_phy' mem)(d' , b'))), ((pm_phy'rw_addr \ { d' }) \cup pm_phy'ro_addr))	
{-4}	unchanged_memory_invariant?(pm_phy' mem, pm_phy' states, memory_write_transformers(pm_phy' mem, pm_phy'rw_addr), pm_phy'ro_addr)	
{-5}	OK?(linear_resolve(waddr', Write)(s'))	
{-6}	data(linear_resolve(waddr', Write)(s')) = d'	
{-7}	is_linear_plain_memory?(pm')	
{-8}	Mem?(type_of(waddr'))	
{-9}	$0 \leq$ waddr'offset	
{-10}	waddr'type_of = waddr'type_of	
{-11}	waddr'offset < max_linear_offset	
{-12}	pm' states = pm_phy' states	
{-13}	plain_memory?(pm_phy)	
{-14}	pm' states(s')	
{-15}	$b' <$ max_byte	
{-16}	pm'rw_addr(waddr')	
{1}	unchanged_memory_invariant?(pm_phy' mem, pm_phy' states, singleton(expr_2_super(memory_write(pm_phy' mem)(d' , b'))), extend[Address, Memory_Address_4G, bool, FALSE] (virt_to_phys_range(s' , restrict [Address, Mem- ory_Address_4G, boolean] ((pm'rw_addr \ {waddr'})), (singleton(Read) \cup singleton(Execute))))))	

Hiding formulas: (-1 -3),

Using lemma unchanged_memory_invariant_mono,

Rewriting using subset_reflexive, matching in *,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Hiding formulas: (-1 2),

Hiding formulas: (-1 -5 -6 -7 -12),

Using lemma pm_linear_blessed,

Expanding the definition of linear_blessed?,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Keeping (-3 -7 -8 -9 -10 -15 1) and hiding *,

Installing automatic rewrites from: subset? member

Expanding the definition of subset?,

Repeatedly Skolemizing and flattening,

Expanding the definition of extend,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Instantiating quantified variables,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 2 subgoals:

linear_plain_unchanged_memory_write_invariant.2.4.1.1.1:

{-1}	restrict[Address, Memory_Address_4G, boolean]((pm_phy'ro_addr ∪ pm_phy'rw_addr))(x')
{-2}	$\forall (a_1, a_2: \text{Memory_Address_4G}):$ restrict[Address, Memory_Address_4G, boolean](pm'rw_addr)(a ₁) ∧ union(restrict[Address, Memory_Address_4G, boolean](pm'ro_addr), restrict[Address, Memory_Address_4G, boolean](pm'rw_addr)) (a ₂) $\supset \neg \text{linear_shared?}(s')(a_1, a_2)$
{-3}	Mem?(x' type_of)
{-4}	0 ≤ x' offset
{-5}	x' offset < max_linear_offset
{-6}	virt_to_phys_range(s', restrict[Address, Memory_Address_4G, boolean] ((pm'rw_addr \ {waddr'})), (singleton(Read) ∪ singleton(Execute))) (x')
{-7}	OK?(linear_resolve(waddr', Write)(s'))
{-8}	data(linear_resolve(waddr', Write)(s')) = d'
{-9}	is_linear_plain_memory?(pm')
{1}	union((pm_phy'rw_addr \ {d'}), pm_phy'ro_addr)(x')

Expanding the definition of restrict,

Expanding the definition of virt_to_phys_range,

Repeatedly Skolemizing and flattening,

Instantiating the top quantifier in -7 with the terms: (waddr!1 lin_a!1),

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 we get 2 subgoals:

linear_plain_unchanged_memory_write_invariant.2.4.1.1.1.1:

{-1}	Mem?(lin_a' type_of)
{-2}	0 ≤ lin_a' offset
{-3}	lin_a' offset < max_linear_offset
{-4}	remove(waddr', pm'rw_addr)(lin_a')
{-5}	union(singleton(Read), singleton(Execute))(ac')
{-6}	union(pm_phy'ro_addr, pm_phy'rw_addr)(x')
{-7}	Mem?(x' type_of)
{-8}	0 ≤ x' offset
{-9}	x' offset < max_linear_offset
{-10}	OK?(linear_resolve(lin_a', ac')(s'))
{-11}	data(linear_resolve(lin_a', ac')(s')) = x'
{-12}	OK?(linear_resolve(waddr', Write)(s'))
{-13}	data(linear_resolve(waddr', Write)(s')) = d'
{-14}	is_linear_plain_memory?(pm')
{1}	linear_shared?(s')(waddr', lin_a')
{2}	union((pm_phy'rw_addr \ {d'}), pm_phy'ro_addr)(x')

Expanding the definition of linear_shared?,

Instantiating the top quantifier in 1 with the terms: (Write ac!1),

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Replacing using formula -12,

Replacing using formula -13,

Keeping (-6 1 2) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `linear_plain_unchanged_memory_write_invariant.2.4.1.1.1.1`.
`linear_plain_unchanged_memory_write_invariant.2.4.1.1.1.2`:

{-1}	Mem?(lin_a' type_of)
{-2}	$0 \leq \text{lin_a}'\text{offset}$
{-3}	$\text{lin_a}'\text{offset} < \text{max_linear_offset}$
{-4}	remove(waddr', pm'rw_addr)(lin_a')
{-5}	union singleton(Read), singleton(Execute))(ac')
{-6}	union(pm_phy'ro_addr, pm_phy'rw_addr)(x')
{-7}	Mem?(x' type_of)
{-8}	$0 \leq x'\text{offset}$
{-9}	$x'\text{offset} < \text{max_linear_offset}$
{-10}	OK?(linear_resolve(lin_a', ac')(s'))
{-11}	data(linear_resolve(lin_a', ac')(s')) = x'
{-12}	OK?(linear_resolve(waddr', Write)(s'))
{-13}	data(linear_resolve(waddr', Write)(s')) = d'
{-14}	is_linear_plain_memory?(pm')
{1}	union($\lambda (s: \text{Memory_Address_4G}): \text{pm}'\text{ro_addr}(s),$ $\lambda (s: \text{Memory_Address_4G}): \text{pm}'\text{rw_addr}(s)$ $(\text{lin_a}')$)
{2}	union((pm_phy'rw_addr \ {d'}), pm_phy'ro_addr)(x')

Keeping (-4 1) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `linear_plain_unchanged_memory_write_invariant.2.4.1.1.1.2`.
`linear_plain_unchanged_memory_write_invariant.2.4.1.1.2`:

{-1}	$\forall (a_1, a_2: \text{Memory_Address_4G}):$ restrict[Address, Memory_Address_4G, boolean](pm'rw_addr)(a ₁) \wedge union(restrict[Address, Memory_Address_4G, boolean](pm'ro_addr), restrict[Address, Memory_Address_4G, boolean](pm'rw_addr)) (a ₂) $\supset \neg \text{linear_shared?}(s')(a_1, a_2)$
{-2}	Mem?(x' type_of)
{-3}	$0 \leq x'\text{offset}$
{-4}	$x'\text{offset} < \text{max_linear_offset}$
{-5}	virt_to_phys_range(s', restrict[Address, Memory_Address_4G, boolean] ((pm'rw_addr \ {waddr'}), (singleton(Read) \cup singleton(Execute))) (x')
{-6}	OK?(linear_resolve(waddr', Write)(s'))
{-7}	data(linear_resolve(waddr', Write)(s')) = d'
{-8}	is_linear_plain_memory?(pm')
{1}	virt_to_phys_range(s', (restrict[Address, Memory_Address_4G, boolean](pm'ro_addr) \cup restrict [Address, (singleton(Read) \cup singleton(Execute))) (x')
{2}	union((pm_phy'rw_addr \ {d'}), pm_phy'ro_addr)(x')

Keeping (-5 1) and hiding *,

Expanding the definition of virt_to_phys_range,

Repeatedly Skolemizing and flattening,

Instantiating quantified variables,

C Proof scripts

we get 2 subgoals:

`linear_plain_unchanged_memory_write_invariant.2.4.1.1.2.1:`

{-1}	Mem?(lin_a' type_of)
{-2}	$0 \leq \text{lin_a}'\text{'offset}$
{-3}	$\text{lin_a}'\text{'offset} < \text{max_linear_offset}$
{-4}	restrict[Address, Memory_Address_4G, boolean]((pm'rw_addr \ {waddr'})(lin_a')
{-5}	union singleton(Read), singleton(Execute))(ac')
{-6}	OK?(linear_resolve(lin_a', ac')(s'))
{-7}	data(linear_resolve(lin_a', ac')(s')) = x'
{1}	OK?(linear_resolve(lin_a', ac')(s')) \wedge data(linear_resolve(lin_a', ac')(s')) = x'

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `linear_plain_unchanged_memory_write_invariant.2.4.1.1.2.1`.

`linear_plain_unchanged_memory_write_invariant.2.4.1.1.2.2:`

{-1}	Mem?(lin_a' type_of)
{-2}	$0 \leq \text{lin_a}'\text{'offset}$
{-3}	$\text{lin_a}'\text{'offset} < \text{max_linear_offset}$
{-4}	restrict[Address, Memory_Address_4G, boolean]((pm'rw_addr \ {waddr'})(lin_a')
{-5}	union singleton(Read), singleton(Execute))(ac')
{-6}	OK?(linear_resolve(lin_a', ac')(s'))
{-7}	data(linear_resolve(lin_a', ac')(s')) = x'
{1}	union [Memory_Address_4G] (restrict[Address, Memory_Address_4G, boolean](pm'ro_addr), restrict[Address, Memory_Address_4G, boolean](pm'rw_addr)) (lin_a')

Keeping (-4 1) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `linear_plain_unchanged_memory_write_invariant.2.4.1.1.2.2`.

linear_plain_unchanged_memory_write_invariant.2.4.1.2:

{-1}	pm_phy'rw_addr(d')	
{-2}	unchanged_memory_invariant?(pm_phy' mem, pm_phy' states, singleton(expr_2_super(memory_write(pm_phy' mem)(d' , b'))), (pm_phy'rw_addr \ { d' }))	
{-3}	unchanged_memory_invariant?(pm_phy' mem, pm_phy' states, memory_write_transformers(pm_phy' mem, pm_phy' rw_addr), pm_phy' ro_addr)	
{-4}	OK?(linear_resolve(waddr', Write)(s'))	
{-5}	data(linear_resolve(waddr', Write)(s')) = d'	
{-6}	is_linear_plain_memory?(pm')	
{-7}	Mem?(type_of(waddr'))	
{-8}	$0 \leq$ waddr'offset	
{-9}	waddr'type_of = waddr'type_of	
{-10}	waddr'offset < max_linear_offset	
{-11}	pm' states = pm_phy' states	
{-12}	plain_memory?(pm_phy)	
{-13}	pm' states(s')	
{-14}	$b' <$ max_byte	
{-15}	pm' rw_addr(waddr')	
{1}	unchanged_memory_invariant?(pm_phy' mem, pm_phy' states, singleton(expr_2_super(memory_write(pm_phy' mem)(d' , b'))), pm_phy' ro_addr)	
{2}	unchanged_memory_invariant?(pm_phy' mem, pm_phy' states, singleton(expr_2_super(memory_write(pm_phy' mem)(d' , b'))), extend[Address, Memory_Address_4G, bool, FALSE] (virt_to_phys_range(s' , restrict [Address, Mem- ory_Address_4G, boolean] ((pm'rw_addr \ {waddr'})), (singleton(Read) \cup singleton(Execute))))))	

Hiding formulas: (-2 2),

Using lemma unchanged_memory_invariant_mono,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Hiding formulas: (-1 2),

Expanding the definition of subset?,

Repeatedly Skolemizing and flattening,

Expanding the definition of member,

Expanding the definition of memory_write_transformers,

Instantiating quantified variables,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of linear_plain_unchanged_memory_write_invariant.2.4.1.2.

linear_plain_unchanged_memory_write_invariant.2.4.2:

{-1}	$\text{unchanged_memory_invariant?}(\text{pm_phy}'\text{mem}, \text{pm_phy}'\text{states},$ $\text{singleton}(\text{expr_2_super}(\text{memory_write}(\text{pm_phy}'\text{mem})(d', b'))),$ $(\text{pm_phy}'\text{rw_addr} \setminus \{d'\}))$ \wedge $\text{unchanged_memory_invariant?}(\text{pm_phy}'\text{mem}, \text{pm_phy}'\text{states},$ $\text{singleton}(\text{expr_2_super}(\text{memory_write}(\text{pm_phy}'\text{mem})(d', b'))),$ $\text{pm_phy}'\text{ro_addr})$ \supset $\text{unchanged_memory_invariant?}(\text{pm_phy}'\text{mem}, \text{pm_phy}'\text{states},$ $\text{singleton}(\text{expr_2_super}(\text{memory_write}(\text{pm_phy}'\text{mem})(d', b'))),$ $((\text{pm_phy}'\text{rw_addr} \setminus \{d'\}) \cup \text{pm_phy}'\text{ro_addr}))$
{-2}	$\text{pm_phy}'\text{rw_addr}(d') \supset$ $\text{unchanged_memory_invariant?}(\text{pm_phy}'\text{mem}, \text{pm_phy}'\text{states},$ $\text{singleton}(\text{expr_2_super}(\text{memory_write}(\text{pm_phy}'\text{mem})(d', b'))),$ $(\text{pm_phy}'\text{rw_addr} \setminus \{d'\}))$
{-3}	$\text{unchanged_memory_invariant?}(\text{pm_phy}'\text{mem}, \text{pm_phy}'\text{states},$ $\text{memory_write_transformers}(\text{pm_phy}'\text{mem}, \text{pm_phy}'\text{rw_addr}),$ $\text{pm_phy}'\text{ro_addr})$
{-4}	$\text{OK?}(\text{linear_resolve}(\text{waddr}', \text{Write})(s'))$
{-5}	$\text{data}(\text{linear_resolve}(\text{waddr}', \text{Write})(s')) = d'$
{-6}	$\text{is_linear_plain_memory?}(\text{pm}')$
{-7}	$\text{Mem?}(\text{type_of}(\text{waddr}'))$
{-8}	$0 \leq \text{waddr}'\text{'offset}$
{-9}	$\text{waddr}'\text{'type_of} = \text{waddr}'\text{'type_of}$
{-10}	$\text{waddr}'\text{'offset} < \text{max_linear_offset}$
{-11}	$\text{pm}'\text{'states} = \text{pm_phy}'\text{states}$
{-12}	$\text{plain_memory?}(\text{pm_phy})$
{-13}	$\text{pm}'\text{'states}(s')$
{-14}	$b' < \text{max_byte}$
{-15}	$\text{pm}'\text{'rw_addr}(\text{waddr}')$
{1}	$\text{pm_phy}'\text{rw_addr}(d')$
{2}	$\text{unchanged_memory_invariant?}(\text{pm_phy}'\text{mem}, \text{pm_phy}'\text{states},$ $\text{singleton}(\text{expr_2_super}(\text{memory_write}(\text{pm_phy}'\text{mem})(d', b'))),$ $\text{extend}[\text{Address}, \text{Memory_Address_4G}, \text{bool}, \text{FALSE}]$ $(\text{virt_to_phys_range}(s',$ <div style="text-align: right; margin-right: 20px;"> restrict $[\text{Address}, \text{Mem-}$ </div> $\text{ory_Address_4G}, \text{boolean}]$ $((\text{pm}'\text{'rw_addr} \setminus \{\text{waddr}'\})))$ $(\text{singleton}(\text{Read}) \cup \text{singleton}(\text{Ex$

Hiding formulas: (-1 -2 -3 2),

Using lemma pm_linear_blessed,

Expanding the definition of linear_blessed?,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Keeping (-4 -8 -9 -18 1) and hiding *,

Expanding the definition of subset?,

Expanding the definition of member,

Instantiating quantified variables,

we get 2 subgoals:

linear_plain_unchanged_memory_write_invariant.2.4.2.1:

{-1}	virt_to_phys_range(s' , restrict[Address, Memory_Address_4G, boolean](pm' 'rw_addr), singleton(Write)) (d') \Rightarrow restrict[Address, Memory_Address_4G, boolean](pm_phy' 'rw_addr)(d')
{-2}	OK?(linear_resolve(waddr', Write)(s'))
{-3}	data(linear_resolve(waddr', Write)(s')) = d'
{-4}	$b' < \text{max_byte}$
{1}	pm_phy' 'rw_addr(d')

Expanding the definition of restrict,

Expanding the definition of virt_to_phys_range,

Instantiating quantified variables,

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of linear_plain_unchanged_memory_write_invariant.2.4.2.1.

linear_plain_unchanged_memory_write_invariant.2.4.2.2:

{-1}	OK?(linear_resolve(waddr', Write)(s'))
{-2}	data(linear_resolve(waddr', Write)(s')) = d'
{-3}	$b' < \text{max_byte}$
{1}	Mem?(d' 'type_of) \wedge $0 \leq d'$ 'offset \wedge d' 'offset < max_linear_offset
{2}	pm_phy' 'rw_addr(d')

Using lemma linear_resolve_memory_address,

Expanding the definition of every,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of linear_plain_unchanged_memory_write_invariant.2.4.2.2.

linear_plain_unchanged_memory_write_invariant.3:

{-1}	is_linear_plain_memory?(pm')
{-2}	$b' < \text{max_byte}$
{-3}	pm' 'rw_addr(waddr')
{1}	(restrict[Address, Memory_Address_4G, boolean]((pm' 'rw_addr \ {waddr'})) \subseteq restrict [Address, Memory_Address_4G, boolean](pm' 'mem, pm' 'states, singleton(expr_2_super(memory_write(pm' 'mem)(waddr', b')), (pm' 'rw_addr \ {waddr'}))))
{2}	unchanged_memory_invariant?(pm' 'mem, pm' 'states, singleton(expr_2_super(memory_write(pm' 'mem)(waddr', b')), (pm' 'rw_addr \ {waddr'}))))

Keeping (1) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of linear_plain_unchanged_memory_write_invariant.3.

Q.E.D.

C.116.5 Linear_Memory_Blessing_Properties.linear_plain_transformer_invariant

Terse proof for linear_plain_transformer_invariant.

linear_plain_transformer_invariant:

{1}	\forall (pm : Plain_Memory[Physical_memory]): is_linear_plain_memory?(pm) \supset transformer_invariant?(pm' 'states, (pm' 'other_actions \cup (memory_read_transformers(pm' 'mem, (pm' 'ro_addr \cup pm' 'rw_addr))))
-----	--

C Proof scripts

Repeatedly Skolemizing and flattening,

Using lemma `linear_plain_unchanged_memory_invariant`,

Using lemma `unchanged_memory_invariant_invariant`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Using lemma `linear_plain_unchanged_memory_invariant_write`,

Using lemma `unchanged_memory_invariant_invariant`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Keeping `(-2 -5 1)` and hiding `*`,

Using lemma `transformer_invariant_union_transformers`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Case splitting on `union(memory_write_transformers(pm!1'mem, pm!1'rw_addr), union(union(pm!1'other_actions, memory_read_transformers(pm!1'mem, union (pm!1'ro_addr, pm!1'rw_addr))), union(memory_read_side_effect_super_transformers(pm!1'mem, union (pm!1'ro_addr, pm!1'rw_addr)), memory_write_side_effect_super_transformers(pm!1'mem, pm!1'rw_addr)))) = union(union(pm!1'other_actions, union(memory_read_transformers(pm!1'mem, union (pm!1'ro_addr, pm!1'rw_addr)), memory_write_transformers(pm!1'mem, pm!1'rw_addr))), union(memory_read_side_effect_super_transformers(pm!1'mem, union (pm!1'ro_addr, pm!1'rw_addr)), memory_write_side_effect_super_transformers(pm!1'mem, pm!1'rw_addr)))`,

we get 2 subgoals:

`linear_plain_transformer_invariant.1`:

{-1}	$(\text{memory_write_transformers}(\text{pm}'\text{mem}, \text{pm}'\text{rw_addr}) \cup ((\text{pm}'\text{other_actions} \cup \text{memory_read_transformers}(\text{pm}'\text{mem}, \text{pm}'\text{ro_addr} \cup \text{pm}'\text{rw_addr}))) \cup \text{memory_read_side_effect_super_transformers}(\text{pm}'\text{mem}, \text{pm}'\text{ro_addr} \cup \text{pm}'\text{rw_addr})) \cup \text{memory_write_side_effect_super_transformers}(\text{pm}'\text{mem}, \text{pm}'\text{rw_addr}))$
	=
{-2}	$\text{transformer_invariant?}(\text{pm}'\text{states}, \text{memory_write_transformers}(\text{pm}'\text{mem}, \text{pm}'\text{rw_addr}))$
{-3}	$\text{transformer_invariant?}(\text{pm}'\text{states},$
	$((\text{pm}'\text{other_actions} \cup \text{memory_read_transformers}(\text{pm}'\text{mem}, \text{pm}'\text{ro_addr} \cup \text{pm}'\text{rw_addr}))) \cup \text{memory_read_side_effect_super_transformers}(\text{pm}'\text{mem}, \text{pm}'\text{ro_addr} \cup \text{pm}'\text{rw_addr})) \cup \text{memory_write_side_effect_super_transformers}(\text{pm}'\text{mem}, \text{pm}'\text{rw_addr}))$
{-4}	$\text{transformer_invariant?}(\text{pm}'\text{states},$
	$(\text{memory_write_transformers}(\text{pm}'\text{mem}, \text{pm}'\text{rw_addr}) \cup ((\text{pm}'\text{other_actions} \cup \text{memory_read_transformers}(\text{pm}'\text{mem}, \text{pm}'\text{ro_addr} \cup \text{pm}'\text{rw_addr}))) \cup \text{memory_read_side_effect_super_transformers}(\text{pm}'\text{mem}, \text{pm}'\text{ro_addr} \cup \text{pm}'\text{rw_addr})) \cup \text{memory_write_side_effect_super_transformers}(\text{pm}'\text{mem}, \text{pm}'\text{rw_addr}))$
{1}	$\text{transformer_invariant?}(\text{pm}'\text{states},$
	$((\text{pm}'\text{other_actions} \cup \text{memory_read_transformers}(\text{pm}'\text{mem}, \text{pm}'\text{ro_addr} \cup \text{pm}'\text{rw_addr}))) \cup \text{memory_read_side_effect_super_transformers}(\text{pm}'\text{mem}, \text{pm}'\text{ro_addr} \cup \text{pm}'\text{rw_addr})) \cup \text{memory_write_side_effect_super_transformers}(\text{pm}'\text{mem}, \text{pm}'\text{rw_addr}))$

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `linear_plain_transformer_invariant.1`.

`linear_plain_transformer_invariant.2`:

{-1}	$\text{transformer_invariant?}(\text{pm}'\text{states}, \text{memory_write_transformers}(\text{pm}'\text{mem}, \text{pm}'\text{rw_addr}))$
{-2}	$\text{transformer_invariant?}(\text{pm}'\text{states},$
	$((\text{pm}'\text{other_actions} \cup \text{memory_read_transformers}(\text{pm}'\text{mem}, \text{pm}'\text{ro_addr} \cup \text{pm}'\text{rw_addr}))) \cup \text{memory_read_side_effect_super_transformers}(\text{pm}'\text{mem}, \text{pm}'\text{ro_addr} \cup \text{pm}'\text{rw_addr})) \cup \text{memory_write_side_effect_super_transformers}(\text{pm}'\text{mem}, \text{pm}'\text{rw_addr}))$
{-3}	$\text{transformer_invariant?}(\text{pm}'\text{states},$
	$(\text{memory_write_transformers}(\text{pm}'\text{mem}, \text{pm}'\text{rw_addr}) \cup ((\text{pm}'\text{other_actions} \cup \text{memory_read_transformers}(\text{pm}'\text{mem}, \text{pm}'\text{ro_addr} \cup \text{pm}'\text{rw_addr}))) \cup \text{memory_read_side_effect_super_transformers}(\text{pm}'\text{mem}, \text{pm}'\text{ro_addr} \cup \text{pm}'\text{rw_addr})) \cup \text{memory_write_side_effect_super_transformers}(\text{pm}'\text{mem}, \text{pm}'\text{rw_addr}))$
{1}	$(\text{memory_write_transformers}(\text{pm}'\text{mem}, \text{pm}'\text{rw_addr}) \cup ((\text{pm}'\text{other_actions} \cup \text{memory_read_transformers}(\text{pm}'\text{mem}, \text{pm}'\text{ro_addr} \cup \text{pm}'\text{rw_addr}))) \cup \text{memory_read_side_effect_super_transformers}(\text{pm}'\text{mem}, \text{pm}'\text{ro_addr} \cup \text{pm}'\text{rw_addr})) \cup \text{memory_write_side_effect_super_transformers}(\text{pm}'\text{mem}, \text{pm}'\text{rw_addr}))$
	=
{2}	$\text{transformer_invariant?}(\text{pm}'\text{states},$
	$((\text{pm}'\text{other_actions} \cup \text{memory_read_transformers}(\text{pm}'\text{mem}, \text{pm}'\text{ro_addr} \cup \text{pm}'\text{rw_addr}))) \cup \text{memory_read_side_effect_super_transformers}(\text{pm}'\text{mem}, \text{pm}'\text{ro_addr} \cup \text{pm}'\text{rw_addr})) \cup \text{memory_write_side_effect_super_transformers}(\text{pm}'\text{mem}, \text{pm}'\text{rw_addr}))$

Keeping (1) and hiding `*`,

Applying `decompose-equality`,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `linear_plain_transformer_invariant.2`.

Q.E.D.

C.116.6 Linear_Memory_Blessing_Properties.linear_plain_changed_memory_invariant

Terse proof for linear_plain_changed_memory_invariant.

linear_plain_changed_memory_invariant:

$$\frac{\{-1\}}{\{1\} \quad \forall (pm: Plain_Memory[Physical_memory]): \quad is_linear_plain_memory?(pm) \supset changed_memory_invariant?(pm' mem, pm' states, pm' rw_addr)}$$

Installing automatic rewrites from: has_next_state fatal_result singleton

Expanding the definition of changed_memory_invariant?,

Repeatedly Skolemizing and flattening,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 3 subgoals:

linear_plain_changed_memory_invariant.1:

$$\frac{\{-1\} \quad is_linear_plain_memory?(pm')}{\{1\} \quad \forall (s: Physical_memory, a: Address, b: Byte): \quad pm' states(s) \wedge pm' rw_addr(a) \wedge OK?(memory_write(pm' mem)(a, b)(s)) \wedge OK?(memory_read(pm' mem)(a)(state(memory_write(pm' mem)(a, b)(s)))) \supset data(memory_read(pm' mem)(a)(state(memory_write(pm' mem)(a, b)(s)))) = b}$$

Repeatedly Skolemizing and flattening,

Rewriting using pm_read_linear, matching in *,

Rewriting using pm_write_linear, matching in *,

Expanding the definition of linear_read,

Installing automatic rewrites from: (expr_2_super! expr_2_super_res! ##!)

Expanding the definition of linear_write,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 2 subgoals:

C Proof scripts

linear_plain_changed_memory_invariant.1.1:

```

{-1}  b' < max_byte
{-2}  is_linear_plain_memory?(pm')
{-3}  pm' `states(s')
{-4}  pm' `rw_addr(a')
{-5}  in_memory(min_linear, max_linear)(a')
{-6}  Mem?(type_of(a'))
{-7}  OK?(linear_resolve(a', Write)(s'))
{-8}  OK?(memory_write(pm_phy `mem
      (data(linear_resolve(a', Write)(s')), b')
      (state(linear_resolve(a', Write)(s')))))
{-9}  OK?(linear_resolve(a', Read)
      (state(memory_write(pm_phy `mem
      (data(linear_resolve(a', Write)(s')), b')
      (state(linear_resolve(a', Write)(s'))))))))
{-10} OK?(memory_read(pm_phy `mem
      (data(linear_resolve(a', Read)
      (state(memory_write(pm_phy `mem
      (data(linear_resolve(a', Write)(s')), b')
      (state(linear_resolve(a', Write)(s'))))))))
      (state(linear_resolve(a', Read)
      (state(memory_write(pm_phy `mem
      (data(linear_resolve(a', Write)(s')), b')
      (state(linear_resolve(a', Write)
      (s'))))))))))))
-----
{1}   data(memory_read(pm_phy `mem
      (data(linear_resolve(a', Read)
      (state(memory_write(pm_phy `mem
      (data(linear_resolve(a', Write)(s')), b')
      (state(linear_resolve(a', Write)
      (s'))))))))
      (state(linear_resolve(a', Read)
      (state(memory_write(pm_phy `mem
      (data(linear_resolve(a', Write)(s')), b')
      (state(linear_resolve(a', Write)
      (s'))))))))))))
      = b'

```

Using lemma linear_resolve_states,

Using lemma pm_plain_phy,

Using lemma pm_states,

Case splitting on union(pm!1'ro_addr, pm!1'rw_addr)(a!1),

we get 2 subgoals:

linear_plain_changed_memory_invariant.1.1.1:

<pre> {-1} union(pm'ro_addr, pm'rw_addr)(a') {-2} is_linear_plain_memory?(pm') ⊃ pm'states = pm_phy'states {-3} is_linear_plain_memory?(pm') ⊃ plain_memory?(pm_phy) {-4} union(pm'ro_addr, pm'rw_addr)(a') ∧ is_linear_plain_memory?(pm') ∧ pm'states(s') ∧ OK?(linear_resolve(a', Write)(s')) ⊃ pm_phy'states(state(linear_resolve(a', Write)(s'))) {-5} b' < max_byte {-6} is_linear_plain_memory?(pm') {-7} pm'states(s') {-8} pm'rw_addr(a') {-9} in_memory(min_linear, max_linear)(a') {-10} Mem?(type_of(a')) {-11} OK?(linear_resolve(a', Write)(s')) {-12} OK?(memory_write(pm_phy'mem) (data(linear_resolve(a', Write)(s')), b') (state(linear_resolve(a', Write)(s')))) {-13} OK?(linear_resolve(a', Read) (state(memory_write(pm_phy'mem) (data(linear_resolve(a', Write)(s')), b') (state(linear_resolve(a', Write)(s')))))) {-14} OK?(memory_read(pm_phy'mem) (data(linear_resolve(a', Read) (state(memory_write(pm_phy'mem) (data(linear_resolve(a', Write)(s')), b') (state(linear_resolve(a', Write)(s')))))))) </pre>	<pre> (s')))))) </pre> <hr style="border: 0.5px solid black;"/> <pre> {1} data(memory_read(pm_phy'mem) (data(linear_resolve(a', Read) (state(memory_write(pm_phy'mem) (data(linear_resolve(a', Write)(s')), b') (state(linear_resolve(a', Write) (s')))))))) (state(linear_resolve(a', Read) (state(memory_write(pm_phy'mem) (data(linear_resolve(a', Write)(s')), b') (state(linear_resolve(a', Write) (s')))))))) = b' </pre>
--	---

Using lemma super_transformer_invariant_next_ok,

Case splitting on pm_phy'rw_addr(data(linear_resolve(a!1, Write)(s!1))),

we get 2 subgoals:

linear_plain_changed_memory_invariant.1.1.1.1:

```

{-1} pm_phy'rw_addr(data(linear_resolve(a', Write)(s')))
{-2} transformer_invariant?(pm_phy'states, memory_write_transformers(pm_phy'mem, pm_phy'rw_
    pm_phy'states(state(linear_resolve(a', Write)(s'))) ^
    memory_write_transformers(pm_phy'mem, pm_phy'rw_addr)
    (expr_2_super(memory_write(pm_phy'mem)
    (data(linear_resolve(a', Write)(s')),
    b')))

    ^
    has_next_state(expr_2_super(memory_write(pm_phy'mem)
    (data(linear_resolve(a', Write)(s')), b'))
    (state(linear_resolve(a', Write)(s'))))

    ⊃
    pm_phy'states
    (state(expr_2_super(memory_write(pm_phy'mem)
    (data(linear_resolve(a', Write)(s')), b'))
    (state(linear_resolve(a', Write)(s'))))

{-3} union(pm'ro_addr, pm'rw_addr)(a')
{-4} is_linear_plain_memory?(pm') ⊃ pm'states = pm_phy'states
{-5} is_linear_plain_memory?(pm') ⊃ plain_memory?(pm_phy)
{-6} union(pm'ro_addr, pm'rw_addr)(a') ^
    is_linear_plain_memory?(pm') ^
    pm'states(s') ^ OK?(linear_resolve(a', Write)(s'))
    ⊃ pm_phy'states(state(linear_resolve(a', Write)(s')))
{-7} b' < max_byte
{-8} is_linear_plain_memory?(pm')
{-9} pm'states(s')
{-10} pm'rw_addr(a')
{-11} in_memory(min_linear, max_linear)(a')
{-12} Mem?(type_of(a'))
{-13} OK?(linear_resolve(a', Write)(s'))
{-14} OK?(memory_write(pm_phy'mem)
    (data(linear_resolve(a', Write)(s')), b')
    (state(linear_resolve(a', Write)(s'))))
{-15} OK?(linear_resolve(a', Read)
    (state(memory_write(pm_phy'mem)
    (data(linear_resolve(a', Write)(s')), b')
    (state(linear_resolve(a', Write)(s'))))))
{-16} OK?(memory_read(pm_phy'mem)
    (data(linear_resolve(a', Read)
    (state(memory_write(pm_phy'mem)
    (data(linear_resolve(a', Write)(s')), b')
    (state(linear_resolve(a', Write)(s'))))))))

    (state(linear_resolve(a', Read)
    (state(memory_write(pm_phy'mem)
    (data(linear_resolve(a', Write)(s')), b')
    (state(linear_resolve(a', Write)
    (s'))))))))



---


{1} data(memory_read(pm_phy'mem)
    (data(linear_resolve(a', Read)
    (state(memory_write(pm_phy'mem)
    (data(linear_resolve(a', Write)(s')), b')
    (state(linear_resolve(a', Write)
    (s'))))))))

    (state(linear_resolve(a', Read)
    (state(memory_write(pm_phy'mem)
    (data(linear_resolve(a', Write)(s')), b')
    (state(linear_resolve(a', Write)
    (s'))))))))

    = b'

```

C.116 Proofs for Linear_Memory_Blessing_Properties (challenge-linear.pvs)

Simplifying, rewriting, and recording with decision procedures,

Using lemma `plain_memory_changed_memory_invariant`,

Expanding the definition of `changed_memory_invariant?`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 2 subgoals:

linear_plain_changed_memory_invariant.1.1.1.1.1:

```

{-1} transformer_invariant?(pm_phy' states, memory_read_transformers(pm_phy' mem, pm_phy' rw_a
{-2} transformer_invariant?(pm_phy' states, memory_write_transformers(pm_phy' mem, pm_phy' rw_a
{-3} ∀ (s: Physical_memory, a: Address, b: Byte):
      pm_phy' states(s) ∧
      pm_phy' rw_addr(a) ∧
      OK?(memory_write(pm_phy' mem)(a, b)(s)) ∧
      OK?(memory_read(pm_phy' mem)(a)(state(memory_write(pm_phy' mem)(a, b)(s))))
      ⊃
      data(memory_read(pm_phy' mem)(a)(state(memory_write(pm_phy' mem)(a, b)(s))))
      = b
{-4} pm_phy' rw_addr(data(linear_resolve(a', Write)(s')))
{-5} pm_phy' states(state(linear_resolve(a', Write)(s')))
{-6} pm_phy' states
      (state(memory_write(pm_phy' mem)
                    (data(linear_resolve(a', Write)(s')), b')
                    (state(linear_resolve(a', Write)(s')))))
{-7} union(pm' ro_addr, pm' rw_addr)(a')
{-8} pm' states = pm_phy' states
{-9} plain_memory?(pm_phy)
{-10} b' < max_byte
{-11} is_linear_plain_memory?(pm')
{-12} pm' states(s')
{-13} pm' rw_addr(a')
{-14} in_memory(min_linear, max_linear)(a')
{-15} Mem?(type_of(a'))
{-16} OK?(linear_resolve(a', Write)(s'))
{-17} OK?(memory_write(pm_phy' mem)
      (data(linear_resolve(a', Write)(s')), b')
      (state(linear_resolve(a', Write)(s'))))
{-18} OK?(linear_resolve(a', Read)
      (state(memory_write(pm_phy' mem)
                    (data(linear_resolve(a', Write)(s')), b')
                    (state(linear_resolve(a', Write)(s'))))))
{-19} OK?(memory_read(pm_phy' mem)
      (data(linear_resolve(a', Read)
                    (state(memory_write(pm_phy' mem)
                    (data(linear_resolve(a', Write)(s')), b')
                    (state(linear_resolve(a', Write)(s'))))))))
      (state(linear_resolve(a', Read)
                    (state(memory_write(pm_phy' mem)
                    (data(linear_resolve(a', Write)(s')), b')
                    (state(linear_resolve(a', Write)
                    (s'))))))))
-----
{1} data(memory_read(pm_phy' mem)
      (data(linear_resolve(a', Read)
                    (state(memory_write(pm_phy' mem)
                    (data(linear_resolve(a', Write)(s')), b')
                    (state(linear_resolve(a', Write)
                    (s'))))))))
      (state(linear_resolve(a', Read)
                    (state(memory_write(pm_phy' mem)
                    (data(linear_resolve(a', Write)(s')), b')
                    (state(linear_resolve(a', Write)
                    (s'))))))))
      = b'

```

C.116 Proofs for Linear_Memory_Blessing_Properties (challenge-linear.pvs)

Instantiating the top quantifier in -3 with the terms: $(\text{state}(\text{linear_resolve}(\text{a!1}, \text{Write})(\text{s!1}))$
 $\text{data}(\text{linear_resolve}(\text{a!1}, \text{Write})(\text{s!1})) \text{ b!1}),$

Using lemma `linear_resolve_unchanged_pm_phy`,

Using lemma `unchanged_memory_invariant_unchanged`,

Case splitting on $\text{data}(\text{linear_resolve}(\text{a!1}, \text{Read})(\text{state}(\text{memory_write}(\text{pm_phy}'\text{mem})(\text{data}(\text{linear_resolve}(\text{a!1},$
 $\text{Write})(\text{s!1})), \text{b!1})(\text{state}(\text{linear_resolve}(\text{a!1}, \text{Write})(\text{s!1})))))) = \text{data}(\text{linear_resolve}(\text{a!1}, \text{Write})(\text{s!1}),$

we get 2 subgoals:

linear_plain_changed_memory_invariant.1.1.1.1.1.1:

```

{-1}  data(linear_resolve(a', Read)
      (state(memory_write(pm_phy' mem)
                 (data(linear_resolve(a', Write)(s')), b')
                 (state(linear_resolve(a', Write)(s'))))))
      = data(linear_resolve(a', Write)(s'))
{-2}  unchanged_memory_invariant?(pm_phy' mem, pm' states,
      singleton(expr_2_super[Physical_memory, Address]
                 (linear_resolve(a', Read))),
      ((pm_phy'ro_addr ∪ pm_phy'rw_addr) \ extend [Address, Mem-
      ^
      pm' states
      (state(memory_write(pm_phy' mem)
                 (data(linear_resolve(a', Write)(s')), b')
                 (state(linear_resolve(a', Write)(s'))))))
      ^
      singleton(expr_2_super[Physical_memory, Address](linear_resolve(a', Read)))
      (expr_2_super(linear_resolve(a', Read)))
      ^
      difference((pm_phy'ro_addr ∪ pm_phy'rw_addr),
      extend[Address, Memory_Address_4G, bool, FALSE]
      (address_in_pt_range?(s',
      restrict[Address, Mem-
      ^
      memory_Address_4G, boolean]
      ((pm'ro_addr ∪ pm'rw_addr))))
      (data(linear_resolve(a', Write)(s'))))
      ^
      OK?(expr_2_super(linear_resolve(a', Read))
      (state(memory_write(pm_phy' mem)
                 (data(linear_resolve(a', Write)(s')), b')
                 (state(linear_resolve(a', Write)(s'))))))
      ^
      OK?(memory_read(pm_phy' mem)
      (data(linear_resolve(a', Write)(s')))
      (state(memory_write(pm_phy' mem)
                 (data(linear_resolve(a', Write)(s')), b')
                 (state(linear_resolve(a', Write)(s'))))))
      ^
      OK?(memory_read(pm_phy' mem)
      (data(linear_resolve(a', Write)(s')))
      (state(expr_2_super(linear_resolve(a', Read))
                 (state(memory_write(pm_phy' mem)
                 (data(linear_resolve(a', Write)(s')),
                 b')
                 (state(linear_resolve(a', Write)
                 (s')))))))))
      ⊃
      data(memory_read(pm_phy' mem)
      (data(linear_resolve(a', Write)(s')))
      (state(expr_2_super(linear_resolve(a', Read))
                 (state(memory_write(pm_phy' mem)
                 (data(linear_resolve(a', Write)(s')), b')
                 (state(linear_resolve(a', Write)
                 (s')))))))))
      =
      data(memory_read(pm_phy' mem)
      (data(linear_resolve(a', Write)(s')))
      (state(memory_write(pm_phy' mem)
                 (data(linear_resolve(a', Write)(s')), b')
                 (state(linear_resolve(a', Write)(s'))))))
      (state(linear_resolve(a', Write)(s'))))
      (state(linear_resolve(a', Write)(s'))))
{-3}  union(pm'ro_addr, pm'rw_addr)(a') ∧ is_linear_plain_memory?(pm') ∧ pm' states(s')
      ⊃
      unchanged_memory_invariant?[Physical_memory]
      (pm_phy' mem, pm' states,

```

Replacing using formula -1,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 3 subgoals:


```
linear_plain_changed_memory_invariant.1.1.1.1.1.1.1:
```

```
{-1} data(linear_resolve(a', Read)
      (state(memory_write(pm_phy' mem)
                (data(linear_resolve(a', Write)(s')), b')
                (state(linear_resolve(a', Write)(s'))))))
      = data(linear_resolve(a', Write)(s'))
{-2} unchanged_memory_invariant?(pm_phy' mem, pm' states,
      singleton(expr_2_super[Physical_memory, Address]
                (linear_resolve(a', Read))),
      ((pm_phy' ro_addr ∪ pm_phy' rw_addr) \ extend [Address, Mem
{-3} union(pm' ro_addr, pm' rw_addr)(a')
{-4} is_linear_plain_memory?(pm')
{-5} pm' states(s')
{-6} transformer_invariant?(pm_phy' states, memory_read_transformers(pm_phy' mem, pm_phy' rw_a
{-7} transformer_invariant?(pm_phy' states, memory_write_transformers(pm_phy' mem, pm_phy' rw_a
{-8} pm_phy' states(state(linear_resolve(a', Write)(s')))
{-9} pm_phy' rw_addr(data(linear_resolve(a', Write)(s')))
{-10} OK?(memory_write(pm_phy' mem)
      (data(linear_resolve(a', Write)(s')), b')
      (state(linear_resolve(a', Write)(s'))))
{-11} pm_phy' states
      (state(memory_write(pm_phy' mem)
                (data(linear_resolve(a', Write)(s')), b')
                (state(linear_resolve(a', Write)(s'))))))
{-12} pm' states = pm_phy' states
{-13} plain_memory?(pm_phy)
{-14} b' < max_byte
{-15} pm' rw_addr(a')
{-16} in_memory(min_linear, max_linear)(a')
{-17} Mem?(type_of(a'))
{-18} OK?(linear_resolve(a', Write)(s'))
{-19} OK?(linear_resolve(a', Read)
      (state(memory_write(pm_phy' mem)
                (data(linear_resolve(a', Write)(s')), b')
                (state(linear_resolve(a', Write)(s'))))))
{-20} OK?(memory_read(pm_phy' mem)
      (data(linear_resolve(a', Write)(s')))
      (state(linear_resolve(a', Read)
                (state(memory_write(pm_phy' mem)
                          (data(linear_resolve(a', Write)(s')), b')
                          (state(linear_resolve(a', Write)
                                (s'))))))))


---


{1} OK?(memory_read(pm_phy' mem)
      (data(linear_resolve(a', Write)(s')))
      (state(memory_write(pm_phy' mem)
                (data(linear_resolve(a', Write)(s')), b')
                (state(linear_resolve(a', Write)(s'))))))
{2} data(memory_read(pm_phy' mem)
      (data(linear_resolve(a', Write)(s')))
      (state(linear_resolve(a', Read)
                (state(memory_write(pm_phy' mem)
                          (data(linear_resolve(a', Write)(s')), b')
                          (state(linear_resolve(a', Write)
                                (s'))))))))
      = b'
```

C.116 Proofs for Linear_Memory_Blessing_Properties (challenge-linear.pvs)

Using lemma `plain_memory_transformers_ok_read_ro_rw`,

Using lemma `expr_transformers_ok_ok`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Rewriting using `memory_read_transformers_memory_read`, matching in `*`,

Keeping `(-12 1)` and hiding `*`,

Rewriting using `union_right`, matching in `*`,

This completes the proof of `linear_plain_changed_memory_invariant.1.1.1.1.1.1.1.1`.

linear_plain_changed_memory_invariant.1.1.1.1.1.1.2:

```

{-1}  data(linear_resolve(a', Read)
      (state(memory_write(pm_phy' mem)
                (data(linear_resolve(a', Write)(s')), b')
                (state(linear_resolve(a', Write)(s'))))))
      = data(linear_resolve(a', Write)(s'))
{-2}  unchanged_memory_invariant?(pm_phy' mem, pm' states,
      singleton(expr_2_super[Physical_memory, Address]
                (linear_resolve(a', Read))),
      ((pm_phy'ro_addr ∪ pm_phy'rw_addr) \ extend [Address, Mem
{-3}  union(pm'ro_addr, pm'rw_addr)(a')
{-4}  is_linear_plain_memory?(pm')
{-5}  pm' states(s')
{-6}  transformer_invariant?(pm_phy' states, memory_read_transformers(pm_phy' mem, pm_phy'rw_a
{-7}  transformer_invariant?(pm_phy' states, memory_write_transformers(pm_phy' mem, pm_phy'rw_a
{-8}  pm_phy' states(state(linear_resolve(a', Write)(s')))
{-9}  pm_phy'rw_addr(data(linear_resolve(a', Write)(s')))
{-10} OK?(memory_write(pm_phy' mem)
      (data(linear_resolve(a', Write)(s')), b')
      (state(linear_resolve(a', Write)(s'))))
{-11} data(memory_read(pm_phy' mem)
      (data(linear_resolve(a', Write)(s')))
      (state(memory_write(pm_phy' mem)
                (data(linear_resolve(a', Write)(s')), b')
                (state(linear_resolve(a', Write)(s'))))))
      = b'
{-12} pm_phy' states
      (state(memory_write(pm_phy' mem)
                (data(linear_resolve(a', Write)(s')), b')
                (state(linear_resolve(a', Write)(s'))))))
{-13} pm' states = pm_phy' states
{-14} plain_memory?(pm_phy)
{-15} b' < max_byte
{-16} pm'rw_addr(a')
{-17} in_memory(min_linear, max_linear)(a')
{-18} Mem?(type_of(a'))
{-19} OK?(linear_resolve(a', Write)(s'))
{-20} OK?(linear_resolve(a', Read)
      (state(memory_write(pm_phy' mem)
                (data(linear_resolve(a', Write)(s')), b')
                (state(linear_resolve(a', Write)(s'))))))
{-21} OK?(memory_read(pm_phy' mem)
      (data(linear_resolve(a', Write)(s')))
      (state(linear_resolve(a', Read)
                (state(memory_write(pm_phy' mem)
                          (data(linear_resolve(a', Write)(s')), b')
                          (state(linear_resolve(a', Write)
                                (s'))))))))
-----
{1}  difference((pm_phy'ro_addr ∪ pm_phy'rw_addr),
      extend[Address, Memory_Address_4G, bool, FALSE]
      (address_in_pt_range?(s',
      restrict[Address, Mem-
      ory_Address_4G, boolean]
      ((pm'ro_addr ∪ pm'rw_addr))))
      (data(linear_resolve(a', Write)(s')))
{2}  data(memory_read(pm_phy' mem)
      (data(linear_resolve(a', Write)(s')))
      (state(linear_resolve(a', Read)
                (state(memory_write(pm_phy' mem)
                          (data(linear_resolve(a', Write)(s')), b')
                          (state(linear_resolve(a', Write)
                                (s'))))))))
      = b'

```

Hiding formulas: 2,

Using lemma `pm_resolve_address_write`,

we get 2 subgoals:

linear_plain_changed_memory_invariant.1.1.1.1.1.1.2.1:

{-1}	$\text{is_linear_plain_memory?}(pm') \wedge$ $pm' \text{'states}(s') \wedge$ $\text{virt_to_phys_range}(s', \text{restrict}[\text{Address}, \text{Memory_Address_4G}, \text{boolean}](pm' \text{'rw_addr}),$ $\text{singleton}(\text{Write}))$ $(\text{data}(\text{linear_resolve}(a', \text{Write})(s'))))$ \supset $\text{difference}((pm_phy' \text{'ro_addr} \cup pm_phy' \text{'rw_addr}),$ $\text{extend}[\text{Address}, \text{Memory_Address_4G}, \text{bool}, \text{FALSE}]$ $(\text{address_in_pt_range?}(s',$ $\text{restrict}[\text{Address}, \text{Mem-}$ $\text{ory_Address_4G}, \text{boolean}]$ $((pm' \text{'ro_addr} \cup pm' \text{'rw_addr}))))))$ $(\text{data}(\text{linear_resolve}(a', \text{Write})(s')))$
{-2}	$\text{data}(\text{linear_resolve}(a', \text{Read})$ $(\text{state}(\text{memory_write}(pm_phy' \text{'mem}$ $(\text{data}(\text{linear_resolve}(a', \text{Write})(s')), b')$ $(\text{state}(\text{linear_resolve}(a', \text{Write})(s'))))))$ $= \text{data}(\text{linear_resolve}(a', \text{Write})(s'))$
{-3}	$\text{unchanged_memory_invariant?}(pm_phy' \text{'mem}, pm' \text{'states},$ $\text{singleton}(\text{expr_2_super}[\text{Physical_memory}, \text{Address}]$ $(\text{linear_resolve}(a', \text{Read}))),$ $((pm_phy' \text{'ro_addr} \cup pm_phy' \text{'rw_addr}) \setminus \text{extend} [\text{Address}, \text{Mem-}$
{-4}	$\text{union}(pm' \text{'ro_addr}, pm' \text{'rw_addr})(a')$
{-5}	$\text{is_linear_plain_memory?}(pm')$
{-6}	$pm' \text{'states}(s')$
{-7}	$\text{transformer_invariant?}(pm_phy' \text{'states}, \text{memory_read_transformers}(pm_phy' \text{'mem}, pm_phy' \text{'rw_a}$
{-8}	$\text{transformer_invariant?}(pm_phy' \text{'states}, \text{memory_write_transformers}(pm_phy' \text{'mem}, pm_phy' \text{'rw_a}$
{-9}	$pm_phy' \text{'states}(\text{state}(\text{linear_resolve}(a', \text{Write})(s')))$
{-10}	$pm_phy' \text{'rw_addr}(\text{data}(\text{linear_resolve}(a', \text{Write})(s')))$
{-11}	$\text{OK?}(\text{memory_write}(pm_phy' \text{'mem}$ $(\text{data}(\text{linear_resolve}(a', \text{Write})(s')), b')$ $(\text{state}(\text{linear_resolve}(a', \text{Write})(s'))))$
{-12}	$\text{data}(\text{memory_read}(pm_phy' \text{'mem}$ $(\text{data}(\text{linear_resolve}(a', \text{Write})(s'))$ $(\text{state}(\text{memory_write}(pm_phy' \text{'mem}$ $(\text{data}(\text{linear_resolve}(a', \text{Write})(s')), b')$ $(\text{state}(\text{linear_resolve}(a', \text{Write})(s'))))))$
{-13}	$= b'$ $pm_phy' \text{'states}$ $(\text{state}(\text{memory_write}(pm_phy' \text{'mem}$ $(\text{data}(\text{linear_resolve}(a', \text{Write})(s')), b')$ $(\text{state}(\text{linear_resolve}(a', \text{Write})(s'))))$
{-14}	$pm' \text{'states} = pm_phy' \text{'states}$
{-15}	$\text{plain_memory?}(pm_phy)$
{-16}	$b' < \text{max_byte}$
{-17}	$pm' \text{'rw_addr}(a')$
{-18}	$\text{in_memory}(\text{min_linear}, \text{max_linear})(a')$
{-19}	$\text{Mem?}(\text{type_of}(a'))$
{-20}	$\text{OK?}(\text{linear_resolve}(a', \text{Write})(s'))$
{-21}	$\text{OK?}(\text{linear_resolve}(a', \text{Read})$ $(\text{state}(\text{memory_write}(pm_phy' \text{'mem}$ $(\text{data}(\text{linear_resolve}(a', \text{Write})(s')), b')$ $(\text{state}(\text{linear_resolve}(a', \text{Write})(s'))))))$
{-22}	$\text{OK?}(\text{memory_read}(pm_phy' \text{'mem}$ $(\text{data}(\text{linear_resolve}(a', \text{Write})(s'))$ $(\text{state}(\text{linear_resolve}(a', \text{Read})$ $(\text{state}(\text{memory_write}(pm_phy' \text{'mem}$ $(\text{data}(\text{linear_resolve}(a', \text{Write})(s')), b')$ $(\text{state}(\text{linear_resolve}(a', \text{Write})(s'))))))$ $(s'))))))))$
{1}	$\text{difference}((pm_phy' \text{'ro_addr} \cup pm_phy' \text{'rw_addr}),$ $\text{extend}[\text{Address}, \text{Memory_Address_4G}, \text{bool}, \text{FALSE}]$ $(\text{address_in_pt_range?}(s',$ $\text{restrict}[\text{Address}, \text{Mem-}$

C.116 Proofs for Linear_Memory_Blessing_Properties (challenge-linear.pvs)

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Expanding the definition of `virt_to_phys_range`,

Instantiating quantified variables,

Expanding the definition of `singleton`,

Expanding the definition of `restrict`,

which is trivially true.

This completes the proof of `linear_plain_changed_memory_invariant.1.1.1.1.1.1.2.1`.

linear_plain_changed_memory_invariant.1.1.1.1.1.1.2.2:

```

{-1}  data(linear_resolve(a', Read)
      (state(memory_write(pm_phy' mem)
                (data(linear_resolve(a', Write)(s')), b')
                (state(linear_resolve(a', Write)(s'))))))
      = data(linear_resolve(a', Write)(s'))
{-2}  unchanged_memory_invariant?(pm_phy' mem, pm' states,
      singleton(expr_2_super[Physical_memory, Address]
                (linear_resolve(a', Read))),
      ((pm_phy'ro_addr ∪ pm_phy'rw_addr) \ extend [Address, Mem
{-3}  union(pm'ro_addr, pm'rw_addr)(a')
{-4}  is_linear_plain_memory?(pm')
{-5}  pm' states(s')
{-6}  transformer_invariant?(pm_phy' states, memory_read_transformers(pm_phy' mem, pm_phy'rw_a
{-7}  transformer_invariant?(pm_phy' states, memory_write_transformers(pm_phy' mem, pm_phy'rw_a
{-8}  pm_phy' states(state(linear_resolve(a', Write)(s')))
{-9}  pm_phy'rw_addr(data(linear_resolve(a', Write)(s')))
{-10} OK?(memory_write(pm_phy' mem)
      (data(linear_resolve(a', Write)(s')), b')
      (state(linear_resolve(a', Write)(s'))))
{-11} data(memory_read(pm_phy' mem)
      (data(linear_resolve(a', Write)(s')))
      (state(memory_write(pm_phy' mem)
                (data(linear_resolve(a', Write)(s')), b')
                (state(linear_resolve(a', Write)(s'))))))
      = b'
{-12} pm_phy' states
      (state(memory_write(pm_phy' mem)
                (data(linear_resolve(a', Write)(s')), b')
                (state(linear_resolve(a', Write)(s'))))))
{-13} pm' states = pm_phy' states
{-14} plain_memory?(pm_phy)
{-15} b' < max_byte
{-16} pm'rw_addr(a')
{-17} in_memory(min_linear, max_linear)(a')
{-18} Mem?(type_of(a'))
{-19} OK?(linear_resolve(a', Write)(s'))
{-20} OK?(linear_resolve(a', Read)
      (state(memory_write(pm_phy' mem)
                (data(linear_resolve(a', Write)(s')), b')
                (state(linear_resolve(a', Write)(s'))))))
{-21} OK?(memory_read(pm_phy' mem)
      (data(linear_resolve(a', Write)(s')))
      (state(linear_resolve(a', Read)
                (state(memory_write(pm_phy' mem)
                          (data(linear_resolve(a', Write)(s')), b')
                          (state(linear_resolve(a', Write)
                                (s'))))))))


---


{1}  Mem?(data[Physical_memory, Address]
      (linear_resolve[Physical_memory, pm_phy](a', Write)(s'))'type_of)
      ^
      0 ≤
      data[Physical_memory, Address]
      (linear_resolve[Physical_memory, pm_phy](a', Write)(s'))'offset
      ^
      data[Physical_memory, Address]
      (linear_resolve[Physical_memory, pm_phy](a', Write)(s'))'offset
      < max_linear_offset
{2}  difference((pm_phy'ro_addr ∪ pm_phy'rw_addr),
      extend[Address, Memory_Address_4G, bool, FALSE]
      (address_in_pt_range?(s',
      restrict[Address, Mem-
      ory_Address_4G, boolean]
      ((pm'ro_addr ∪ pm'rw_addr))))
      (data(linear_resolve(a', Write)(s')))

```

Using lemma `linear_resolve_memory_address`,

Expanding the definition of `every`,

which is trivially true.

This completes the proof of `linear_plain_changed_memory_invariant.1.1.1.1.1.2.2`.

linear_plain_changed_memory_invariant.1.1.1.1.1.1.3:

```

{-1}  data(linear_resolve(a', Read)
      (state(memory_write(pm_phy' mem)
                (data(linear_resolve(a', Write)(s')), b')
                (state(linear_resolve(a', Write)(s'))))))
      = data(linear_resolve(a', Write)(s'))
{-2}  unchanged_memory_invariant?(pm_phy' mem, pm' states,
      singleton(expr_2_super[Physical_memory, Address]
                (linear_resolve(a', Read))),
      ((pm_phy' ro_addr ∪ pm_phy' rw_addr) \ extend [Address, Mem
{-3}  union(pm' ro_addr, pm' rw_addr)(a')
{-4}  is_linear_plain_memory?(pm')
{-5}  pm' states(s')
{-6}  transformer_invariant?(pm_phy' states, memory_read_transformers(pm_phy' mem, pm_phy' rw_a
{-7}  transformer_invariant?(pm_phy' states, memory_write_transformers(pm_phy' mem, pm_phy' rw_a
{-8}  pm_phy' states(state(linear_resolve(a', Write)(s')))
{-9}  pm_phy' rw_addr(data(linear_resolve(a', Write)(s')))
{-10} OK?(memory_write(pm_phy' mem)
      (data(linear_resolve(a', Write)(s')), b')
      (state(linear_resolve(a', Write)(s'))))
{-11} data(memory_read(pm_phy' mem)
      (data(linear_resolve(a', Write)(s')))
      (state(memory_write(pm_phy' mem)
                (data(linear_resolve(a', Write)(s')), b')
                (state(linear_resolve(a', Write)(s'))))))
      = b'
{-12} pm_phy' states
      (state(memory_write(pm_phy' mem)
                (data(linear_resolve(a', Write)(s')), b')
                (state(linear_resolve(a', Write)(s'))))
{-13} pm' states = pm_phy' states
{-14} plain_memory?(pm_phy)
{-15} b' < max_byte
{-16} pm' rw_addr(a')
{-17} in_memory(min_linear, max_linear)(a')
{-18} Mem?(type_of(a'))
{-19} OK?(linear_resolve(a', Write)(s'))
{-20} OK?(linear_resolve(a', Read)
      (state(memory_write(pm_phy' mem)
                (data(linear_resolve(a', Write)(s')), b')
                (state(linear_resolve(a', Write)(s'))))))
{-21} OK?(memory_read(pm_phy' mem)
      (data(linear_resolve(a', Write)(s')))
      (state(linear_resolve(a', Read)
                (state(memory_write(pm_phy' mem)
                          (data(linear_resolve(a', Write)(s')), b')
                          (state(linear_resolve(a', Write)
                                (s'))))))))


---


{1}  expr_2_super(linear_resolve(a', Read)) =
      expr_2_super[Physical_memory, Address](linear_resolve(a', Read))
{2}  data(memory_read(pm_phy' mem)
      (data(linear_resolve(a', Write)(s')))
      (state(linear_resolve(a', Read)
                (state(memory_write(pm_phy' mem)
                          (data(linear_resolve(a', Write)(s')), b')
                          (state(linear_resolve(a', Write)
                                (s'))))))))
      = b'

```

Keeping (1) and hiding *,

Applying `decompose-equality`,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `linear_plain_changed_memory_invariant.1.1.1.1.1.3`.

linear_plain_changed_memory_invariant.1.1.1.1.1.2:

```

{-1}  unchanged_memory_invariant?(pm_phy'mem, pm'states,
                                         singleton(expr_2_super[Physical_memory, Address]
                                                         (linear_resolve(a', Read))),
                                         ((pm_phy'ro_addr ∪ pm_phy'rw_addr) \ extend [Address, Mem
                                         ^
                                         pm'states
                                         (state(memory_write(pm_phy'mem)
                                                         (data(linear_resolve(a', Write)(s')), b')
                                                         (state(linear_resolve(a', Write)(s')))))
                                         ^
                                         singleton(expr_2_super[Physical_memory, Address](linear_resolve(a', Read)))
                                         (expr_2_super(linear_resolve(a', Read)))
                                         ^
                                         difference((pm_phy'ro_addr ∪ pm_phy'rw_addr),
                                                         extend[Address, Memory_Address_4G, bool, FALSE]
                                                         (address_in_pt_range?(s',
                                                         restrict[Address, Mem-
memory_Address_4G, boolean]
                                                         ((pm'ro_addr ∪ pm'rw_addr))))
                                         (data(linear_resolve(a', Write)(s')))
                                         ^
                                         OK?(expr_2_super(linear_resolve(a', Read))
                                                         (state(memory_write(pm_phy'mem)
                                                         (data(linear_resolve(a', Write)(s')), b')
                                                         (state(linear_resolve(a', Write)(s')))))
                                         ^
                                         OK?(memory_read(pm_phy'mem)
                                                         (data(linear_resolve(a', Write)(s')))
                                                         (state(memory_write(pm_phy'mem)
                                                         (data(linear_resolve(a', Write)(s')), b')
                                                         (state(linear_resolve(a', Write)(s')))))
                                         ^
                                         OK?(memory_read(pm_phy'mem)
                                                         (data(linear_resolve(a', Write)(s')))
                                                         (state(expr_2_super(linear_resolve(a', Read))
                                                         (state(memory_write(pm_phy'mem)
                                                         (data(linear_resolve(a', Write)(s')),
                                                         b')
                                                         (state(linear_resolve(a', Write)
                                                         (s'))))))))
                                         ⊃
                                         data(memory_read(pm_phy'mem)
                                                         (data(linear_resolve(a', Write)(s')))
                                                         (state(expr_2_super(linear_resolve(a', Read))
                                                         (state(memory_write(pm_phy'mem)
                                                         (data(linear_resolve(a', Write)(s')), b')
                                                         (state(linear_resolve(a', Write)
                                                         (s'))))))))
                                         =
                                         data(memory_read(pm_phy'mem)
                                                         (data(linear_resolve(a', Write)(s')))
                                                         (state(memory_write(pm_phy'mem)
                                                         (data(linear_resolve(a', Write)(s')), b')
                                                         (state(linear_resolve(a', Write)(s')))))
{-2}  union(pm'ro_addr, pm'rw_addr)(a') ∧ is_linear_plain_memory?(pm') ∧ pm'states(s')
                                         ⊃
                                         unchanged_memory_invariant?[Physical_memory]
                                         (pm_phy'mem, pm'states,
                                         singleton(expr_2_super[Physical_memory, Address](linear_resolve(a', Read))),
                                         ((pm_phy'ro_addr ∪ pm_phy'rw_addr) \ extend [Address, Memory_Address_4G, bool, F
{-3}  transformer_invariant?(pm_phy'states, memory_read_transformers(pm_phy'mem, pm_phy'rw_a
{-4}  transformer_invariant?(pm_phy'states, memory_write_transformers(pm_phy'mem, pm_phy'rw_a
{-5}  pm_phy'states(state(linear_resolve(a', Write)(s'))) ∧

```

Hiding formulas: 2,

Using lemma `linear_resolve_same_result`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `linear_plain_changed_memory_invariant.1.1.1.1.1.2`.

linear_plain_changed_memory_invariant.1.1.1.1.2:

```

{-1} transformer_invariant?(pm_phy' states, memory_read_transformers(pm_phy' mem, pm_phy' rw_a
{-2} transformer_invariant?(pm_phy' states, memory_write_transformers(pm_phy' mem, pm_phy' rw_a
{-3} ∀ (s: Physical_memory, a: Address, b: Byte):
      pm_phy' states(s) ∧
      pm_phy' rw_addr(a) ∧
      OK?(memory_write(pm_phy' mem)(a, b)(s)) ∧
      OK?(memory_read(pm_phy' mem)(a)(state(memory_write(pm_phy' mem)(a, b)(s))))
      ⊃
      data(memory_read(pm_phy' mem)(a)(state(memory_write(pm_phy' mem)(a, b)(s))))
      = b
{-4} pm_phy' rw_addr(data(linear_resolve(a', Write)(s')))
{-5} pm_phy' states(state(linear_resolve(a', Write)(s')))
{-6} union(pm' ro_addr, pm' rw_addr)(a')
{-7} pm' states = pm_phy' states
{-8} plain_memory?(pm_phy)
{-9} b' < max_byte
{-10} is_linear_plain_memory?(pm')
{-11} pm' states(s')
{-12} pm' rw_addr(a')
{-13} in_memory(min_linear, max_linear)(a')
{-14} Mem?(type_of(a'))
{-15} OK?(linear_resolve(a', Write)(s'))
{-16} OK?(memory_write(pm_phy' mem)
      (data(linear_resolve(a', Write)(s')), b')
      (state(linear_resolve(a', Write)(s'))))
{-17} OK?(linear_resolve(a', Read)
      (state(memory_write(pm_phy' mem)
      (data(linear_resolve(a', Write)(s')), b')
      (state(linear_resolve(a', Write)(s'))))))
{-18} OK?(memory_read(pm_phy' mem)
      (data(linear_resolve(a', Read)
      (state(memory_write(pm_phy' mem)
      (data(linear_resolve(a', Write)(s')), b')
      (state(linear_resolve(a', Write)(s'))))))))
      (state(linear_resolve(a', Read)
      (state(memory_write(pm_phy' mem)
      (data(linear_resolve(a', Write)(s')), b')
      (state(linear_resolve(a', Write)
      (s'))))))))
-----
{1} memory_write_transformers(pm_phy' mem, pm_phy' rw_addr)
      (expr_2_super(memory_write(pm_phy' mem)
      (data(linear_resolve(a', Write)(s')),
      b')))
{2} data(memory_read(pm_phy' mem)
      (data(linear_resolve(a', Read)
      (state(memory_write(pm_phy' mem)
      (data(linear_resolve(a', Write)(s')), b')
      (state(linear_resolve(a', Write)
      (s'))))))))
      (state(linear_resolve(a', Read)
      (state(memory_write(pm_phy' mem)
      (data(linear_resolve(a', Write)(s')), b')
      (state(linear_resolve(a', Write)
      (s'))))))))
      = b'

```

C.116 Proofs for Linear_Memory_Blessing_Properties (challenge-linear.pvs)

Rewriting using `memory_write_transformers_memory_write`, matching in `*`,

This completes the proof of `linear_plain_changed_memory_invariant.1.1.1.1.2`.

linear_plain_changed_memory_invariant.1.1.1.2:

```

{-1} transformer_invariant?(pm_phy' states, memory_write_transformers(pm_phy' mem, pm_phy' rw_
      pm_phy' states(state(linear_resolve(a', Write)(s'))) ^
      memory_write_transformers(pm_phy' mem, pm_phy' rw_addr)
      (expr_2_super(memory_write(pm_phy' mem)
      (data(linear_resolve(a', Write)(s')),
      b'))))
      ^
      has_next_state(expr_2_super(memory_write(pm_phy' mem)
      (data(linear_resolve(a', Write)(s')), b'))
      (state(linear_resolve(a', Write)(s'))))
      ⊃
      pm_phy' states
      (state(expr_2_super(memory_write(pm_phy' mem)
      (data(linear_resolve(a', Write)(s')), b'))
      (state(linear_resolve(a', Write)(s'))))
{-2} union(pm' ro_addr, pm' rw_addr)(a')
{-3} is_linear_plain_memory?(pm') ⊃ pm' states = pm_phy' states
{-4} is_linear_plain_memory?(pm') ⊃ plain_memory?(pm_phy)
{-5} union(pm' ro_addr, pm' rw_addr)(a') ^
      is_linear_plain_memory?(pm') ^
      pm' states(s') ^ OK?(linear_resolve(a', Write)(s'))
      ⊃ pm_phy' states(state(linear_resolve(a', Write)(s'))
{-6} b' < max_byte
{-7} is_linear_plain_memory?(pm')
{-8} pm' states(s')
{-9} pm' rw_addr(a')
{-10} in_memory(min_linear, max_linear)(a')
{-11} Mem?(type_of(a'))
{-12} OK?(linear_resolve(a', Write)(s'))
{-13} OK?(memory_write(pm_phy' mem)
      (data(linear_resolve(a', Write)(s')), b')
      (state(linear_resolve(a', Write)(s'))))
{-14} OK?(linear_resolve(a', Read)
      (state(memory_write(pm_phy' mem)
      (data(linear_resolve(a', Write)(s')), b')
      (state(linear_resolve(a', Write)(s'))))))
{-15} OK?(memory_read(pm_phy' mem)
      (data(linear_resolve(a', Read)
      (state(memory_write(pm_phy' mem)
      (data(linear_resolve(a', Write)(s')), b')
      (state(linear_resolve(a', Write)(s'))))))))
      (state(linear_resolve(a', Read)
      (state(memory_write(pm_phy' mem)
      (data(linear_resolve(a', Write)(s')), b')
      (state(linear_resolve(a', Write)
      (s'))))))))
-----
{1} pm_phy' rw_addr(data(linear_resolve(a', Write)(s')))
{2} data(memory_read(pm_phy' mem)
      (data(linear_resolve(a', Read)
      (state(memory_write(pm_phy' mem)
      (data(linear_resolve(a', Write)(s')), b')
      (state(linear_resolve(a', Write)
      (s'))))))))
      (state(linear_resolve(a', Read)
      (state(memory_write(pm_phy' mem)
      (data(linear_resolve(a', Write)(s')), b')
      (state(linear_resolve(a', Write)
      (s'))))))))
      = b'

```

Hiding formulas: 2,

Using lemma pm_linear_blessed,

Expanding the definition of linear_blessed?,

Hiding formulas: (-2 -6),

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Hiding formulas: (-1 -2 -3 -5 -6 -7),

Expanding the definition of subset?,

Expanding the definition of member,

Expanding the definition of restrict,

Instantiating quantified variables,

we get 2 subgoals:

linear_plain_changed_memory_invariant.1.1.1.2.1:

{-1}	virt_to_phys_range(s' , $\lambda (s: \text{Memory_Address_4G}): \text{pm}'\text{rw_addr}(s), \text{singleton}(\text{Write}))$ $(\text{data}(\text{linear_resolve}(a', \text{Write})(s')))$ $\Rightarrow \text{pm_phy}'\text{rw_addr}(\text{data}(\text{linear_resolve}(a', \text{Write})(s')))$
{-2}	$\text{union}(\text{pm}'\text{ro_addr}, \text{pm}'\text{rw_addr})(a')$
{-3}	$\text{is_linear_plain_memory?}(\text{pm}')$
{-4}	$\text{pm}'\text{states} = \text{pm_phy}'\text{states}$
{-5}	$\text{plain_memory?}(\text{pm_phy})$
{-6}	$b' < \text{max_byte}$
{-7}	$\text{pm}'\text{states}(s')$
{-8}	$\text{pm}'\text{rw_addr}(a')$
{-9}	$\text{in_memory}(\text{min_linear}, \text{max_linear})(a')$
{-10}	$\text{Mem?}(\text{type_of}(a'))$
{-11}	$\text{OK?}(\text{linear_resolve}(a', \text{Write})(s'))$
{-12}	$\text{OK?}(\text{memory_write}(\text{pm_phy}'\text{mem})$ $(\text{data}(\text{linear_resolve}(a', \text{Write})(s')), b')$ $(\text{state}(\text{linear_resolve}(a', \text{Write})(s'))))$
{-13}	$\text{OK?}(\text{linear_resolve}(a', \text{Read})$ $(\text{state}(\text{memory_write}(\text{pm_phy}'\text{mem})$ $(\text{data}(\text{linear_resolve}(a', \text{Write})(s')), b')$ $(\text{state}(\text{linear_resolve}(a', \text{Write})(s'))))$
{-14}	$\text{OK?}(\text{memory_read}(\text{pm_phy}'\text{mem})$ $(\text{data}(\text{linear_resolve}(a', \text{Read})$ $(\text{state}(\text{memory_write}(\text{pm_phy}'\text{mem})$ $(\text{data}(\text{linear_resolve}(a', \text{Write})(s')), b')$ $(\text{state}(\text{linear_resolve}(a', \text{Write})(s'))))$ $(\text{state}(\text{linear_resolve}(a', \text{Read})$ $(\text{state}(\text{memory_write}(\text{pm_phy}'\text{mem})$ $(\text{data}(\text{linear_resolve}(a', \text{Write})(s')), b')$ $(\text{state}(\text{linear_resolve}(a', \text{Write})$ $(s'))))))))$
{1}	$\text{pm_phy}'\text{rw_addr}(\text{data}(\text{linear_resolve}(a', \text{Write})(s')))$

Expanding the definition of virt_to_phys_range,

Instantiating quantified variables,

This completes the proof of linear_plain_changed_memory_invariant.1.1.1.2.1.

linear_plain_changed_memory_invariant.1.1.1.2.2:

```

{-1} union(pm'ro_addr, pm'rw_addr)(a')
{-2} is_linear_plain_memory?(pm')
{-3} pm'states = pm_phy'states
{-4} plain_memory?(pm_phy)
{-5} b' < max_byte
{-6} pm'states(s')
{-7} pm'rw_addr(a')
{-8} in_memory(min_linear, max_linear)(a')
{-9} Mem?(type_of(a'))
{-10} OK?(linear_resolve(a', Write)(s'))
{-11} OK?(memory_write(pm_phy'mem)
      (data(linear_resolve(a', Write)(s')), b')
      (state(linear_resolve(a', Write)(s'))))
{-12} OK?(linear_resolve(a', Read)
      (state(memory_write(pm_phy'mem)
      (data(linear_resolve(a', Write)(s')), b')
      (state(linear_resolve(a', Write)(s'))))))
{-13} OK?(memory_read(pm_phy'mem)
      (data(linear_resolve(a', Read)
      (state(memory_write(pm_phy'mem)
      (data(linear_resolve(a', Write)(s')), b')
      (state(linear_resolve(a', Write)(s'))))))))
      (state(linear_resolve(a', Read)
      (state(memory_write(pm_phy'mem)
      (data(linear_resolve(a', Write)(s')), b')
      (state(linear_resolve(a', Write)
      (s'))))))))
-----
{1} Mem?(data[Physical_memory, Address]
      (linear_resolve[Physical_memory, pm_phy](a', Write)(s'))'type_of)
      ^
      0 ≤
      data[Physical_memory, Address]
      (linear_resolve[Physical_memory, pm_phy](a', Write)(s'))'offset
      ^
      data[Physical_memory, Address]
      (linear_resolve[Physical_memory, pm_phy](a', Write)(s'))'offset
      < max_linear_offset
{2} pm_phy'rw_addr(data(linear_resolve(a', Write)(s')))

```

Using lemma linear_resolve_memory_address,

Expanding the definition of every,

which is trivially true.

This completes the proof of linear_plain_changed_memory_invariant.1.1.1.2.2.

linear_plain_changed_memory_invariant.1.1.2:

{-1}	is_linear_plain_memory?(pm') \supset pm' 'states = pm_phy 'states
{-2}	is_linear_plain_memory?(pm') \supset plain_memory?(pm_phy)
{-3}	union(pm' 'ro_addr, pm' 'rw_addr)(a') \wedge is_linear_plain_memory?(pm') \wedge pm' 'states(s') \wedge OK?(linear_resolve(a', Write)(s')) \supset pm_phy 'states(state(linear_resolve(a', Write)(s')))
{-4}	b' < max_byte
{-5}	is_linear_plain_memory?(pm')
{-6}	pm' 'states(s')
{-7}	pm' 'rw_addr(a')
{-8}	in_memory(min_linear, max_linear)(a')
{-9}	Mem?(type_of(a'))
{-10}	OK?(linear_resolve(a', Write)(s'))
{-11}	OK?(memory_write(pm_phy 'mem) (data(linear_resolve(a', Write)(s')), b') (state(linear_resolve(a', Write)(s'))))
{-12}	OK?(linear_resolve(a', Read) (state(memory_write(pm_phy 'mem) (data(linear_resolve(a', Write)(s')), b') (state(linear_resolve(a', Write)(s'))))))
{-13}	OK?(memory_read(pm_phy 'mem) (data(linear_resolve(a', Read) (state(memory_write(pm_phy 'mem) (data(linear_resolve(a', Write)(s')), b') (state(linear_resolve(a', Write)(s')))))))) (state(linear_resolve(a', Read) (state(memory_write(pm_phy 'mem) (data(linear_resolve(a', Write)(s')), b') (state(linear_resolve(a', Write) (s'))))))))
{1} union(pm' 'ro_addr, pm' 'rw_addr)(a')	
{2}	data(memory_read(pm_phy 'mem) (data(linear_resolve(a', Read) (state(memory_write(pm_phy 'mem) (data(linear_resolve(a', Write)(s')), b') (state(linear_resolve(a', Write) (s')))))))) (state(linear_resolve(a', Read) (state(memory_write(pm_phy 'mem) (data(linear_resolve(a', Write)(s')), b') (state(linear_resolve(a', Write) (s'))))))))
	= b'

Rewriting using union_right, matching in *,

This completes the proof of linear_plain_changed_memory_invariant.1.1.2.

C Proof scripts

`linear_plain_changed_memory_invariant.1.2:`

{-1}	<code>b' < max_byte</code>
{-2}	<code>is_linear_plain_memory?(pm')</code>
{-3}	<code>pm' 'states(s')</code>
{-4}	<code>pm' 'rw_addr(a')</code>
{-5}	<code>in_memory(min_linear, max_linear)(a')</code>
{-6}	<code>OK?(memory_write(pm_phy 'mem)(a', b')(s'))</code>
{-7}	<code>OK?(memory_read(pm_phy 'mem)(a')(state(memory_write(pm_phy 'mem)(a', b')(s'))))</code>
{1}	<code>Mem?(type_of(a'))</code>
{2}	<code>data(memory_read(pm_phy 'mem)(a')(state(memory_write(pm_phy 'mem)(a', b')(s'))))</code> <code>= b'</code>

Using lemma `pm_memory_addr`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Rewriting using `union_right`, matching in `*`,

This completes the proof of `linear_plain_changed_memory_invariant.1.2`.

`linear_plain_changed_memory_invariant.2:`

{-1}	<code>is_linear_plain_memory?(pm')</code>
{1}	<code>transformer_invariant?(pm' 'states, memory_write_transformers(pm' 'mem, pm' 'rw_addr))</code>

Using lemma `linear_plain_transformer_invariant`,

Simplifying, rewriting, and recording with decision procedures,

Rewriting using `transformer_invariant_mono_transformers`, matching in `*` where `transformers_1` gets `memory_write_transformers(pm!1'mem, pm!1'rw_addr)`, `transformers_2` gets `union(union(pm!1'other_actions, union(memory_read_transformers(pm!1'mem, union (pm!1'ro_addr, pm!1'rw_addr)), memory_write_transformers(pm!1'mem, pm!1'rw_addr))), union(memory_read_side_effect_super_tr union (pm!1'ro_addr, pm!1'rw_addr), memory_write_side_effect_super_transformers(pm!1'mem, pm!1'rw_addr)))`,

Keeping (1) and hiding `*`,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `linear_plain_changed_memory_invariant.2`.

`linear_plain_changed_memory_invariant.3:`

{-1}	<code>is_linear_plain_memory?(pm')</code>
{1}	<code>transformer_invariant?(pm' 'states, memory_read_transformers(pm' 'mem, pm' 'rw_addr))</code>

Using lemma `linear_plain_transformer_invariant`,

Simplifying, rewriting, and recording with decision procedures,

Rewriting using `transformer_invariant_mono_transformers`, matching in `*` where `transformers_1` gets `memory_read_transformers(pm!1'mem, pm!1'rw_addr)`, `transformers_2` gets `union(union(pm!1'other_actions, union(memory_read_transformers(pm!1'mem, union (pm!1'ro_addr, pm!1'rw_addr)), memory_write_transformers(pm!1'mem, pm!1'rw_addr))), union(memory_read_side_effect_super_tr union (pm!1'ro_addr, pm!1'rw_addr), memory_write_side_effect_super_transformers(pm!1'mem, pm!1'rw_addr)))`,

Keeping (1) and hiding `*`,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `linear_plain_changed_memory_invariant.3`.

Q.E.D.

C.116.7 Linear Memory Blessing Properties.linear_plain_read_side_effect_unchanged

Terse proof for `linear_plain_read_side_effect_unchanged`.

linear_plain_read_side_effect_unchanged:

```
{1}  ∀ (pm: Plain_Memory [Physical_Memory]):
      is_linear_plain_memory?(pm) ⊃
      side_effect_content_unchanged((pm'ro_addr ∪ pm'rw_addr), pm'states,
                                    memory_read_side_effect(pm'mem))
```

Expanding the definition of side_effect_content_unchanged,

Repeatedly Skolemizing and flattening,

Hiding formulas: -1,

Rewriting using pm_read_side_effect_linear, matching in *,

Expanding the definition of linear_read_side_effect,

Case splitting on Mem?(type_of(a!1)),

we get 2 subgoals:

linear_plain_read_side_effect_unchanged.1:

```
{-1} Mem?(type_of(a'))
{-2} is_linear_plain_memory?(pm')
{-3} pm'states(s')
{-4} (address_block(a', length(bl')) ⊆ (pm'ro_addr ∪ pm'rw_addr))
{-5} OK?(IF null?(bl') ∨
          (reg_base(min_linear)(type_of(a')) ≤ a' ∧
           a' + length(bl') ≤ reg_size(max_linear)(type_of(a')))
        THEN IF Mem?(type_of(a'))
              THEN apply_side_effects(split(min_page, a', bl'),
                                       linear_read_side_effect_in_page)
                                       (s')
              ELSE memory_read_side_effect(pm_phy'mem)(a', bl', cp')(s')
              ENDIF
        ELSE Fatal
        ENDIF)
{1}  IF null?(bl') ∨
      (reg_base(min_linear)(type_of(a')) ≤ a' ∧
       a' + length(bl') ≤ reg_size(max_linear)(type_of(a')))
    THEN IF Mem?(type_of(a'))
          THEN data(apply_side_effects(split(min_page, a', bl'),
                                       linear_read_side_effect_in_page)
                  (s'))
          ELSE data(memory_read_side_effect(pm_phy'mem)(a', bl', cp')(s'))
          ENDIF
    ELSE data(Fatal)
    ENDIF
    = bl'
```

Case splitting on null?(bl!1),

we get 2 subgoals:

linear_plain_read_side_effect_unchanged.1.1:

```

{-1} null?(bl')
{-2} Mem?(type_of(a'))
{-3} is_linear_plain_memory?(pm')
{-4} pm' 'states(s')
{-5} (address_block(a', length(bl')) ⊆ (pm' 'ro_addr ∪ pm' 'rw_addr))
{-6} OK?(IF null?(bl') ∨
      (reg_base(min_linear)(type_of(a')) ≤ a' ∧
       a' + length(bl') ≤ reg_size(max_linear)(type_of(a')))
      THEN IF Mem?(type_of(a'))
            THEN apply_side_effects(split(min_page, a', bl'),
                                     linear_read_side_effect_in_page)
                                     (s')
            ELSE memory_read_side_effect(pm_phy 'mem)(a', bl', cp')(s')
            ENDIF
      ELSE Fatal
      ENDIF)


---


{1} IF null?(bl') ∨
     (reg_base(min_linear)(type_of(a')) ≤ a' ∧
      a' + length(bl') ≤ reg_size(max_linear)(type_of(a')))
     THEN IF Mem?(type_of(a'))
           THEN data(apply_side_effects(split(min_page, a', bl'),
                                         linear_read_side_effect_in_page)
                    (s'))
           ELSE data(memory_read_side_effect(pm_phy 'mem)(a', bl', cp')(s'))
           ENDIF
     ELSE data(Fatal)
     ENDIF
     = bl'

```

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Expanding the definition of apply_side_effects,

Expanding the definition of ok_result,

Expanding the definition of reduce,

Using lemma split_null,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of linear_plain_read_side_effect_unchanged.1.1.

linear_plain_read_side_effect_unchanged.1.2:

```

{-1} Mem?(type_of(a'))
{-2} is_linear_plain_memory?(pm')
{-3} pm' 'states(s')
{-4} (address_block(a', length(bl')) ⊆ (pm' 'ro_addr ∪ pm' 'rw_addr))
{-5} OK?(IF null?(bl') ∨
      (reg_base(min_linear)(type_of(a')) ≤ a' ∧
       a' + length(bl') ≤ reg_size(max_linear)(type_of(a')))
      THEN IF Mem?(type_of(a'))
            THEN apply_side_effects(split(min_page, a', bl'),
                                     linear_read_side_effect_in_page)
                                     (s'))
            ELSE memory_read_side_effect(pm_phy 'mem)(a', bl', cp')(s')
            ENDIF
      ELSE Fatal
      ENDIF)
-----
{1} null?(bl')
{2} IF null?(bl') ∨
      (reg_base(min_linear)(type_of(a')) ≤ a' ∧
       a' + length(bl') ≤ reg_size(max_linear)(type_of(a')))
      THEN IF Mem?(type_of(a'))
            THEN data(apply_side_effects(split(min_page, a', bl'),
                                          linear_read_side_effect_in_page)
                    (s'))
            ELSE data(memory_read_side_effect(pm_phy 'mem)(a', bl', cp')(s'))
            ENDIF
      ELSE data(Fatal)
      ENDIF
      = bl'

```

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Using lemma `apply_side_effects_same_result`,

Case splitting on `offset(a1) < max_linear_offset`,

we get 2 subgoals:

linear_plain_read_side_effect_unchanged.1.2.1:

```

{-1} offset(a') < max_linear_offset
{-2} is_linear_plain_memory?(pm') ∧
      pm' 'states(s') ∧
      (∀ (s1: (pm' 'states)):
        every(λ (e: [Memory_Address_4G, list[Byte]]):
          OK?(linear_read_side_effect_in_page(e)(s1))
            (split(min_page, a', bl'))
        )
      )
      ∧
      (∀ (s1: (pm' 'states)):
        every(λ (e: [Memory_Address_4G, list[Byte]]):
          OK?(linear_read_side_effect_in_page(e)(s1)) ⊃
            data(linear_read_side_effect_in_page(e)(s1) = e'2
              (split(min_page, a', bl'))
            )
        )
      )
      ∧
      every(λ (e: [Memory_Address_4G, list[Byte]]):
        transformer_invariant?(pm' 'states,
          singleton(expr_2_super(linear_read_side_effect_in_page
            (e))))
      )
      (split(min_page, a', bl'))
      ⊃
      data(apply_side_effects(split(min_page, a', bl'), linear_read_side_effect_in_page)(s'))
      =
      reduce(null,
        λ (e: [Address, list[Byte]], tail: list[Byte]):
          LET head = e'2 IN append(head, tail)
        )
      (split(min_page, a', bl'))
{-3} Mem?(type_of(a'))
{-4} is_linear_plain_memory?(pm')
{-5} pm' 'states(s')
{-6} (address_block(a', length(bl')) ⊆ (pm' 'ro_addr ∪ pm' 'rw_addr))
{-7} reg_base(min_linear)(type_of(a')) ≤ a'
{-8} a' + length(bl') ≤ reg_size(max_linear)(type_of(a'))
{-9} OK?(apply_side_effects(split(min_page, a', bl'), linear_read_side_effect_in_page)(s'))
-----
{1} null?(bl')
{2} data(apply_side_effects(split(min_page, a', bl'), linear_read_side_effect_in_page)(s'))
      = bl'

```

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 4 subgoals:

linear_plain_read_side_effect_unchanged.1.2.1.1:

{-1}	offset(a') < max_linear_offset
{-2}	is_linear_plain_memory?(pm')
{-3}	pm' 'states(s')
{-4}	data(apply_side_effects(split(min_page, a' , bl'), linear_read_side_effect_in_page)(s')) = reduce(null, $\lambda (e: [\text{Address}, \text{list}[\text{Byte}]], \text{tail}: \text{list}[\text{Byte}]): \text{append}(e'2, \text{tail}))$ (split(min_page, a' , bl'))
{-5}	Mem?(type_of(a'))
{-6}	(address_block(a' , length(bl')) \subseteq (pm' 'ro_addr \cup pm' 'rw_addr))
{-7}	reg_base(min_linear)(type_of(a')) $\leq a'$
{-8}	$a' + \text{length}(bl') \leq \text{reg_size}(\text{max_linear})(\text{type_of}(a'))$
{-9}	OK?(apply_side_effects(split(min_page, a' , bl'), linear_read_side_effect_in_page)(s'))
{1}	null?(bl')
{2}	data(apply_side_effects(split(min_page, a' , bl'), linear_read_side_effect_in_page)(s')) = bl'

Replacing using formula -4,

Using lemma split_pair_concat,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of linear_plain_read_side_effect_unchanged.1.2.1.1.

linear_plain_read_side_effect_unchanged.1.2.1.2:

{-1}	offset(a') < max_linear_offset
{-2}	is_linear_plain_memory?(pm')
{-3}	pm' 'states(s')
{-4}	Mem?(type_of(a'))
{-5}	(address_block(a' , length(bl')) \subseteq (pm' 'ro_addr \cup pm' 'rw_addr))
{-6}	reg_base(min_linear)(type_of(a')) $\leq a'$
{-7}	$a' + \text{length}(bl') \leq \text{reg_size}(\text{max_linear})(\text{type_of}(a'))$
{-8}	OK?(apply_side_effects(split(min_page, a' , bl'), linear_read_side_effect_in_page)(s'))
{1}	every($\lambda (e: [\text{Memory_Address_4G}, \text{list}[\text{Byte}]]):$ transformer_invariant?(pm' 'states, singleton(expr_2_super(linear_read_side_effect_in_page(e)))) (split(min_page, a' , bl'))
{2}	null?(bl')
{3}	data(apply_side_effects(split(min_page, a' , bl'), linear_read_side_effect_in_page)(s')) = bl'

Rewriting using split_linear_read_side_effects_states, matching in *,

Keeping (-4 -6 1) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of linear_plain_read_side_effect_unchanged.1.2.1.2.

`linear_plain_read_side_effect_unchanged.1.2.1.3:`

{-1}	<code>offset(a') < max_linear_offset</code>
{-2}	<code>is_linear_plain_memory?(pm')</code>
{-3}	<code>pm' 'states(s')</code>
{-4}	<code>Mem?(type_of(a'))</code>
{-5}	<code>(address_block(a', length(bl')) ⊆ (pm' 'ro_addr ∪ pm' 'rw_addr))</code>
{-6}	<code>reg_base(min_linear)(type_of(a')) ≤ a'</code>
{-7}	<code>a' + length(bl') ≤ reg_size(max_linear)(type_of(a'))</code>
{-8}	<code>OK?(apply_side_effects(split(min_page, a', bl'), linear_read_side_effect_in_page)(s'))</code>
{1}	$\forall (s_1: (pm' 'states)):$ $\text{every}(\lambda (e: [\text{Memory_Address_4G}, \text{list}[\text{Byte}]]):$ $\quad \text{OK?}(\text{linear_read_side_effect_in_page}(e)(s_1)) \supset$ $\quad \text{data}(\text{linear_read_side_effect_in_page}(e)(s_1)) = e'2)$ $\quad (\text{split}(\text{min_page}, a', \text{bl}'))$
{2}	<code>null?(bl')</code>
{3}	<code>data(apply_side_effects(split(min_page, a', bl'), linear_read_side_effect_in_page)(s'))</code> <code>= bl'</code>

Repeatedly Skolemizing and flattening,

Using lemma `split_linear_read_side_effects_data`,

we get 2 subgoals:

`linear_plain_read_side_effect_unchanged.1.2.1.3.1:`

{-1}	<code>is_linear_plain_memory?(pm') ∧</code> <code>(address_block(a', length(bl')) ⊆ (pm' 'ro_addr ∪ pm' 'rw_addr)) ∧</code> <code>pm' 'states(s'_1)</code> \supset $\text{every}(\lambda (e: [\text{Memory_Address_4G}, \text{list}[\text{Byte}]]):$ $\quad \text{OK?}(\text{linear_read_side_effect_in_page}(e)(s'_1)) \supset$ $\quad \text{data}(\text{linear_read_side_effect_in_page}(e)(s'_1)) = e'2)$ $\quad (\text{split}(\text{min_page}, a', \text{bl}'))$
{-2}	<code>pm' 'states(s'_1)</code>
{-3}	<code>offset(a') < max_linear_offset</code>
{-4}	<code>is_linear_plain_memory?(pm')</code>
{-5}	<code>pm' 'states(s')</code>
{-6}	<code>Mem?(type_of(a'))</code>
{-7}	<code>(address_block(a', length(bl')) ⊆ (pm' 'ro_addr ∪ pm' 'rw_addr))</code>
{-8}	<code>reg_base(min_linear)(type_of(a')) ≤ a'</code>
{-9}	<code>a' + length(bl') ≤ reg_size(max_linear)(type_of(a'))</code>
{-10}	<code>OK?(apply_side_effects(split(min_page, a', bl'), linear_read_side_effect_in_page)(s'))</code>
{1}	$\text{every}(\lambda (e: [\text{Memory_Address_4G}, \text{list}[\text{Byte}]]):$ $\quad \text{OK?}(\text{linear_read_side_effect_in_page}(e)(s'_1)) \supset$ $\quad \text{data}(\text{linear_read_side_effect_in_page}(e)(s'_1)) = e'2)$ $\quad (\text{split}(\text{min_page}, a', \text{bl}'))$
{2}	<code>null?(bl')</code>
{3}	<code>data(apply_side_effects(split(min_page, a', bl'), linear_read_side_effect_in_page)(s'))</code> <code>= bl'</code>

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `linear_plain_read_side_effect_unchanged.1.2.1.3.1`.

linear_plain_read_side_effect_unchanged.1.2.1.3.2:

{-1}	pm' 'states(s')
{-2}	offset(a') < max_linear_offset
{-3}	is_linear_plain_memory?(pm')
{-4}	pm' 'states(s')
{-5}	Mem?(type_of(a'))
{-6}	(address_block(a', length(bl'))) \subseteq (pm' 'ro_addr \cup pm' 'rw_addr)
{-7}	reg_base(min_linear)(type_of(a')) \leq a'
{-8}	a' + length(bl') \leq reg_size(max_linear)(type_of(a'))
{-9}	OK?(apply_side_effects(split(min_page, a', bl'), linear_read_side_effect_in_page)(s'))
{1}	0 \leq a' 'offset
{2}	every(λ (e: [Memory_Address_4G, list[Byte]]): OK?(linear_read_side_effect_in_page(e)(s')) \supset data(linear_read_side_effect_in_page(e)(s')) = e'2 (split(min_page, a', bl'))
{3}	null?(bl')
{4}	data(apply_side_effects(split(min_page, a', bl'), linear_read_side_effect_in_page)(s')) = bl'

Keeping (-5 -7 1) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of linear_plain_read_side_effect_unchanged.1.2.1.3.2.

linear_plain_read_side_effect_unchanged.1.2.1.4:

{-1}	offset(a') < max_linear_offset
{-2}	is_linear_plain_memory?(pm')
{-3}	pm' 'states(s')
{-4}	Mem?(type_of(a'))
{-5}	(address_block(a', length(bl'))) \subseteq (pm' 'ro_addr \cup pm' 'rw_addr)
{-6}	reg_base(min_linear)(type_of(a')) \leq a'
{-7}	a' + length(bl') \leq reg_size(max_linear)(type_of(a'))
{-8}	OK?(apply_side_effects(split(min_page, a', bl'), linear_read_side_effect_in_page)(s'))
{1}	\forall (s ₁ : (pm' 'states)): every(λ (e: [Memory_Address_4G, list[Byte]]): OK?(linear_read_side_effect_in_page(e)(s ₁))) (split(min_page, a', bl'))
{2}	null?(bl')
{3}	data(apply_side_effects(split(min_page, a', bl'), linear_read_side_effect_in_page)(s')) = bl'

Repeatedly Skolemizing and flattening,

Using lemma split_linear_read_side_effects_ok,

we get 2 subgoals:

linear_plain_read_side_effect_unchanged.1.2.1.4.1:

{-1}	is_linear_plain_memory?(pm') \wedge (address_block(a', length(bl')) \subseteq (pm'ro_addr \cup pm'rw_addr)) \wedge pm'states(s' ₁) \supset every(λ (e: [Memory_Address_4G, list[Byte]]): OK?(linear_read_side_effect_in_page(e)(s' ₁))) (split(min_page, a', bl'))
{-2}	pm'states(s' ₁)
{-3}	offset(a') < max_linear_offset
{-4}	is_linear_plain_memory?(pm')
{-5}	pm'states(s')
{-6}	Mem?(type_of(a'))
{-7}	(address_block(a', length(bl')) \subseteq (pm'ro_addr \cup pm'rw_addr))
{-8}	reg_base(min_linear)(type_of(a')) \leq a'
{-9}	a' + length(bl') \leq reg_size(max_linear)(type_of(a'))
{-10}	OK?(apply_side_effects(split(min_page, a', bl'), linear_read_side_effect_in_page)(s'))
{1}	every(λ (e: [Memory_Address_4G, list[Byte]]): OK?(linear_read_side_effect_in_page(e)(s' ₁))) (split(min_page, a', bl'))
{2}	null?(bl')
{3}	data(apply_side_effects(split(min_page, a', bl'), linear_read_side_effect_in_page)(s')) = bl'

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of linear_plain_read_side_effect_unchanged.1.2.1.4.1.

linear_plain_read_side_effect_unchanged.1.2.1.4.2:

{-1}	pm'states(s' ₁)
{-2}	offset(a') < max_linear_offset
{-3}	is_linear_plain_memory?(pm')
{-4}	pm'states(s')
{-5}	Mem?(type_of(a'))
{-6}	(address_block(a', length(bl')) \subseteq (pm'ro_addr \cup pm'rw_addr))
{-7}	reg_base(min_linear)(type_of(a')) \leq a'
{-8}	a' + length(bl') \leq reg_size(max_linear)(type_of(a'))
{-9}	OK?(apply_side_effects(split(min_page, a', bl'), linear_read_side_effect_in_page)(s'))
{1}	0 \leq a'offset
{2}	every(λ (e: [Memory_Address_4G, list[Byte]]): OK?(linear_read_side_effect_in_page(e)(s' ₁))) (split(min_page, a', bl'))
{3}	null?(bl')
{4}	data(apply_side_effects(split(min_page, a', bl'), linear_read_side_effect_in_page)(s')) = bl'

Keeping (-5 -7 1) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of linear_plain_read_side_effect_unchanged.1.2.1.4.2.

linear_plain_read_side_effect_unchanged.1.2.2:

{-1}	$\begin{aligned} & \text{is_linear_plain_memory?}(pm') \wedge \\ & pm' \text{'states}(s') \wedge \\ & (\forall (s_1: (pm' \text{'states})): \\ & \quad \text{every}(\lambda (e: [\text{Memory_Address_4G}, \text{list}[\text{Byte}]]): \\ & \quad \quad \text{OK?}(\text{linear_read_side_effect_in_page}(e)(s_1)) \\ & \quad \quad (\text{split}(\text{min_page}, a', bl')))) \\ & \wedge \\ & (\forall (s_1: (pm' \text{'states})): \\ & \quad \text{every}(\lambda (e: [\text{Memory_Address_4G}, \text{list}[\text{Byte}]]): \\ & \quad \quad \text{OK?}(\text{linear_read_side_effect_in_page}(e)(s_1)) \supset \\ & \quad \quad \text{data}(\text{linear_read_side_effect_in_page}(e)(s_1)) = e'2) \\ & \quad \quad (\text{split}(\text{min_page}, a', bl')))) \\ & \wedge \\ & \text{every}(\lambda (e: [\text{Memory_Address_4G}, \text{list}[\text{Byte}]]): \\ & \quad \text{transformer_invariant?}(pm' \text{'states}, \\ & \quad \quad \text{singleton}(\text{expr_2_super}(\text{linear_read_side_effect_in_page} \\ & \quad \quad \quad (e)))))) \\ & (\text{split}(\text{min_page}, a', bl')) \\ & \supset \\ & \text{data}(\text{apply_side_effects}(\text{split}(\text{min_page}, a', bl'), \text{linear_read_side_effect_in_page})(s')) \\ & = \\ & \text{reduce}(\text{null}, \\ & \quad \lambda (e: [\text{Address}, \text{list}[\text{Byte}]], \text{tail}: \text{list}[\text{Byte}]): \\ & \quad \quad \text{LET head} = e'2 \text{ IN append}(\text{head}, \text{tail}) \\ & \quad \quad (\text{split}(\text{min_page}, a', bl')) \end{aligned}$
{-2}	Mem?(type_of(a'))
{-3}	is_linear_plain_memory?(pm')
{-4}	pm' 'states(s')
{-5}	(address_block(a', length(bl')) \subseteq (pm' 'ro_addr \cup pm' 'rw_addr))
{-6}	reg_base(min_linear)(type_of(a')) \leq a'
{-7}	a' + length(bl') \leq reg_size(max_linear)(type_of(a'))
{-8}	OK?(apply_side_effects(split(min_page, a', bl'), linear_read_side_effect_in_page)(s'))
{1}	offset(a') < max_linear_offset
{2}	null?(bl')
{3}	data(apply_side_effects(split(min_page, a', bl'), linear_read_side_effect_in_page)(s')) = bl'

Keeping (-2 -5 -6 -7 1 2) and hiding *,

Expanding the definition of length,

Installing automatic rewrites from: Mem max_linear reg_size

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of linear_plain_read_side_effect_unchanged.1.2.2.

linear_plain_read_side_effect_unchanged.2:

```

{-1} is_linear_plain_memory?(pm')
{-2} pm' `states(s')
{-3} (address_block(a', length(bl')) ⊆ (pm' `ro_addr ∪ pm' `rw_addr))
{-4} OK?(IF null?(bl') ∨
      (reg_base(min_linear)(type_of(a')) ≤ a' ∧
       a' + length(bl') ≤ reg_size(max_linear)(type_of(a')))
      THEN IF Mem?(type_of(a'))
            THEN apply_side_effects(split(min_page, a', bl'),
                                     linear_read_side_effect_in_page)
                                     (s')
            ELSE memory_read_side_effect(pm_phy `mem)(a', bl', cp')(s')
            ENDIF
      ELSE Fatal
      ENDIF)
-----
{1} Mem?(type_of(a'))
{2} IF null?(bl') ∨
      (reg_base(min_linear)(type_of(a')) ≤ a' ∧
       a' + length(bl') ≤ reg_size(max_linear)(type_of(a')))
      THEN IF Mem?(type_of(a'))
            THEN data(apply_side_effects(split(min_page, a', bl'),
                                     linear_read_side_effect_in_page)
                                     (s'))
            ELSE data(memory_read_side_effect(pm_phy `mem)(a', bl', cp')(s'))
            ENDIF
      ELSE data(Fatal)
      ENDIF
      = bl'

```

Using lemma pm_memory_addr,

Case splitting on null?(bl!1),

we get 2 subgoals:

linear_plain_read_side_effect_unchanged.2.1:

<pre> {-1} null?(bl') {-2} is_linear_plain_memory?(pm') ∧ union(pm'ro_addr, pm'rw_addr)(a') ⊃ in_memory(min_linear, max_linear)(a') ∧ Mem?(type_of(a')) {-3} is_linear_plain_memory?(pm') {-4} pm'states(s') {-5} (address_block(a', length(bl')) ⊆ (pm'ro_addr ∪ pm'rw_addr)) {-6} OK?(IF null?(bl') ∨ (reg_base(min_linear)(type_of(a')) ≤ a' ∧ a' + length(bl') ≤ reg_size(max_linear)(type_of(a'))) THEN IF Mem?(type_of(a')) THEN apply_side_effects(split(min_page, a', bl'), linear_read_side_effect_in_page (s')) ELSE memory_read_side_effect(pm_phy'mem)(a', bl', cp')(s') ENDIF ELSE Fatal ENDIF) </pre>	<hr style="border: 0.5px solid black;"/> <pre> {1} Mem?(type_of(a')) {2} IF null?(bl') ∨ (reg_base(min_linear)(type_of(a')) ≤ a' ∧ a' + length(bl') ≤ reg_size(max_linear)(type_of(a'))) THEN IF Mem?(type_of(a')) THEN data(apply_side_effects(split(min_page, a', bl'), linear_read_side_effect_in_page (s'))) ELSE data(memory_read_side_effect(pm_phy'mem)(a', bl', cp')(s')) ENDIF ELSE data(Fatal) ENDIF = bl' </pre>
--	--

Simplifying, rewriting, and recording with decision procedures,

Using lemma pm_plain_phy,

Using lemma pm_states,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Using lemma plain_memory_side_effect_content_unchanged_read_block,

Expanding the definition of side_effect_content_unchanged,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 2 subgoals:

C Proof scripts

`linear_plain_read_side_effect_unchanged.2.1.1:`

<pre> {-1} ∀ (s: Physical_memory, a: Address, bl: list[Byte], cp: bool): pm_phy'states(s) ∧ (address_block(a, length(bl)) ⊆ address_block(a', length(bl'))) ∧ OK?(memory_read_side_effect(pm_phy'mem)(a, bl, cp)(s)) ⊃ data(memory_read_side_effect(pm_phy'mem)(a, bl, cp)(s)) = bl {-2} is_linear_plain_memory?(pm') {-3} pm' states = pm_phy' states {-4} plain_memory?(pm_phy) {-5} null?(bl') {-6} pm' states(s') {-7} (address_block(a', length(bl'))) ⊆ (pm'ro_addr ∪ pm'rw_addr) {-8} OK?(memory_read_side_effect(pm_phy'mem)(a', bl', cp')(s')) </pre>	<pre> {1} union(pm'ro_addr, pm'rw_addr)(a') {2} Mem?(type_of(a')) {3} data(memory_read_side_effect(pm_phy'mem)(a', bl', cp')(s')) = bl' </pre>
--	---

Instantiating quantified variables,

Rewriting using `subset_reflexive`, matching in `*`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `linear_plain_read_side_effect_unchanged.2.1.1`.

`linear_plain_read_side_effect_unchanged.2.1.2:`

<pre> {-1} is_linear_plain_memory?(pm') {-2} pm' states = pm_phy' states {-3} plain_memory?(pm_phy) {-4} null?(bl') {-5} pm' states(s') {-6} (address_block(a', length(bl'))) ⊆ (pm'ro_addr ∪ pm'rw_addr) {-7} OK?(memory_read_side_effect(pm_phy'mem)(a', bl', cp')(s')) </pre>	<pre> {1} (address_block(a', length(bl'))) ⊆ (pm_phy'ro_addr ∪ pm_phy'rw_addr) {2} union(pm'ro_addr, pm'rw_addr)(a') {3} Mem?(type_of(a')) {4} data(memory_read_side_effect(pm_phy'mem)(a', bl', cp')(s')) = bl' </pre>
---	---

Hiding formulas: (-1 -2 -3 -7 4),

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `linear_plain_read_side_effect_unchanged.2.1.2`.

linear_plain_read_side_effect_unchanged.2.2:

```

{-1} is_linear_plain_memory?(pm') ∧ union(pm'ro_addr, pm'rw_addr)(a') ⊃
      in_memory(min_linear, max_linear)(a') ∧ Mem?(type_of(a'))
{-2} is_linear_plain_memory?(pm')
{-3} pm'states(s')
{-4} (address_block(a', length(bl')) ⊆ (pm'ro_addr ∪ pm'rw_addr))
{-5} OK?(IF null?(bl') ∨
      (reg_base(min_linear)(type_of(a')) ≤ a' ∧
       a' + length(bl') ≤ reg_size(max_linear)(type_of(a')))
      THEN IF Mem?(type_of(a'))
            THEN apply_side_effects(split(min_page, a', bl'),
                                   linear_read_side_effect_in_page)
                                   (s')
            ELSE memory_read_side_effect(pm_phy'mem)(a', bl', cp')(s')
            ENDIF
      ELSE Fatal
      ENDIF)


---


{1} null?(bl')
{2} Mem?(type_of(a'))
{3} IF null?(bl') ∨
      (reg_base(min_linear)(type_of(a')) ≤ a' ∧
       a' + length(bl') ≤ reg_size(max_linear)(type_of(a')))
      THEN IF Mem?(type_of(a'))
            THEN data(apply_side_effects(split(min_page, a', bl'),
                                   linear_read_side_effect_in_page)
                                   (s'))
            ELSE data(memory_read_side_effect(pm_phy'mem)(a', bl', cp')(s'))
            ENDIF
      ELSE data(Fatal)
      ENDIF
      = bl'

```

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 Keeping (-3 1 2) and hiding *,
 Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of linear_plain_read_side_effect_unchanged.2.2.
 Q.E.D.

C.116.8 Linear_Memory_Blessing_Properties.linear_plain_write_side_effect_unchanged

Terse proof for linear_plain_write_side_effect_unchanged.

linear_plain_write_side_effect_unchanged:

```

{1} ∃ (pm: Plain_Memory [Physical_memory]):
      is_linear_plain_memory?(pm) ⊃
      side_effect_content_unchanged(pm'rw_addr, pm'states, mem-
      ory_write_side_effect(pm'mem))

```

Expanding the definition of side_effect_content_unchanged,
 Repeatedly Skolemizing and flattening,
 Hiding formulas: -1,

C Proof scripts

Rewriting using `pm_write_side_effect_linear`, matching in `*`,

Expanding the definition of `linear_write_side_effect`,

Case splitting on `Mem?(type_of(a!1))`,

we get 2 subgoals:

`linear_plain_write_side_effect_unchanged.1:`

```

{-1} Mem?(type_of(a'))
{-2} is_linear_plain_memory?(pm')
{-3} pm' 'states(s')
{-4} (address_block(a', length(bl')) ⊆ pm'rw_addr)
{-5} OK?(IF null?(bl') ∨
      (reg_base(min_linear)(type_of(a')) ≤ a' ∧
       a' + length(bl') ≤ reg_size(max_linear)(type_of(a')))
      THEN IF Mem?(type_of(a'))
            THEN apply_side_effects(split(min_page, a', bl'),
                                     linear_write_side_effect_in_page)
                                     (s')
            ELSE memory_write_side_effect(pm_phy'mem)(a', bl', cp')(s')
            ENDIF
      ELSE Fatal
      ENDIF)
-----
{1} IF null?(bl') ∨
    (reg_base(min_linear)(type_of(a')) ≤ a' ∧
     a' + length(bl') ≤ reg_size(max_linear)(type_of(a')))
    THEN IF Mem?(type_of(a'))
          THEN data(apply_side_effects(split(min_page, a', bl'),
                                       linear_write_side_effect_in_page)
                  (s'))
          ELSE data(memory_write_side_effect(pm_phy'mem)(a', bl', cp')(s'))
          ENDIF
    ELSE data(Fatal)
    ENDIF
    = bl'

```

Case splitting on `null?(bl!1)`,

we get 2 subgoals:

linear_plain_write_side_effect_unchanged.1.1:

<pre> {-1} null?(bl') {-2} Mem?(type_of(a')) {-3} is_linear_plain_memory?(pm') {-4} pm' 'states(s') {-5} (address_block(a', length(bl')) ⊆ pm'rw_addr) {-6} OK?(IF null?(bl') ∨ (reg_base(min_linear)(type_of(a')) ≤ a' ∧ a' + length(bl') ≤ reg_size(max_linear)(type_of(a'))) THEN IF Mem?(type_of(a')) THEN apply_side_effects(split(min_page, a', bl'), linear_write_side_effect_in_page (s')) ELSE memory_write_side_effect(pm_phy'mem)(a', bl', cp')(s') ENDIF ELSE Fatal ENDIF) </pre>	<hr/> <pre> {1} IF null?(bl') ∨ (reg_base(min_linear)(type_of(a')) ≤ a' ∧ a' + length(bl') ≤ reg_size(max_linear)(type_of(a'))) THEN IF Mem?(type_of(a')) THEN data(apply_side_effects(split(min_page, a', bl'), linear_write_side_effect_in_page (s'))) ELSE data(memory_write_side_effect(pm_phy'mem)(a', bl', cp')(s')) ENDIF ELSE data(Fatal) ENDIF = bl' </pre>
--	--

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Expanding the definition of apply_side_effects,

Expanding the definition of ok_result,

Expanding the definition of reduce,

Using lemma split_null,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of linear_plain_write_side_effect_unchanged.1.1.

linear_plain_write_side_effect_unchanged.1.2:

```

{-1} Mem?(type_of(a'))
{-2} is_linear_plain_memory?(pm')
{-3} pm' `states(s')
{-4} (address_block(a', length(bl')) ⊆ pm' `rw_addr)
{-5} OK?(IF null?(bl') ∨
      (reg_base(min_linear)(type_of(a')) ≤ a' ∧
       a' + length(bl') ≤ reg_size(max_linear)(type_of(a'))))
      THEN IF Mem?(type_of(a'))
            THEN apply_side_effects(split(min_page, a', bl'),
                                     linear_write_side_effect_in_page)
                                     (s')
            ELSE memory_write_side_effect(pm_phy `mem)(a', bl', cp')(s')
            ENDIF
      ELSE Fatal
      ENDIF)
-----
{1} null?(bl')
{2} IF null?(bl') ∨
      (reg_base(min_linear)(type_of(a')) ≤ a' ∧
       a' + length(bl') ≤ reg_size(max_linear)(type_of(a'))))
      THEN IF Mem?(type_of(a'))
            THEN data(apply_side_effects(split(min_page, a', bl'),
                                           linear_write_side_effect_in_page)
                    (s'))
            ELSE data(memory_write_side_effect(pm_phy `mem)(a', bl', cp')(s'))
            ENDIF
      ELSE data(Fatal)
      ENDIF
      = bl'

```

Expanding the definition of reg_base,

Expanding the definition of <=,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Expanding the definition of min_linear,

Expanding the definition of Mem,

Using lemma apply_side_effects_same_result,

Case splitting on offset(a!1) < max_linear_offset,

we get 2 subgoals:

linear_plain_write_side_effect_unchanged.1.2.1:

```

{-1}  offset(a') < max_linear_offset
{-2}  is_linear_plain_memory?(pm') ^
      pm' 'states(s') ^
      (∀ (s1: (pm' 'states)):
        every(λ (e: [Memory_Address_4G, list[Byte]]):
          OK?(linear_write_side_effect_in_page(e)(s1))
            (split(min_page, a', bl'))))
      ^
      (∀ (s1: (pm' 'states)):
        every(λ (e: [Memory_Address_4G, list[Byte]]):
          OK?(linear_write_side_effect_in_page(e)(s1)) ⊃
            data(linear_write_side_effect_in_page(e)(s1)) = e'2
              (split(min_page, a', bl'))))
      ^
      every(λ (e: [Memory_Address_4G, list[Byte]]):
        transformer_invariant?(pm' 'states,
          singleton(expr_2_super(linear_write_side_effect_in_page
            (e))))))
      (split(min_page, a', bl'))
      ⊃
      data(apply_side_effects(split(min_page, a', bl'), linear_write_side_effect_in_page
        (s')))
      =
      reduce(null,
        λ (e: [Address, list[Byte]], tail: list[Byte]):
          LET head = e'2 IN append(head, tail))
        (split(min_page, a', bl'))
{-3}  Mem?(type_of(a'))
{-4}  is_linear_plain_memory?(pm')
{-5}  pm' 'states(s')
{-6}  (address_block(a', length(bl')) ⊆ pm' 'rw_addr)
{-7}  Mem_ = a' 'type_of
{-8}  0 ≤ a' 'offset
{-9}  (a' + length(bl')) 'type_of = reg_size(max_linear)(type_of(a')) 'type_of
{-10} (a' + length(bl')) 'offset ≤ reg_size(max_linear)(type_of(a')) 'offset
{-11} OK?(apply_side_effects(split(min_page, a', bl'), linear_write_side_effect_in_page)(s'))
-----
{1}  null?(bl')
{2}  data(apply_side_effects(split(min_page, a', bl'), linear_write_side_effect_in_page)(s'))
      = bl'

```

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 4 subgoals:

C Proof scripts

`linear_plain_write_side_effect_unchanged.1.2.1.1:`

{-1}	<code>offset(a') < max_linear_offset</code>
{-2}	<code>is_linear_plain_memory?(pm')</code>
{-3}	<code>pm' 'states(s')</code>
{-4}	<code>data(apply_side_effects(split(min_page, a', bl'), linear_write_side_effect_in_page)(s'))</code> <code>=</code> <code>reduce(null, λ (e: [Address, list[Byte]], tail: list[Byte]): append(e'2, tail)</code> <code>(split(min_page, a', bl'))</code>
{-5}	<code>Mem?(type_of(a'))</code>
{-6}	<code>(address_block(a', length(bl')) ⊆ pm'rw_addr)</code>
{-7}	<code>0 ≤ a'offset</code>
{-8}	<code>(a' + length(bl'))'type_of = reg_size(max_linear)(type_of(a'))'type_of</code>
{-9}	<code>(a' + length(bl'))'offset ≤ reg_size(max_linear)(type_of(a'))'offset</code>
{-10}	<code>OK?(apply_side_effects(split(min_page, a', bl'), linear_write_side_effect_in_page)(s'))</code>
{1}	<code>null?(bl')</code>
{2}	<code>data(apply_side_effects(split(min_page, a', bl'), linear_write_side_effect_in_page)(s'))</code> <code>= bl'</code>

Replacing using formula -4,

Using lemma `split_pair_concat`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `linear_plain_write_side_effect_unchanged.1.2.1.1`.

`linear_plain_write_side_effect_unchanged.1.2.1.2:`

{-1}	<code>offset(a') < max_linear_offset</code>
{-2}	<code>is_linear_plain_memory?(pm')</code>
{-3}	<code>pm' 'states(s')</code>
{-4}	<code>Mem?(type_of(a'))</code>
{-5}	<code>(address_block(a', length(bl')) ⊆ pm'rw_addr)</code>
{-6}	<code>0 ≤ a'offset</code>
{-7}	<code>(a' + length(bl'))'type_of = reg_size(max_linear)(type_of(a'))'type_of</code>
{-8}	<code>(a' + length(bl'))'offset ≤ reg_size(max_linear)(type_of(a'))'offset</code>
{-9}	<code>OK?(apply_side_effects(split(min_page, a', bl'), linear_write_side_effect_in_page)(s'))</code>
{1}	<code>every(λ (e: [Memory_Address_4G, list[Byte]]):</code> <code>transformer_invariant?(pm' 'states,</code> <code>singleton(expr_2_super(linear_write_side_effect_in_page(e))),</code> <code>(split(min_page, a', bl'))</code>
{2}	<code>null?(bl')</code>
{3}	<code>data(apply_side_effects(split(min_page, a', bl'), linear_write_side_effect_in_page)(s'))</code> <code>= bl'</code>

Rewriting using `split_linear_write_side_effects_states`, matching in `*`,

This completes the proof of `linear_plain_write_side_effect_unchanged.1.2.1.2`.

linear_plain_write_side_effect_unchanged.1.2.1.3:

{-1}	offset(a') < max_linear_offset
{-2}	is_linear_plain_memory?(pm')
{-3}	pm' 'states(s')
{-4}	Mem?(type_of(a'))
{-5}	(address_block(a' , length(bl')) \subseteq pm' 'rw_addr)
{-6}	$0 \leq a'$ 'offset
{-7}	($a' + \text{length}(bl')$)'type_of = reg_size(max_linear)(type_of(a'))'type_of
{-8}	($a' + \text{length}(bl')$)'offset \leq reg_size(max_linear)(type_of(a'))'offset
{-9}	OK?(apply_side_effects(split(min_page, a' , bl'), linear_write_side_effect_in_page)(s'))
{1}	$\forall (s_1: (pm'$ 'states)): every($\lambda (e: [\text{Memory_Address_4G}, \text{list}[\text{Byte}]]):$ OK?(linear_write_side_effect_in_page(e)(s_1)) \supset data(linear_write_side_effect_in_page(e)(s_1)) = e '2) (split(min_page, a' , bl'))
{2}	null?(bl')
{3}	data(apply_side_effects(split(min_page, a' , bl'), linear_write_side_effect_in_page)(s')) = bl'

Repeatedly Skolemizing and flattening,

Using lemma split_linear_write_side_effects_data,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of linear_plain_write_side_effect_unchanged.1.2.1.3.

linear_plain_write_side_effect_unchanged.1.2.1.4:

{-1}	offset(a') < max_linear_offset
{-2}	is_linear_plain_memory?(pm')
{-3}	pm' 'states(s')
{-4}	Mem?(type_of(a'))
{-5}	(address_block(a' , length(bl')) \subseteq pm' 'rw_addr)
{-6}	$0 \leq a'$ 'offset
{-7}	($a' + \text{length}(bl')$)'type_of = reg_size(max_linear)(type_of(a'))'type_of
{-8}	($a' + \text{length}(bl')$)'offset \leq reg_size(max_linear)(type_of(a'))'offset
{-9}	OK?(apply_side_effects(split(min_page, a' , bl'), linear_write_side_effect_in_page)(s'))
{1}	$\forall (s_1: (pm'$ 'states)): every($\lambda (e: [\text{Memory_Address_4G}, \text{list}[\text{Byte}]]):$ OK?(linear_write_side_effect_in_page(e)(s_1))) (split(min_page, a' , bl'))
{2}	null?(bl')
{3}	data(apply_side_effects(split(min_page, a' , bl'), linear_write_side_effect_in_page)(s')) = bl'

Repeatedly Skolemizing and flattening,

Using lemma split_linear_write_side_effects_ok,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of linear_plain_write_side_effect_unchanged.1.2.1.4.

linear_plain_write_side_effect_unchanged.1.2.2:

<pre> {-1} is_linear_plain_memory?(pm') ∧ pm' 'states(s') ∧ (∀ (s₁: (pm' 'states)): every(λ (e: [Memory_Address_4G, list[Byte]]): OK?(linear_write_side_effect_in_page(e)(s₁)) (split(min_page, a', bl')))) ∧ (∀ (s₁: (pm' 'states)): every(λ (e: [Memory_Address_4G, list[Byte]]): OK?(linear_write_side_effect_in_page(e)(s₁)) ⊃ data(linear_write_side_effect_in_page(e)(s₁)) = e'2 (split(min_page, a', bl')))) ∧ every(λ (e: [Memory_Address_4G, list[Byte]]): transformer_invariant?(pm' 'states, singleton(expr_2_super(linear_write_side_effect_in_page (e)))) (split(min_page, a', bl')) ⊃ data(apply_side_effects(split(min_page, a', bl'), linear_write_side_effect_in_page (s')) = reduce(null, λ (e: [Address, list[Byte]], tail: list[Byte]): LET head = e'2 IN append(head, tail) (split(min_page, a', bl')) </pre>	<pre> {-2} Mem?(type_of(a')) {-3} is_linear_plain_memory?(pm') {-4} pm' 'states(s') {-5} (address_block(a', length(bl')) ⊆ pm' 'rw_addr) {-6} Mem_ = a' 'type_of {-7} 0 ≤ a' 'offset {-8} (a' + length(bl')) 'type_of = reg_size(max_linear)(type_of(a')) 'type_of {-9} (a' + length(bl')) 'offset ≤ reg_size(max_linear)(type_of(a')) 'offset {-10} OK?(apply_side_effects(split(min_page, a', bl'), linear_write_side_effect_in_page)(s')) </pre> <hr style="border: 0.5px solid black;"/> <pre> {1} offset(a') < max_linear_offset {2} null?(bl') {3} data(apply_side_effects(split(min_page, a', bl'), linear_write_side_effect_in_page)(s')) = bl' </pre>
---	--

Keeping (-6 -9 1 2) and hiding *,

Expanding the definition of length,

Installing automatic rewrites from: Mem max_linear

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of linear_plain_write_side_effect_unchanged.1.2.2.

linear_plain_write_side_effect_unchanged.2:

```

{-1} is_linear_plain_memory?(pm')
{-2} pm' 'states(s')
{-3} (address_block(a', length(bl')) ⊆ pm'rw_addr)
{-4} OK?(IF null?(bl') ∨
      (reg_base(min_linear)(type_of(a')) ≤ a' ∧
       a' + length(bl') ≤ reg_size(max_linear)(type_of(a')))
      THEN IF Mem?(type_of(a'))
            THEN apply_side_effects(split(min_page, a', bl'),
                                     linear_write_side_effect_in_page)
                                     (s')
            ELSE memory_write_side_effect(pm_phy 'mem)(a', bl', cp')(s')
            ENDIF
      ELSE Fatal
      ENDIF)
-----
{1} Mem?(type_of(a'))
{2} IF null?(bl') ∨
    (reg_base(min_linear)(type_of(a')) ≤ a' ∧
     a' + length(bl') ≤ reg_size(max_linear)(type_of(a')))
    THEN IF Mem?(type_of(a'))
          THEN data(apply_side_effects(split(min_page, a', bl'),
                                       linear_write_side_effect_in_page)
                   (s'))
          ELSE data(memory_write_side_effect(pm_phy 'mem)(a', bl', cp')(s'))
          ENDIF
    ELSE data(Fatal)
    ENDIF
= bl'

```

Using lemma pm_memory_addr,

Case splitting on null?(bl1),

we get 2 subgoals:

linear_plain_write_side_effect_unchanged.2.1:

```

{-1} null?(bl')
{-2} is_linear_plain_memory?(pm') ∧ union(pm'ro_addr, pm'rw_addr)(a') ⊃
    in_memory(min_linear, max_linear)(a') ∧ Mem?(type_of(a'))
{-3} is_linear_plain_memory?(pm')
{-4} pm'states(s')
{-5} (address_block(a', length(bl')) ⊆ pm'rw_addr)
{-6} OK?(IF null?(bl') ∨
    (reg_base(min_linear)(type_of(a')) ≤ a' ∧
     a' + length(bl') ≤ reg_size(max_linear)(type_of(a')))
    THEN IF Mem?(type_of(a'))
        THEN apply_side_effects(split(min_page, a', bl'),
                                linear_write_side_effect_in_page
                                (s'))
        ELSE memory_write_side_effect(pm_phy'mem)(a', bl', cp')(s')
    ENDIF
    ELSE Fatal
    ENDIF)
-----
{1} Mem?(type_of(a'))
{2} IF null?(bl') ∨
    (reg_base(min_linear)(type_of(a')) ≤ a' ∧
     a' + length(bl') ≤ reg_size(max_linear)(type_of(a')))
    THEN IF Mem?(type_of(a'))
        THEN data(apply_side_effects(split(min_page, a', bl'),
                                      linear_write_side_effect_in_page
                                      (s')))
        ELSE data(memory_write_side_effect(pm_phy'mem)(a', bl', cp')(s'))
    ENDIF
    ELSE data(Fatal)
    ENDIF
    = bl'

```

Simplifying, rewriting, and recording with decision procedures,

Using lemma pm_plain_phy,

Using lemma pm_states,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Using lemma plain_memory_side_effect_content_unchanged_write_block,

Expanding the definition of side_effect_content_unchanged,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 2 subgoals:

linear_plain_write_side_effect_unchanged.2.1.1:

{-1}	$\forall (s: \text{Physical_memory}, a: \text{Address}, \text{bl}: \text{list}[\text{Byte}], \text{cp}: \text{bool}):$ $\text{pm_phy}'\text{states}(s) \wedge$ $(\text{address_block}(a, \text{length}(\text{bl})) \subseteq \text{address_block}(a', \text{length}(\text{bl}')) \wedge$ $\text{OK}?(\text{memory_write_side_effect}(\text{pm_phy}'\text{mem})(a, \text{bl}, \text{cp})(s))$ $\supset \text{data}(\text{memory_write_side_effect}(\text{pm_phy}'\text{mem})(a, \text{bl}, \text{cp})(s)) = \text{bl}$
{-2}	$\text{is_linear_plain_memory}?(\text{pm}')$
{-3}	$\text{pm}'\text{states} = \text{pm_phy}'\text{states}$
{-4}	$\text{plain_memory}?(\text{pm_phy})$
{-5}	$\text{null}?(\text{bl}')$
{-6}	$\text{pm}'\text{states}(s')$
{-7}	$(\text{address_block}(a', \text{length}(\text{bl}')) \subseteq \text{pm}'\text{rw_addr})$
{-8}	$\text{OK}?(\text{memory_write_side_effect}(\text{pm_phy}'\text{mem})(a', \text{bl}', \text{cp}')(s'))$
{1}	$\text{union}(\text{pm}'\text{ro_addr}, \text{pm}'\text{rw_addr})(a')$
{2}	$\text{Mem}?(\text{type_of}(a'))$
{3}	$\text{data}(\text{memory_write_side_effect}(\text{pm_phy}'\text{mem})(a', \text{bl}', \text{cp}')(s')) = \text{bl}'$

Instantiating quantified variables,

Rewriting using subset_reflexive, matching in *,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of linear_plain_write_side_effect_unchanged.2.1.1.

linear_plain_write_side_effect_unchanged.2.1.2:

{-1}	$\text{is_linear_plain_memory}?(\text{pm}')$
{-2}	$\text{pm}'\text{states} = \text{pm_phy}'\text{states}$
{-3}	$\text{plain_memory}?(\text{pm_phy})$
{-4}	$\text{null}?(\text{bl}')$
{-5}	$\text{pm}'\text{states}(s')$
{-6}	$(\text{address_block}(a', \text{length}(\text{bl}')) \subseteq \text{pm}'\text{rw_addr})$
{-7}	$\text{OK}?(\text{memory_write_side_effect}(\text{pm_phy}'\text{mem})(a', \text{bl}', \text{cp}')(s'))$
{1}	$(\text{address_block}(a', \text{length}(\text{bl}')) \subseteq \text{pm_phy}'\text{rw_addr})$
{2}	$\text{union}(\text{pm}'\text{ro_addr}, \text{pm}'\text{rw_addr})(a')$
{3}	$\text{Mem}?(\text{type_of}(a'))$
{4}	$\text{data}(\text{memory_write_side_effect}(\text{pm_phy}'\text{mem})(a', \text{bl}', \text{cp}')(s')) = \text{bl}'$

Hiding formulas: (-1 -2 -3 -7 4),

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of linear_plain_write_side_effect_unchanged.2.1.2.

linear_plain_write_side_effect_unchanged.2.2:

{-1}	is_linear_plain_memory?(pm') \wedge union(pm'ro_addr, pm'rw_addr)(a') \supset in_memory(min_linear, max_linear)(a') \wedge Mem?(type_of(a'))
{-2}	is_linear_plain_memory?(pm')
{-3}	pm' states(s')
{-4}	(address_block(a', length(bl')) \subseteq pm'rw_addr)
{-5}	OK?(IF null?(bl') \vee (reg_base(min_linear)(type_of(a')) \leq a' \wedge a' + length(bl') \leq reg_size(max_linear)(type_of(a'))) THEN IF Mem?(type_of(a')) THEN apply_side_effects(split(min_page, a', bl'), linear_write_side_effect_in_page (s')) ELSE memory_write_side_effect(pm_phy'mem)(a', bl', cp')(s') ENDIF ELSE Fatal ENDIF)
{1}	null?(bl')
{2}	Mem?(type_of(a'))
{3}	IF null?(bl') \vee (reg_base(min_linear)(type_of(a')) \leq a' \wedge a' + length(bl') \leq reg_size(max_linear)(type_of(a'))) THEN IF Mem?(type_of(a')) THEN data(apply_side_effects(split(min_page, a', bl'), linear_write_side_effect_in_page (s')) ELSE data(memory_write_side_effect(pm_phy'mem)(a', bl', cp')(s')) ENDIF ELSE data(Fatal) ENDIF = bl'

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 Keeping (-3 1 2) and hiding *,
 Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of linear_plain_write_side_effect_unchanged.2.2.
 Q.E.D.

C.116.9

Linear_Memory_Blessing_Properties.linear_memory_plain_memory

Terse proof for linear_memory_plain_memory.

linear_memory_plain_memory:

{1}	\forall (pm: Plain_Memory[Physical_memory]): is_linear_plain_memory?(pm) \supset plain_memory?(pm)
-----	--

Expanding the definition of plain_memory?,

Repeatedly Skolemizing and flattening,

Installing automatic rewrites from: (linear_plain_transformers_ok! linear_plain_unchanged_memory_invariant!
 linear_plain_unchanged_memory_invariant_write! linear_plain_unchanged_memory_write_invariant! lin-
 ear_plain_changed_memory_invariant! linear_plain_read_side_effect_unchanged! linear_plain_write_side_effect_uncha

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `linear_memory_plain_memory`.
Q.E.D.

C.117 Proofs for Linear_Memory_Properties (challenge-linear.pvs)

C.117.1 Linear_Memory_Properties.pm_read_linear

Terse proof for `pm_read_linear`.

`pm_read_linear`:

$$\frac{\{1\} \quad \forall (\text{pm}: \text{Plain_Memory}[\text{Linear_memory}], a: \text{Address}):}{\text{is_linear_plain_memory?}(\text{pm}) \supset \text{memory_read}(\text{pm}'\text{mem})(a) = \text{linear_read}(a)}$$

Repeatedly Skolemizing and flattening,
Expanding the definition of `is_linear_plain_memory?`,
Applying disjunctive simplification to flatten sequent,
Expanding the definition of `linear_pm`,
Replacing using formula -1,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `pm_read_linear`.
Q.E.D.

C.117.2 Linear_Memory_Properties.pm_write_linear

Terse proof for `pm_write_linear`.

`pm_write_linear`:

$$\frac{\{1\} \quad \forall (\text{pm}: \text{Plain_Memory}[\text{Linear_memory}], a: \text{Address}, b: \text{Byte}):}{\text{is_linear_plain_memory?}(\text{pm}) \supset \text{memory_write}(\text{pm}'\text{mem})(a, b) = \text{linear_write}(a, b)}$$

Repeatedly Skolemizing and flattening,
Expanding the definition of `is_linear_plain_memory?`,
Applying disjunctive simplification to flatten sequent,
Replacing using formula -2,
Expanding the definition of `linear_pm`,
which is trivially true.
This completes the proof of `pm_write_linear`.
Q.E.D.

C.117.3 Linear_Memory_Properties.pm_read_side_effect_linear

Terse proof for `pm_read_side_effect_linear`.

`pm_read_side_effect_linear`:

$$\frac{\{1\} \quad \forall (\text{pm}: \text{Plain_Memory}[\text{Linear_memory}], a: \text{Address}, \text{bl}: \text{list}[\text{Byte}], \text{cp}: \text{bool}):}{\text{is_linear_plain_memory?}(\text{pm}) \supset \text{memory_read_side_effect}(\text{pm}'\text{mem})(a, \text{bl}, \text{cp}) = \text{linear_read_side_effect}(a, \text{bl}, \text{cp})}$$

Repeatedly Skolemizing and flattening,

C Proof scripts

Expanding the definition of `is_linear_plain_memory?`,
Applying disjunctive simplification to flatten sequent,
Replacing using formula -2,
Expanding the definition of `linear_pm`,
which is trivially true.
This completes the proof of `pm_read_side_effect_linear`.
Q.E.D.

C.117.4 Linear_Memory_Properties.pm_write_side_effect_linear

Terse proof for `pm_write_side_effect_linear`.

`pm_write_side_effect_linear`:

$$\{1\} \quad \forall (pm: \text{Plain_Memory}[\text{Linear_memory}], a: \text{Address}, bl: \text{list}[\text{Byte}], cp: \text{bool}):$$

$$\text{is_linear_plain_memory?}(pm) \supset$$
$$\text{memory_write_side_effect}(pm\text{'mem})(a, bl, cp) = \text{linear_write_side_effect}(a, bl, cp)$$

Repeatedly Skolemizing and flattening,
Expanding the definition of `is_linear_plain_memory?`,
Expanding the definition of `linear_pm`,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `pm_write_side_effect_linear`.
Q.E.D.

C.117.5 Linear_Memory_Properties.pm_states

Terse proof for `pm_states`.

`pm_states`:

$$\{1\} \quad \forall (pm: \text{Plain_Memory}[\text{Linear_memory}]):$$

$$\text{is_linear_plain_memory?}(pm) \supset pm\text{'states} = pm\text{'phy}\text{'states}$$

Expanding the definition of `is_linear_plain_memory?`,
Repeatedly Skolemizing and flattening,
This completes the proof of `pm_states`.
Q.E.D.

C.117.6 Linear_Memory_Properties.pm_plain_phy

Terse proof for `pm_plain_phy`.

`pm_plain_phy`:

$$\{1\} \quad \forall (pm: \text{Plain_Memory}[\text{Linear_memory}]): \text{is_linear_plain_memory?}(pm) \supset \text{plain_memory?}(pm\text{'phy})$$

Repeatedly Skolemizing and flattening,
Expanding the definition of `is_linear_plain_memory?`,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `pm_plain_phy`.
Q.E.D.

C.117.7 Linear_Memory_Properties.pm_memory_addr_TCC1

Terse proof for pm_memory_addr_TCC1.

pm_memory_addr_TCC1:

{1}	$\forall (pm: \text{Plain_Memory}[\text{Linear_memory}], a: \text{Address}):$ $\text{is_linear_plain_memory?}(pm) \wedge \text{union}(pm\text{'ro_addr}, pm\text{'rw_addr})(a) \supset$ $\text{Mem?}(\text{min_linear}'\text{type_of})$
-----	--

Repeatedly Skolemizing and flattening,
 Expanding the definition of min_linear,
 Expanding the definition of Mem,
 which is trivially true.

This completes the proof of pm_memory_addr_TCC1.

Q.E.D.

C.117.8 Linear_Memory_Properties.pm_memory_addr_TCC2

Terse proof for pm_memory_addr_TCC2.

pm_memory_addr_TCC2:

{1}	$\forall (pm: \text{Plain_Memory}[\text{Linear_memory}], a: \text{Address}):$ $\text{is_linear_plain_memory?}(pm) \wedge \text{union}(pm\text{'ro_addr}, pm\text{'rw_addr})(a) \supset$ $\text{Mem?}(\text{max_linear}'\text{type_of})$
-----	--

Repeatedly Skolemizing and flattening,
 Expanding the definition of max_linear,
 Expanding the definition of Mem,
 which is trivially true.

This completes the proof of pm_memory_addr_TCC2.

Q.E.D.

C.117.9 Linear_Memory_Properties.pm_memory_addr

Terse proof for pm_memory_addr.

pm_memory_addr:

{1}	$\forall (pm: \text{Plain_Memory}[\text{Linear_memory}], a: \text{Address}):$ $\text{is_linear_plain_memory?}(pm) \wedge \text{union}(pm\text{'ro_addr}, pm\text{'rw_addr})(a) \supset$ $\text{in_memory}(\text{min_linear}, \text{max_linear})(a) \wedge \text{Mem?}(\text{type_of}(a))$
-----	--

Expanding the definition of is_linear_plain_memory?,
 Repeatedly Skolemizing and flattening,
 Expanding the definition of max_linear,
 Expanding the definition of Mem,
 Instantiating quantified variables,
 Keeping (-13 1) and hiding *,
 Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of pm_memory_addr.

Q.E.D.

C.117.10 Linear_Memory_Properties.pm_linear_blessed

Terse proof for pm_linear_blessed.

pm_linear_blessed:

$$\{1\} \quad \forall (pm: \text{Plain_Memory}[\text{Linear_memory}], s: \text{Linear_memory}[\text{Physical_memory}, pm_phy]):$$

$$\text{is_linear_plain_memory?}(pm) \wedge pm' \text{states}(s) \supset$$

$$\text{linear_blessed?}(s, \text{restrict}[\text{Address}, \text{Memory_Address_4G}, \text{boolean}](pm' \text{ro_addr}),$$

$$\text{restrict}[\text{Address}, \text{Memory_Address_4G}, \text{boolean}](pm' \text{rw_addr}))$$

Expanding the definition of is_linear_plain_memory?,
 Repeatedly Skolemizing and flattening,
 Instantiating quantified variables,
 This completes the proof of pm_linear_blessed.
 Q.E.D.

C.117.11

Linear_Memory_Properties.pm_linear_resolve_read_ok_TCC1

Terse proof for pm_linear_resolve_read_ok_TCC1.

pm_linear_resolve_read_ok_TCC1:

$$\{1\} \quad \forall (pm: \text{Plain_Memory}[\text{Linear_memory}], s: (pm' \text{states}),$$

$$a: ((pm' \text{ro_addr} \cup pm' \text{rw_addr})):$$

$$\text{is_linear_plain_memory?}(pm) \supset$$

$$\text{Mem?}(a' \text{type_of}) \wedge 0 \leq a' \text{offset} \wedge a' \text{offset} < \text{max_linear_offset}$$

Repeatedly Skolemizing and flattening,
 Expanding the definition of is_linear_plain_memory?,
 Applying disjunctive simplification to flatten sequent,
 Instantiating quantified variables,
 This completes the proof of pm_linear_resolve_read_ok_TCC1.
 Q.E.D.

C.117.12 Linear_Memory_Properties.pm_linear_resolve_read_ok

Terse proof for pm_linear_resolve_read_ok.

pm_linear_resolve_read_ok:

$$\{1\} \quad \forall (pm: \text{Plain_Memory}[\text{Linear_memory}], s: (pm' \text{states}),$$

$$a: ((pm' \text{ro_addr} \cup pm' \text{rw_addr})):$$

$$\text{is_linear_plain_memory?}(pm) \supset \text{OK?}(\text{linear_resolve}(a, \text{Read})(s))$$

Repeatedly Skolemizing and flattening,
 Using lemma pm_linear_blessed,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 Expanding the definition of linear_blessed?,
 Applying disjunctive simplification to flatten sequent,
 Instantiating quantified variables,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 Keeping (-9 1) and hiding *,
 Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of pm_linear_resolve_read_ok.
Q.E.D.

C.117.13

Linear_Memory_Properties.pm_linear_resolve_write_ok_TCC1

Terse proof for pm_linear_resolve_write_ok_TCC1.

pm_linear_resolve_write_ok_TCC1:

$$\{1\} \quad \forall (pm: Plain_Memory[Linear_memory], s: (pm'states), a: (pm'rw_addr)):$$

$$\text{is_linear_plain_memory?}(pm) \supset$$

$$\text{Mem?}(a'\text{type_of}) \wedge 0 \leq a'\text{offset} \wedge a'\text{offset} < \text{max_linear_offset}$$

Expanding the definition of is_linear_plain_memory?,
Repeatedly Skolemizing and flattening,
Instantiating quantified variables,
Keeping (-1 1) and hiding *,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of pm_linear_resolve_write_ok_TCC1.
Q.E.D.

C.117.14 Linear_Memory_Properties.pm_linear_resolve_write_ok

Terse proof for pm_linear_resolve_write_ok.

pm_linear_resolve_write_ok:

$$\{1\} \quad \forall (pm: Plain_Memory[Linear_memory], s: (pm'states), a: (pm'rw_addr)):$$

$$\text{is_linear_plain_memory?}(pm) \supset \text{OK?}(\text{linear_resolve}(a, \text{Write})(s))$$

Repeatedly Skolemizing and flattening,
Using lemma pm_linear_blessed,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Expanding the definition of linear_blessed?,
Applying disjunctive simplification to flatten sequent,
Instantiating quantified variables,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Keeping (-9 1) and hiding *,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of pm_linear_resolve_write_ok.
Q.E.D.

C.117.15

Linear_Memory_Properties.pm_linear_resolve_reg_transformers_other

Terse proof for pm_linear_resolve_reg_transformers_other.

pm_linear_resolve_reg_transformers_other:

$$\{1\} \quad \forall (pm: Plain_Memory[Linear_memory]):$$

$$\text{is_linear_plain_memory?}(pm) \supset$$

$$(\text{linear_resolve_register_transformers} \subseteq \text{pm_phy'other_actions})$$

C Proof scripts

Expanding the definition of `is_linear_plain_memory?`,

Repeatedly Skolemizing and flattening,

This completes the proof of `pm_linear_resolve_reg_transformers_other`.

Q.E.D.

C.117.16 Linear_Memory_Properties.pm_unchanged_singleton_linear_resolve_reg_transformers

Terse proof for `pm_unchanged_singleton_linear_resolve_reg_transformers`.

`pm_unchanged_singleton_linear_resolve_reg_transformers`:

$\{1\} \quad \forall (pm: \text{Plain_Memory}[\text{Linear_memory}],$ $q: [\text{Physical_memory} \rightarrow \text{SuperResult}[\text{Physical_memory}]]):$ $\text{is_linear_plain_memory?}(pm) \wedge \text{linear_resolve_register_transformers}(q) \supset$ $\text{unchanged_memory_invariant?}(pm_phy\text{'mem}, pm_phy\text{'states}, \text{singleton}(q),$ $(pm_phy\text{'ro_addr} \cup pm_phy\text{'rw_addr}))$
--

Repeatedly Skolemizing and flattening,

Using lemma `pm_states`,

Using lemma `pm_plain_phy`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Expanding the definition of `is_linear_plain_memory?`,

Applying disjunctive simplification to flatten sequent,

Expanding the definition of `plain_memory?`,

Applying disjunctive simplification to flatten sequent,

Using lemma `unchanged_memory_invariant_mono`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 2 subgoals:

pm_unchanged_singleton_linear_resolve_reg_transformers.1:

- {-1} unchanged_memory_invariant?(pm_phy' mem, pm_phy' states,
((pm_phy'other_actions ∪ memory_read_transformers(pm_phy'mem, (pm_phy'ro_addr ∪ pm_phy'rw_addr)))
- {-2} pm'mem = linear_pm
- {-3} $\forall (s: (pm' states)): OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s))$
- {-4} $\forall (s_1, s_2: (pm' states)):$
data(read_data(pm_phy, segment_reg_data_type)(CS)(s₁)) =
data(read_data(pm_phy, segment_reg_data_type)(CS)(s₂))
- {-5} $\forall (s: (pm' states)): OK?(read_data(pm_phy, pdbr_data_type)(PDBR)(s))$
- {-6} $\forall (s_1, s_2: (pm' states)):$
data(read_data(pm_phy, pdbr_data_type)(PDBR)(s₁)) =
data(read_data(pm_phy, pdbr_data_type)(PDBR)(s₂))
- {-7} $\forall (s_1, s_2: (pm' states)):$
pe_in_pdir_range?(s₁,
restrict[Address, Memory_Address_4G, boolean]
((pm'ro_addr ∪ pm'rw_addr)))
=
pe_in_pdir_range?(s₂,
restrict[Address, Memory_Address_4G, boolean]
((pm'ro_addr ∪ pm'rw_addr)))
∧
pe_in_ptab_range?(s₁,
restrict[Address, Memory_Address_4G, boolean]
((pm'ro_addr ∪ pm'rw_addr)))
=
pe_in_ptab_range?(s₂,
restrict[Address, Memory_Address_4G, boolean]
((pm'ro_addr ∪ pm'rw_addr)))
- {-8} $\forall (s: (pm' states), lvl: Level,$
a:
pe_in_pt_range?(s,
restrict[Address, Memory_Address_4G, boolean]
((pm'ro_addr ∪ pm'rw_addr)),
lvl))):
OK?(read_data(pm_phy, paging_data_type(lvl))(a)(s))
- {-9} $\forall (s_1, s_2: (pm' states), lvl: Level,$
a:
pe_in_pt_range?(s₁,
restrict[Address, Memory_Address_4G, boolean]
((pm'ro_addr ∪ pm'rw_addr)),
lvl))):
set_reference(data(read_data(pm_phy, paging_data_type(lvl))(a)(s₁)), Write) =
set_reference(data(read_data(pm_phy, paging_data_type(lvl))(a)(s₂)), Write)
- {-10} (pm'other_actions ⊆ pm_phy'other_actions)
- {-11} (linear_resolve_register_transformers ⊆ pm_phy'other_actions)
- {-12} $\forall (a: ((pm'ro_addr \cup pm'rw_addr))):$
Mem?(type_of(a)) ∧ 0 ≤ offset(a) ∧ offset(a) < max_linear_offset
- {-13} $\forall (s: (pm' states)):$
linear_blessed?(s, restrict[Address, Memory_Address_4G, boolean](pm'ro_addr),
restrict[Address, Memory_Address_4G, boolean](pm'rw_addr))
- {-14} unchanged_memory_invariant?(pm_phy' mem, pm_phy' states,
memory_write_transformers(pm_phy'mem, pm_phy'rw_addr),
pm_phy'ro_addr)
- {-15} unchanged_memory_write_invariant?(pm_phy'mem, pm_phy'states, pm_phy'rw_addr)
- {-16} changed_memory_invariant?(pm_phy'mem, pm_phy'states, pm_phy'rw_addr)
- {-17} transformers_ok?(pm_phy'states, 1569
((memory_read_transformers(pm_phy'mem, (pm_phy'ro_addr ∪ pm_phy'rw_addr)) ∪ mem
- {-18} side_effect_content_unchanged((pm_phy'ro_addr ∪ pm_phy'rw_addr), pm_phy'states,
memory_read_side_effect(pm_phy'mem))
- {-19} side_effect_content_unchanged(pm_phy'rw_addr, pm_phy'states,
memory_write_side_effect(pm_phy'mem))
- {-20} pm'states = pm_phy'states
- {-21} linear_resolve_register_transformers(q')
- {1} ((pm_phy'ro_addr ∪ pm_phy'rw_addr) ⊆ (nm_phy'ro_addr ∪ nm_phy'rw_addr))

C Proof scripts

Rewriting using `subset_reflexive`, matching in `*`,

This completes the proof of `pm_unchanged_singleton_linear_resolve_reg_transformers.1`.

pm_unchanged_singleton_linear_resolve_reg_transformers.2:

- {-1} unchanged_memory_invariant?(pm_phy' mem, pm_phy' states,
((pm_phy'other_actions \cup memory_read_transformers(pm_phy' mem, (pm_phy'ro_addr \cup pm_phy'rw_addr)))
- {-2} pm' mem = linear_pm
- {-3} $\forall (s: (pm' states)): OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s))$
- {-4} $\forall (s_1, s_2: (pm' states)):$
data(read_data(pm_phy, segment_reg_data_type)(CS)(s₁)) =
data(read_data(pm_phy, segment_reg_data_type)(CS)(s₂))
- {-5} $\forall (s: (pm' states)): OK?(read_data(pm_phy, pdbr_data_type)(PDBR)(s))$
- {-6} $\forall (s_1, s_2: (pm' states)):$
data(read_data(pm_phy, pdbr_data_type)(PDBR)(s₁)) =
data(read_data(pm_phy, pdbr_data_type)(PDBR)(s₂))
- {-7} $\forall (s_1, s_2: (pm' states)):$
pe_in_pdir_range?(s₁,
restrict[Address, Memory_Address_4G, boolean]
((pm'ro_addr \cup pm'rw_addr)))
=
pe_in_pdir_range?(s₂,
restrict[Address, Memory_Address_4G, boolean]
((pm'ro_addr \cup pm'rw_addr)))
 \wedge
pe_in_ptab_range?(s₁,
restrict[Address, Memory_Address_4G, boolean]
((pm'ro_addr \cup pm'rw_addr)))
=
pe_in_ptab_range?(s₂,
restrict[Address, Memory_Address_4G, boolean]
((pm'ro_addr \cup pm'rw_addr)))
- {-8} $\forall (s: (pm' states), lvl: Level,$
a:
pe_in_pt_range?(s,
restrict[Address, Memory_Address_4G, boolean]
((pm'ro_addr \cup pm'rw_addr)),
lvl))):
OK?(read_data(pm_phy, paging_data_type(lvl))(a)(s))
- {-9} $\forall (s_1, s_2: (pm' states), lvl: Level,$
a:
pe_in_pt_range?(s₁,
restrict[Address, Memory_Address_4G, boolean]
((pm'ro_addr \cup pm'rw_addr)),
lvl))):
set_reference(data(read_data(pm_phy, paging_data_type(lvl))(a)(s₁)), Write) =
set_reference(data(read_data(pm_phy, paging_data_type(lvl))(a)(s₂)), Write)
- {-10} (pm'other_actions \subseteq pm_phy'other_actions)
- {-11} (linear_resolve_register_transformers \subseteq pm_phy'other_actions)
- {-12} $\forall (a: ((pm'ro_addr \cup pm'rw_addr)))$
Mem?(type_of(a)) \wedge 0 \leq offset(a) \wedge offset(a) < max_linear_offset
- {-13} $\forall (s: (pm' states)):$
linear_blessed?(s, restrict[Address, Memory_Address_4G, boolean](pm'ro_addr),
restrict[Address, Memory_Address_4G, boolean](pm'rw_addr))
- {-14} unchanged_memory_invariant?(pm_phy' mem, pm_phy' states,
memory_write_transformers(pm_phy' mem, pm_phy'rw_addr),
pm_phy'ro_addr)
- {-15} unchanged_memory_write_invariant?(pm_phy' mem, pm_phy' states, pm_phy'rw_addr)
- {-16} changed_memory_invariant?(pm_phy' mem, pm_phy' states, pm_phy'rw_addr)
- {-17} transformers_ok?(pm_phy' states,
((memory_read_transformers(pm_phy' mem, (pm_phy'ro_addr \cup pm_phy'rw_addr)) \cup mem
1571
- {-18} side_effect_content_unchanged((pm_phy'ro_addr \cup pm_phy'rw_addr), pm_phy' states,
memory_read_side_effect(pm_phy' mem))
- {-19} side_effect_content_unchanged(pm_phy'rw_addr, pm_phy' states,
memory_write_side_effect(pm_phy' mem))
- {-20} pm' states = pm_phy' states
- {-21} linear_resolve_register_transformers(q')
- {1} (singleton(q') \subseteq ((pm_phy'other_actions \cup memory_read_transformers(pm_phy' mem, (pm_phy'ro_addr \cup pm_phy'rw_addr)) \cup mem

C Proof scripts

Keeping (-11 -21 1) and hiding *,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `pm_unchanged_singleton_linear_resolve_reg_transformers.2`.
Q.E.D.

C.117.17

Linear_Memory_Properties.pm_address_in_pt_range_in_rw_addr

Terse proof for `pm_address_in_pt_range_in_rw_addr`.

`pm_address_in_pt_range_in_rw_addr`:

$$\frac{\{1\} \quad \forall (pm: \text{Plain_Memory}[\text{Linear_memory}], s: \text{Linear_memory}[\text{Physical_memory}, pm_phy]): \\ \text{is_linear_plain_memory?}(pm) \wedge pm' \text{states}(s) \supset \\ (\text{address_in_pt_range?}(s, \text{restrict} [\text{Address}, \text{Memory_Address_4G}, \text{boolean}]((pm'ro_addr \cup pm'r$$

Repeatedly Skolemizing and flattening,

Using lemma `pm_linear_blessed`,

Expanding the definition of `linear_blessed?`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Keeping (-7 1) and hiding *,

Case splitting on `union(restrict[Address, Memory_Address_4G, boolean](pm!1'ro_addr), restrict[Address, Memory_Address_4G, boolean](pm!1'rw_addr)) = restrict[Address, Memory_Address_4G, boolean] (union(pm!1'ro_addr, pm!1'rw_addr))`,

we get 2 subgoals:

`pm_address_in_pt_range_in_rw_addr.1`:

$$\frac{\begin{array}{l} \{-1\} \quad (\text{restrict} [\text{Address}, \text{Memory_Address_4G}, \text{boolean}] (pm'ro_addr) \cup \text{restrict} [\text{Address}, \text{Memory_Address_4G}, \text{boolean}] (pm'rw_addr)) \\ \quad = \text{restrict} [\text{Address}, \text{Memory_Address_4G}, \text{boolean}] ((pm'ro_addr \cup pm'rw_addr)) \\ \{-2\} \quad (\text{address_in_pt_range?}(s', (\text{restrict} [\text{Address}, \text{Memory_Address_4G}, \text{boolean}] (pm'ro_addr) \cup \text{restrict} [\text{Address}, \text{Memory_Address_4G}, \text{boolean}] (pm'rw_addr))) \\ \{1\} \quad (\text{address_in_pt_range?}(s', \text{restrict} [\text{Address}, \text{Memory_Address_4G}, \text{boolean}] ((pm'ro_addr \cup pm'rw_addr))) \end{array}}$$

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `pm_address_in_pt_range_in_rw_addr.1`.

`pm_address_in_pt_range_in_rw_addr.2`:

$$\frac{\begin{array}{l} \{-1\} \quad (\text{address_in_pt_range?}(s', (\text{restrict} [\text{Address}, \text{Memory_Address_4G}, \text{boolean}] (pm'ro_addr) \cup \text{restrict} [\text{Address}, \text{Memory_Address_4G}, \text{boolean}] (pm'rw_addr))) \\ \{1\} \quad (\text{restrict} [\text{Address}, \text{Memory_Address_4G}, \text{boolean}] (pm'ro_addr) \cup \text{restrict} [\text{Address}, \text{Memory_Address_4G}, \text{boolean}] (pm'rw_addr)) \\ \quad = \text{restrict} [\text{Address}, \text{Memory_Address_4G}, \text{boolean}] ((pm'ro_addr \cup pm'rw_addr)) \\ \{2\} \quad (\text{address_in_pt_range?}(s', \text{restrict} [\text{Address}, \text{Memory_Address_4G}, \text{boolean}] ((pm'ro_addr \cup pm'rw_addr))) \end{array}}$$

Keeping (1) and hiding *,

Applying `decompose-equality`,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `pm_address_in_pt_range_in_rw_addr.2`.

Q.E.D.

C.117.18

Linear_Memory_Properties.address_block_in_memory_TCC1

Terse proof for `address_block_in_memory_TCC1`.

address_block_in_memory_TCC1:

$\{1\} \quad \forall (pm: \text{Plain_Memory}[\text{Linear_memory}], a: \text{Address}, bl: \text{list}[\text{Byte}]):$ $\text{is_linear_plain_memory?}(pm) \wedge$ $(\text{address_block}(a, \text{length}(bl)) \subseteq (pm'ro_addr \cup pm'rw_addr)) \wedge \text{cons?}(bl)$ $\supset \text{Mem?}(\text{max_linear}'\text{type_of})$

Expanding the definition of max_linear,

Expanding the definition of Mem,

which is trivially true.

This completes the proof of address_block_in_memory_TCC1.

Q.E.D.

C.117.19 Linear_Memory_Properties.address_block_in_memory

Terse proof for address_block_in_memory.

address_block_in_memory:

$\{1\} \quad \forall (pm: \text{Plain_Memory}[\text{Linear_memory}], a: \text{Address}, bl: \text{list}[\text{Byte}]):$ $\text{is_linear_plain_memory?}(pm) \wedge$ $(\text{address_block}(a, \text{length}(bl)) \subseteq (pm'ro_addr \cup pm'rw_addr)) \wedge \text{cons?}(bl)$ $\supset a + \text{length}(bl) \leq \text{reg_size}(\text{max_linear})(\text{type_of}(a))$

Repeatedly Skolemizing and flattening,

Using lemma pm_memory_addr,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 2 subgoals:

address_block_in_memory.1:

$\{-1\} \quad \text{is_linear_plain_memory?}(pm')$ $\{-2\} \quad \text{in_memory}(\text{min_linear}, \text{max_linear})(a' + (\text{length}(bl') - 1))$ $\{-3\} \quad \text{Mem?}(\text{type_of}(a' + (\text{length}(bl') - 1)))$ $\{-4\} \quad \text{every}(\lambda (x: \text{number}):$ $\quad \text{number_field_pred}(x) \wedge$ $\quad \text{real_pred}(x) \wedge$ $\quad \text{rational_pred}(x) \wedge \text{integer_pred}(x) \wedge x \geq 0 \wedge x < \text{max_byte})$ $\quad (bl')$ $\{-5\} \quad (\text{address_block}(a', \text{length}(bl')) \subseteq (pm'ro_addr \cup pm'rw_addr))$ $\{-6\} \quad \text{cons?}(bl')$ <hr/> $\{1\} \quad a' + \text{length}(bl') \leq \text{reg_size}(\text{max_linear})(\text{type_of}(a'))$
--

Expanding the definition of +,

Keeping (-2 -3 1) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of address_block_in_memory.1.

address_block_in_memory.2:

{-1}	is_linear_plain_memory?(pm')	
{-2}	every($\lambda (x: \text{number})$: number_field_pred(x) \wedge real_pred(x) \wedge rational_pred(x) \wedge integer_pred(x) \wedge $x \geq 0 \wedge x < \text{max_byte}$) (bl')	
{-3}	(address_block(a', length(bl')) \subseteq (pm'ro_addr \cup pm'rw_addr))	
{-4}	cons?(bl')	
{1}		union(pm'ro_addr, pm'rw_addr)(a' + (length(bl') - 1))
{2}	a' + length(bl') \leq reg_size(max_linear)(type_of(a'))	

Keeping (-3 -4 1) and hiding *,
 Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of address_block_in_memory.2.
 Q.E.D.

C.117.20 Linear_Memory_Properties.address_block_in_memory2

Terse proof for address_block_in_memory2.

address_block_in_memory2:

{1}	\forall (pm: Plain_Memory[Linear_memory], a: Address, bl: list[Byte]): is_linear_plain_memory?(pm) \wedge (address_block(a, length(bl)) \subseteq (pm'ro_addr \cup pm'rw_addr)) \wedge cons?(bl) $\supset 0 \leq a'offset$
-----	---

Repeatedly Skolemizing and flattening,
 Hiding formulas: -1,
 Expanding the definition of subset?,
 Expanding the definition of member,
 Instantiating the top quantifier in -2 with the terms: (a!1),
 Expanding the definition of address_block,
 Expanding the definition of length,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 Using lemma pm_memory_addr,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 Keeping (-3 1) and hiding *,
 Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of address_block_in_memory2.
 Q.E.D.

C.117.21

Linear_Memory_Properties.raise_fault_transformer_invariant

Terse proof for raise_fault_transformer_invariant.

raise_fault_transformer_invariant:

{1}	\forall (pm: Plain_Memory [Linear_memory], priv: Memory_privilege, access: Memory_access, present: bool, pfa: Memory_Address_4G): is_linear_plain_memory?(pm) \supset transformer_invariant?(pm_phy 'states, singleton(expr_2_super(raise_fault(priv, access, present, pfa))))
-----	---

Repeatedly Skolemizing and flattening,

Expanding the definition of raise_fault,

Rewriting using expr_composition_transformer_invariant, matching in *,

we get 3 subgoals:

raise_fault_transformer_invariant.1:

{-1}	Mem?(pfa' 'type_of)	
{-2}	$0 \leq$ pfa' 'offset	
{-3}	pfa' 'offset < max_linear_offset	
{-4}	is_linear_plain_memory?(pm')	
{1}	transformer_invariant?(pm_phy 'states, singleton(expr_2_super(write_data(pm_phy, address_data_type) (CR2, pfa'))))	
{2}	transformer_invariant?(pm_phy 'states, singleton(expr_2_super(write_data(pm_phy, address_data_type) (CR2, pfa')) ## exception_result(Page_fault ((#reserved_bit_violation := FALSE, priv- ac- := (singleton [Memory present := present'#))))))	
	ilege := priv', cess	

Hiding formulas: 2,

Using lemma pm_states,

Using lemma pm_unchanged_singleton_linear_resolve_reg_transformers,

Using lemma unchanged_memory_invariant_invariant,

Expanding the definition of linear_resolve_register_transformers,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Instantiating quantified variables,

This completes the proof of raise_fault_transformer_invariant.1.

C Proof scripts

raise_fault_transformer_invariant.2:

```

{-1} Mem?(pfa' 'type_of)
{-2} 0 ≤ pfa' 'offset
{-3} pfa' 'offset < max_linear_offset
{-4} is_linear_plain_memory?(pm')
-----
{1} transformer_invariant?(pm_phy' 'states,
                               singleton(expr_2_super(exception_result(Page_fault
                                                                ((#reserved_bit_v
                                                                := FALSE,
                                                                privi-
                                                                ac-
                                                                := (singleton
                                                                present
                                                                := present' #

legge := priv',

cess

{2} transformer_invariant?(pm_phy' 'states,
                               singleton(expr_2_super(write_data(pm_phy, ad-
                               dress_data_type)
                               (CR2, pfa')
                               ##
                               exception_result(Page_fault
                                                                ((#reserved_bit_v
                                                                := FALSE,
                                                                priv-
                                                                ac-
                                                                := (singleton
                                                                present
                                                                := present' #

ilegge := priv',

cess

```

Keeping (1) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `raise_fault_transformer_invariant.2`.

raise_fault_transformer_invariant.3:

<pre> {-1} Mem?(pfa' 'type_of) {-2} 0 ≤ pfa' 'offset {-3} pfa' 'offset < max_linear_offset {-4} is_linear_plain_memory?(pm') </pre>	<pre> {1} (Read ∈ (singleton [Memory_access] (access') ∪ {Read})) {2} transformer_invariant?(pm_phy' states, singleton(expr_2_super(write_data(pm_phy, ad- dress_data_type) (CR2, pfa') ## exception_result(Page_fault ((#reserved_bit_violation := FALSE, priv- ac- := (singleton [Memory_ present := present'#)))))) </pre>
--	---

Keeping (1) and hiding *,
 Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of raise_fault_transformer_invariant.3.
 Q.E.D.

C.117.22

Linear_Memory_Properties.raise_fault_unchanged_memory_invariant

Terse proof for raise_fault_unchanged_memory_invariant.

raise_fault_unchanged_memory_invariant:

<pre> {1} ∀ (pm: Plain_Memory [Linear_memory], priv: Memory_privilege, access: Mem- ory_access, present: bool, pfa: Memory_Address_4G): is_linear_plain_memory?(pm) ⊃ unchanged_memory_invariant?(pm_phy' mem, pm_phy' states, singleton(expr_2_super(raise_fault(priv, cess, (pm_phy' ro_addr ∪ pm_phy' rw_addr)) </pre>	<pre> ac- present, pfa))), </pre>
---	---

Repeatedly Skolemizing and flattening,
 Expanding the definition of raise_fault,
 Using lemma plain_memory_unchanged_composition [Physical_memory, Unit, [bool, Address]],
 we get 2 subgoals:

raise_fault_unchanged_memory_invariant.1:

```

{-1} (plain_memory?(pm_phy) ∧
      ((pm_phy'ro_addr ∪ pm_phy'rw_addr) ⊆ (pm_phy'ro_addr ∪ pm_phy'rw_addr)) ∧
      (∀ (s: (pm_phy'states)): TRUE) ∧
      unchanged_memory_invariant?(pm_phy' mem, pm_phy' states,
      singleton(expr_2_super(write_data(pm_phy, ad-
      dress_data_type)
      (CR2, pfa')),
      (pm_phy'ro_addr ∪ pm_phy'rw_addr))
      ∧
      (∀ (d: (λ (u: Unit): TRUE)):
      unchanged_memory_invariant?(pm_phy' mem, pm_phy' states,
      singleton(expr_2_super(exception_result
      [Physical_memory,
      [bool, Ad-
      dress]])
      (Page_fault
      ((#reserved_bit_viol-
      := FALSE,
      privi-
      access
      := (singleton
      present
      := present'#)))
      (pm_phy'ro_addr ∪ pm_phy'rw_addr))))
      ⊃
      unchanged_memory_invariant?(pm_phy' mem, pm_phy' states,
      singleton(expr_2_super(write_data(pm_phy, ad-
      dress_data_type)
      (CR2, pfa')
      ##
      (λ (u: Unit):
      exception_result
      [Physical_memory, [bool,
      dress]])
      (Page_fault
      ((#reserved_bit_violation
      privi-
      access
      := (singleton [Mem
      present := present'
      (pm_phy'ro_addr ∪ pm_phy'rw_addr))
{-2} Mem?(pfa' type_of)
{-3} 0 ≤ pfa' offset
{-4} pfa' offset < max_linear_offset
{-5} is_linear_plain_memory?(pm')
-----
{1} unchanged_memory_invariant?(pm_phy' mem, pm_phy' states,
      singleton(expr_2_super(write_data(pm_phy, ad-
      dress_data_type)
      (CR2, pfa')
      ##
      excep-
      tion_result(Page_fault
      ((#reserved
      := FAI
      priv-
      ac-
      := (sin
      present
      := pn

```

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting, we get 5 subgoals:

raise_fault_unchanged_memory_invariant.1.1:

<pre>{-1} unchanged_memory_invariant?(pm_phy' mem, pm_phy' states, dress_data_type) singleton(expr_2_super(write_data(pm_phy, ad- dress))) lege := priv', (pm_phy'ro_addr ∪ pm_phy'rw_addr))</pre>	<pre>(CR2, pfa') ## (λ (u: Unit): exception_result [Physical_memory, [bool, Ad- (Page_fault ((#reserved_bit_violation := FALSE, privi- access := (singleton [Memory_acc present := present'#)))]), (pm_phy'ro_addr ∪ pm_phy'rw_addr))</pre>
<pre>{-2} Mem?(pfa' type_of) {-3} 0 ≤ pfa' offset {-4} pfa' offset < max_linear_offset {-5} is_linear_plain_memory?(pm')</pre>	
<pre>{1} unchanged_memory_invariant?(pm_phy' mem, pm_phy' states, dress_data_type) singleton(expr_2_super(write_data(pm_phy, ad- dress))) lege := priv', cess (pm_phy'ro_addr ∪ pm_phy'rw_addr))</pre>	<pre>(CR2, pfa') ## excep- ((#reserved_bit_violati := FALSE, priv- ac- := (singleton [M present := present'#)))])</pre>

Expanding the definition of ##,

Expanding the definition of ##,

which is trivially true.

This completes the proof of raise_fault_unchanged_memory_invariant.1.1.

C Proof scripts

`raise_fault_unchanged_memory_invariant.1.2:`

{-1}	Mem?(pfa' type_of)	
{-2}	$0 \leq \text{pfa}' \text{offset}$	
{-3}	$\text{pfa}' \text{offset} < \text{max_linear_offset}$	
{-4}	is_linear_plain_memory?(pm')	
{1}	$\forall (d: (\lambda (u: \text{Unit}): \text{TRUE})):$ unchanged_memory_invariant?(pm_phy' mem, pm_phy' states, singleton(expr_2_super(exception_result [Physical_memory, [bool, dress]]) lege := priv', (pm_phy'ro_addr \cup pm_phy'rw_addr)) unchanged_memory_invariant?(pm_phy' mem, pm_phy' states, singleton(expr_2_super(write_data(pm_phy, ad- dress_data_type) (CR2, pfa') ## excep- tion_result(Page_fault ((#reserved := FAI priv- ac- := (sin present := pre	
{2}	(pm_phy'ro_addr \cup pm_phy'rw_addr)) unchanged_memory_invariant?(pm_phy' mem, pm_phy' states, singleton(expr_2_super(write_data(pm_phy, ad- dress_data_type) (CR2, pfa') ## excep- tion_result(Page_fault ((#reserved := FAI priv- ac- := (sin present := pre	

Hiding formulas: 2,

Repeatedly Skolemizing and flattening,

Keeping (1) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `raise_fault_unchanged_memory_invariant.1.2`.

raise_fault_unchanged_memory_invariant.1.3:

<pre> {-1} Mem?(pfa' 'type_of) {-2} 0 ≤ pfa' 'offset {-3} pfa' 'offset < max_linear_offset {-4} is_linear_plain_memory?(pm') </pre>	<pre> {1} unchanged_memory_invariant?(pm_phy' mem, pm_phy' states, singleton(expr_2_super(write_data(pm_phy, ad- dress_data_type) (CR2, pfa'))), (pm_phy'ro_addr ∪ pm_phy'rw_addr)) {2} unchanged_memory_invariant?(pm_phy' mem, pm_phy' states, singleton(expr_2_super(write_data(pm_phy, ad- dress_data_type) (CR2, pfa') ## excep- tion_result(Page_fault ((#reserved_bit_violati := FALSE, priv- ac- := (singleton [M present := present'#)))))) (pm_phy'ro_addr ∪ pm_phy'rw_addr)) </pre>
--	--

Hiding formulas: 2,

Using lemma pm_unchanged_singleton_linear_resolve_reg_transformers,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Expanding the definition of linear_resolve_register_transformers,

Applying disjunctive simplification to flatten sequent,

Instantiating quantified variables,

This completes the proof of raise_fault_unchanged_memory_invariant.1.3.

raise_fault_unchanged_memory_invariant.1.4:

<pre> {-1} Mem?(pfa' 'type_of) {-2} 0 ≤ pfa' 'offset {-3} pfa' 'offset < max_linear_offset {-4} is_linear_plain_memory?(pm') </pre>	<pre> {1} ((pm_phy'ro_addr ∪ pm_phy'rw_addr) ⊆ (pm_phy'ro_addr ∪ pm_phy'rw_addr)) {2} unchanged_memory_invariant?(pm_phy' mem, pm_phy' states, singleton(expr_2_super(write_data(pm_phy, ad- dress_data_type) (CR2, pfa') ## excep- tion_result(Page_fault (#reserved := FA priv- ac- := (sin present := pre (pm_phy'ro_addr ∪ pm_phy'rw_addr)) </pre>
--	--

Rewriting using subset_reflexive, matching in *,

This completes the proof of raise_fault_unchanged_memory_invariant.1.4.

raise_fault_unchanged_memory_invariant.1.5:

<pre> {-1} Mem?(pfa' 'type_of) {-2} 0 ≤ pfa' 'offset {-3} pfa' 'offset < max_linear_offset {-4} is_linear_plain_memory?(pm') </pre>	<pre> {1} plain_memory?(pm_phy) {2} unchanged_memory_invariant?(pm_phy' mem, pm_phy' states, singleton(expr_2_super(write_data(pm_phy, ad- dress_data_type) (CR2, pfa') ## excep- tion_result(Page_fault (#reserved := FA priv- ac- := (sin present := pre (pm_phy'ro_addr ∪ pm_phy'rw_addr)) </pre>
--	--

Using lemma pm_plain_phy,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `raise_fault_unchanged_memory_invariant.1.5`.

`raise_fault_unchanged_memory_invariant.2`:

<pre> {-1} Mem?(pfa' type_of) {-2} 0 ≤ pfa' offset {-3} pfa' offset < max_linear_offset {-4} is_linear_plain_memory?(pm') </pre>	<pre> {1} (Read ∈ (singleton [Memory_access] (access') ∪ {Read})) {2} unchanged_memory_invariant?(pm_phy mem, pm_phy states, singleton(expr_2_super(write_data(pm_phy, ad- dress_data_type) (CR2, pfa') ## excep- tion_result(Page_fault ((#reserved_bit_violati := FALSE, priv- ac- := (singleton [M present := present'#)))))) (pm_phy ro_addr ∪ pm_phy rw_addr)) </pre>
---	--

Keeping (1) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `raise_fault_unchanged_memory_invariant.2`.

Q.E.D.

C.117.23 Linear_Memory_Properties.apply_side_effects_pm_states

Terse proof for `apply_side_effects_pm_states`.

`apply_side_effects_pm_states`:

<pre> {1} ∀ (pm: Plain_Memory [Linear_memory], l: list [[Memory_Address_4G, list [Byte]]], f: [Memory_Address_4G, list [Byte] → [Linear_memory → ExprResult [Linear_memory, list [Byte]]]): is_linear_plain_memory?(pm) ∧ every(λ (e: [Memory_Address_4G, list [Byte]]): transformer_invariant?(pm states, singleton(expr_2_super(f(e)))) (l) ⊃ transformer_invariant?(pm states, singleton(expr_2_super(apply_side_effects(l, f)))) </pre>	
--	--

Expanding the definition of `transformer_invariant?`,

Inducting on `l` on formula 1,

we get 2 subgoals:

C Proof scripts

`apply_side_effects_pm_states.1:`

$$\begin{array}{l}
 \{1\} \quad \forall (pm: \text{Plain_Memory}[\text{Linear_memory}], \\
 \quad f: \\
 \quad \quad [\text{Memory_Address_4G}, \text{list}[\text{Byte}] \rightarrow \\
 \quad \quad \quad [\text{Linear_memory} \rightarrow \text{ExprResult}[\text{Linear_memory}, \text{list}[\text{Byte}]]]]): \\
 \text{is_linear_plain_memory?}(pm) \wedge \\
 \text{every}(\lambda (e: [\text{Memory_Address_4G}, \text{list}[\text{Byte}]]): \\
 \quad \forall (s: \text{Linear_memory}[\text{Physical_memory}, pm_phy], \\
 \quad \quad q: \\
 \quad \quad \quad [\text{Linear_memory}[\text{Physical_memory}, pm_phy] \rightarrow \\
 \quad \quad \quad \quad \text{SuperResult}[\text{Physical_memory}]]): \\
 \quad \quad pm' \text{states}(s) \wedge \text{singleton}(\text{expr_2_super}(f(e)))(q) \supset \\
 \quad \quad \quad \text{result_pred}(pm' \text{states})(q(s))) \\
 \quad \quad (\text{null}) \\
 \quad \supset \\
 \quad (\forall (s: \text{Linear_memory}[\text{Physical_memory}, pm_phy], \\
 \quad \quad q: [\text{Linear_memory}[\text{Physical_memory}, pm_phy] \rightarrow \text{SuperRe-} \\
 \quad \quad \text{sult}[\text{Physical_memory}]]): \\
 \quad \quad pm' \text{states}(s) \wedge \text{singleton}(\text{expr_2_super}(\text{apply_side_effects}(\text{null}, f)))(q) \supset \\
 \quad \quad \quad \text{result_pred}(pm' \text{states})(q(s)))
 \end{array}$$

Repeatedly Skolemizing and flattening,

Expanding the definition of `apply_side_effects`,

Expanding the definition of `singleton`,

Replacing using formula -4,

Expanding the definition of `reduce`,

Installing automatic rewrites from: `result_pred expr_2_super expr_2_super_res ok_result`

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `apply_side_effects_pm_states.1`.

apply_side_effects_pm_states.2:

$$\begin{array}{l}
\{1\} \quad \forall (\text{cons1_var}: [\text{Memory_Address_4G}, \text{list}[\text{Byte}]], \\
\quad \text{cons2_var}: \text{list}[[\text{Memory_Address_4G}, \text{list}[\text{Byte}]]]): \\
\quad (\forall (\text{pm}: \text{Plain_Memory}[\text{Linear_memory}], \\
\quad \quad f: \\
\quad \quad \quad [\text{Memory_Address_4G}, \text{list}[\text{Byte}]] \rightarrow \\
\quad \quad \quad \quad [\text{Linear_memory} \rightarrow \text{ExprResult}[\text{Linear_memory}, \text{list}[\text{Byte}]]]): \\
\quad \text{is_linear_plain_memory?}(\text{pm}) \wedge \\
\quad \text{every}(\lambda (e: [\text{Memory_Address_4G}, \text{list}[\text{Byte}]]): \\
\quad \quad \forall (s: \text{Linear_memory}[\text{Physical_memory}, \text{pm_phy}], \\
\quad \quad \quad q: \\
\quad \quad \quad \quad [\text{Linear_memory}[\text{Physical_memory}, \text{pm_phy}] \rightarrow \\
\quad \quad \quad \quad \quad \text{SuperResult}[\text{Physical_memory}]): \\
\quad \quad \quad \quad \text{pm}'\text{states}(s) \wedge \text{singleton}(\text{expr_2_super}(f(e)))(q) \supset \\
\quad \quad \quad \quad \quad \text{result_pred}(\text{pm}'\text{states})(q(s))) \\
\quad \quad \quad \quad (\text{cons2_var}) \\
\quad \quad \supset \\
\quad \quad (\forall (s: \text{Linear_memory}[\text{Physical_memory}, \text{pm_phy}], \\
\quad \quad \quad q: [\text{Linear_memory}[\text{Physical_memory}, \text{pm_phy}] \rightarrow \text{SuperRe-} \\
\quad \quad \quad \text{sult}[\text{Physical_memory}]]): \\
\quad \quad \quad \quad \text{pm}'\text{states}(s) \wedge \\
\quad \quad \quad \quad \quad \text{singleton}(\text{expr_2_super}(\text{apply_side_effects}(\text{cons2_var}, f)))(q) \\
\quad \quad \quad \quad \quad \supset \text{result_pred}(\text{pm}'\text{states})(q(s))) \\
\quad \quad \supset \\
\quad \quad (\forall (\text{pm}: \text{Plain_Memory}[\text{Linear_memory}], \\
\quad \quad \quad f: \\
\quad \quad \quad \quad [\text{Memory_Address_4G}, \text{list}[\text{Byte}]] \rightarrow \\
\quad \quad \quad \quad \quad [\text{Linear_memory} \rightarrow \text{ExprResult}[\text{Linear_memory}, \text{list}[\text{Byte}]]]): \\
\quad \quad \text{is_linear_plain_memory?}(\text{pm}) \wedge \\
\quad \quad \text{every}(\lambda (e: [\text{Memory_Address_4G}, \text{list}[\text{Byte}]]): \\
\quad \quad \quad \forall (s: \text{Linear_memory}[\text{Physical_memory}, \text{pm_phy}], \\
\quad \quad \quad \quad q: \\
\quad \quad \quad \quad \quad [\text{Linear_memory}[\text{Physical_memory}, \text{pm_phy}] \rightarrow \\
\quad \quad \quad \quad \quad \quad \text{SuperResult}[\text{Physical_memory}]): \\
\quad \quad \quad \quad \quad \quad \text{pm}'\text{states}(s) \wedge \text{singleton}(\text{expr_2_super}(f(e)))(q) \supset \\
\quad \quad \quad \quad \quad \quad \quad \text{result_pred}(\text{pm}'\text{states})(q(s))) \\
\quad \quad \quad \quad \quad (\text{cons}(\text{cons1_var}, \text{cons2_var})) \\
\quad \quad \quad \supset \\
\quad \quad \quad (\forall (s: \text{Linear_memory}[\text{Physical_memory}, \text{pm_phy}], \\
\quad \quad \quad \quad q: \\
\quad \quad \quad \quad \quad [\text{Linear_memory}[\text{Physical_memory}, \text{pm_phy}] \rightarrow \text{SuperRe-} \\
\quad \quad \quad \quad \quad \text{sult}[\text{Physical_memory}]]): \\
\quad \quad \quad \quad \quad \text{pm}'\text{states}(s) \wedge \\
\quad \quad \quad \quad \quad \quad \text{singleton}(\text{expr_2_super}(\text{apply_side_effects}(\text{cons}(\text{cons1_var}, \text{cons2_var}), f)))(q) \\
\quad \quad \quad \quad \quad \quad \supset \text{result_pred}(\text{pm}'\text{states})(q(s))))
\end{array}$$

Repeatedly Skolemizing and flattening,

Instantiating quantified variables,

Expanding the definition of every,

C Proof scripts

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 Instantiating the top quantifier in -3 with the terms: (s!1 expr_2_super(apply_side_effects(cons2_var!1, f!1))),
 Expanding the definition of singleton,
 Replacing using formula -6,
 Hiding formulas: -6,
 Keeping (-3 -4 -5 1) and hiding *,
 Expanding the definition of apply_side_effects,
 Expanding the definition of reduce,
 Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of `apply_side_effects_pm_states.2`.
 Q.E.D.

C.117.24 Linear_Memory_Properties.apply_side_effects_ok

Terse proof for `apply_side_effects_ok`.

`apply_side_effects_ok`:

$$\begin{array}{l}
 \{1\} \quad \forall (pm: \text{Plain_Memory}[\text{Linear_memory}], s: \text{Linear_memory}[\text{Physical_memory}, pm_phy], \\
 \quad l: \text{list}[[\text{Memory_Address_4G}, \text{list}[\text{Byte}]]], \\
 \quad f: \\
 \quad \quad [\text{Memory_Address_4G}, \text{list}[\text{Byte}]] \rightarrow \\
 \quad \quad \quad [\text{Linear_memory} \rightarrow \text{ExprResult}[\text{Linear_memory}, \text{list}[\text{Byte}]]]): \\
 \text{is_linear_plain_memory?}(pm) \wedge \\
 pm' \text{states}(s) \wedge \\
 (\forall (s_1: (pm' \text{states})): \\
 \quad \text{every}(\lambda (e: [\text{Memory_Address_4G}, \text{list}[\text{Byte}]]): \text{OK?}(f(e)(s_1)))(l)) \\
 \wedge \\
 \text{every}(\lambda (e: [\text{Memory_Address_4G}, \text{list}[\text{Byte}]]): \\
 \quad \text{transformer_invariant?}(pm' \text{states}, \text{singleton}(\text{expr_2_super}(f(e)))) \\
 \quad (l)) \\
 \supset \text{OK?}(\text{apply_side_effects}(l, f)(s))
 \end{array}$$

Inducting on l on formula 1,

we get 2 subgoals:

`apply_side_effects_ok.1`:

$$\begin{array}{l}
 \{1\} \quad \forall (pm: \text{Plain_Memory}[\text{Linear_memory}], s: \text{Linear_memory}[\text{Physical_memory}, pm_phy], \\
 \quad f: \\
 \quad \quad [\text{Memory_Address_4G}, \text{list}[\text{Byte}]] \rightarrow \\
 \quad \quad \quad [\text{Linear_memory} \rightarrow \text{ExprResult}[\text{Linear_memory}, \text{list}[\text{Byte}]]]): \\
 \text{is_linear_plain_memory?}(pm) \wedge \\
 pm' \text{states}(s) \wedge \\
 (\forall (s_1: (pm' \text{states})): \\
 \quad \text{every}(\lambda (e: [\text{Memory_Address_4G}, \text{list}[\text{Byte}]]): \text{OK?}(f(e)(s_1)))(\text{null})) \\
 \wedge \\
 \text{every}(\lambda (e: [\text{Memory_Address_4G}, \text{list}[\text{Byte}]]): \\
 \quad \text{transformer_invariant?}(pm' \text{states}, \text{singleton}(\text{expr_2_super}(f(e)))) \\
 \quad (\text{null})) \\
 \supset \text{OK?}(\text{apply_side_effects}(\text{null}, f)(s))
 \end{array}$$

Repeatedly Skolemizing and flattening,

Expanding the definition of `apply_side_effects`,

Expanding the definition of `ok_result`,

Expanding the definition of `reduce`,

which is trivially true.

This completes the proof of `apply_side_effects_ok.1`.

`apply_side_effects_ok.2`:

```

{1}  ∃ (cons1_var: [Memory_Address_4G, list[Byte]],
      cons2_var: list[[Memory_Address_4G, list[Byte]]]):
  (∃ (pm: Plain_Memory[Linear_memory], s: Linear_memory[Physical_memory, pm_phy],
    f:
      [Memory_Address_4G, list[Byte]] →
        [Linear_memory → ExprResult[Linear_memory, list[Byte]]]):
    is_linear_plain_memory?(pm) ∧
    pm'states(s) ∧
    (∃ (s1: (pm'states)):
      every(λ (e: [Memory_Address_4G, list[Byte]]): OK?(f(e)(s1)))
        (cons2_var))
    ∧
    every(λ (e: [Memory_Address_4G, list[Byte]]):
      transformer_invariant?(pm'states, singleton(expr_2_super(f(e))))
        (cons2_var))
    ⊃ OK?(apply_side_effects(cons2_var, f)(s)))
  ⊃
  (∃ (pm: Plain_Memory[Linear_memory], s: Linear_memory[Physical_memory, pm_phy],
    f:
      [Memory_Address_4G, list[Byte]] →
        [Linear_memory → ExprResult[Linear_memory, list[Byte]]]):
    is_linear_plain_memory?(pm) ∧
    pm'states(s) ∧
    (∃ (s1: (pm'states)):
      every(λ (e: [Memory_Address_4G, list[Byte]]): OK?(f(e)(s1)))
        (cons(cons1_var, cons2_var)))
    ∧
    every(λ (e: [Memory_Address_4G, list[Byte]]):
      transformer_invariant?(pm'states, singleton(expr_2_super(f(e))))
        (cons(cons1_var, cons2_var))
    ⊃ OK?(apply_side_effects(cons(cons1_var, cons2_var), f)(s)))
  
```

Repeatedly Skolemizing and flattening,

Instantiating quantified variables,

Expanding the definition of `every`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 2 subgoals:

C Proof scripts

`apply_side_effects_ok.2.1:`

{-1}	<code>is_linear_plain_memory?(pm')</code>
{-2}	<code>pm' 'states(s')</code>
{-3}	<code>every(λ (e: [Memory_Address_4G, list[Byte]]): transformer_invariant?(pm' 'states, singleton(expr_2_super(f'(e)))) (cons2_var'))</code>
{-4}	<code>OK?(apply_side_effects(cons2_var', f')(s'))</code>
{-5}	<code>∀ (s1: (pm' 'states)): every(λ (e: [Memory_Address_4G, list[Byte]]): OK?(f'(e)(s1)) (cons(cons1_var', cons2_var'))</code>
{-6}	<code>transformer_invariant?(pm' 'states, singleton(expr_2_super(f'(cons1_var'))))</code>
{1}	<code>OK?(apply_side_effects(cons(cons1_var', cons2_var'), f')(s'))</code>

Using lemma `apply_side_effects_pm_states`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Using lemma `expr_transformer_invariant_next_ok`,

Installing automatic rewrites from: (`has_next_state!` `singleton!`)

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Expanding the definition of `apply_side_effects`,

Expanding the definition of `reduce`,

Expanding the definition of `##`,

Simplifying, rewriting, and recording with decision procedures,

Instantiating the top quantifier in -7 with the terms: (`state(reduce(ok_result(null), LAMBDA (e: [Memory_Address_4G, list[Byte]], r: [Linear_memory[Physical_memory, pm_phy] -> ExprResult [Linear_memory[Physical_memory, pm_phy], list[Byte]]]): (r ## (LAMBDA (tail: list[Byte]): (f!1(e) ## (LAMBDA (head: list[Byte]): ok_result(append(head, tail)))))) (cons2_var!1)(s!1))),`

Expanding the definition of `every`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Rewriting using `ok_result_ok`, matching in `*`,

This completes the proof of `apply_side_effects_ok.2.1`.

`apply_side_effects_ok.2.2:`

{-1}	<code>is_linear_plain_memory?(pm')</code>
{-2}	<code>pm' 'states(s')</code>
{-3}	<code>every(λ (e: [Memory_Address_4G, list[Byte]]): transformer_invariant?(pm' 'states, singleton(expr_2_super(f'(e)))) (cons2_var'))</code>
{-4}	<code>∀ (s1: (pm' 'states)): every(λ (e: [Memory_Address_4G, list[Byte]]): OK?(f'(e)(s1)) (cons(cons1_var', cons2_var'))</code>
{-5}	<code>transformer_invariant?(pm' 'states, singleton(expr_2_super(f'(cons1_var'))))</code>
{1}	<code>∀ (s1: (pm' 'states)): every(λ (e: [Memory_Address_4G, list[Byte]]): OK?(f'(e)(s1))(cons2_var'))</code>
{2}	<code>OK?(apply_side_effects(cons(cons1_var', cons2_var'), f')(s'))</code>

Repeatedly Skolemizing and flattening,

Instantiating quantified variables,

Expanding the definition of `every`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `apply_side_effects_ok.2.2`.

Q.E.D.

C.117.25**Linear_Memory_Properties.apply_side_effects_same_result_TCC1**

Terse proof for apply_side_effects_same_result_TCC1.

apply_side_effects_same_result_TCC1:

$$\begin{array}{l}
\{1\} \quad \forall (\text{pm}: \text{Plain_Memory}[\text{Linear_memory}], s: \text{Linear_memory}[\text{Physical_memory}, \text{pm_phy}], \\
\quad l: \text{list}[[\text{Memory_Address_4G}, \text{list}[\text{Byte}]]], \\
\quad f: \\
\quad \quad [\text{Memory_Address_4G}, \text{list}[\text{Byte}] \rightarrow \\
\quad \quad \quad [\text{Linear_memory}[\text{Physical_memory}, \text{pm_phy}] \rightarrow \\
\quad \quad \quad \quad \text{ExprResult}[\text{Linear_memory}[\text{Physical_memory}, \text{pm_phy}], \text{list}[\text{Byte}]]]): \\
\text{is_linear_plain_memory?}(\text{pm}) \wedge \\
\text{pm}'\text{states}(s) \wedge \\
(\forall (s_1: (\text{pm}'\text{states})): \\
\quad \text{every}(\lambda (e: [\text{Memory_Address_4G}, \text{list}[\text{Byte}]]): \text{OK?}(f(e)(s_1))(l)) \\
\quad \wedge \\
\quad (\forall (s_1: (\text{pm}'\text{states})): \\
\quad \quad \text{every}(\lambda (e: [\text{Memory_Address_4G}, \text{list}[\text{Byte}]]): \\
\quad \quad \quad \text{OK?}(f(e)(s_1)) \supset \text{data}(f(e)(s_1)) = e'2) \\
\quad \quad \quad (l)) \\
\quad \wedge \\
\quad \text{every}(\lambda (e: [\text{Memory_Address_4G}, \text{list}[\text{Byte}]]): \\
\quad \quad \text{transformer_invariant?}(\text{pm}'\text{states}, \text{singleton}(\text{expr_2_super}(f(e)))) \\
\quad \quad (l)) \\
\quad \supset \\
\text{OK?}[\text{Physical_memory}, \text{list}[\text{Byte}]] \\
\quad (\text{apply_side_effects}[\text{Physical_memory}, \text{pm_phy}](l, f)(s))
\end{array}$$

Repeatedly Skolemizing and flattening,

Using lemma apply_side_effects_ok,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of apply_side_effects_same_result_TCC1.

Q.E.D.

C.117.26 Linear_Memory_Properties.apply_side_effects_same_result

Terse proof for apply_side_effects_same_result.

apply_side_effects_same_result:

$$\begin{array}{l}
 \{1\} \quad \forall (pm: \text{Plain_Memory}[\text{Linear_memory}], s: \text{Linear_memory}[\text{Physical_memory}, pm_phy], \\
 \quad l: \text{list}[[\text{Memory_Address_4G}, \text{list}[\text{Byte}]]], \\
 \quad f: \\
 \quad \quad [\text{Memory_Address_4G}, \text{list}[\text{Byte}] \rightarrow \\
 \quad \quad \quad [\text{Linear_memory} \rightarrow \text{ExprResult}[\text{Linear_memory}, \text{list}[\text{Byte}]]]): \\
 \text{is_linear_plain_memory?}(pm) \wedge \\
 pm' \text{states}(s) \wedge \\
 (\forall (s_1: (pm' \text{states})): \\
 \quad \text{every}(\lambda (e: [\text{Memory_Address_4G}, \text{list}[\text{Byte}]]): \text{OK?}(f(e)(s_1)))(l)) \\
 \wedge \\
 (\forall (s_1: (pm' \text{states})): \\
 \quad \text{every}(\lambda (e: [\text{Memory_Address_4G}, \text{list}[\text{Byte}]]): \\
 \quad \quad \text{OK?}(f(e)(s_1)) \supset \text{data}(f(e)(s_1)) = e'2) \\
 \quad \quad (l)) \\
 \wedge \\
 \text{every}(\lambda (e: [\text{Memory_Address_4G}, \text{list}[\text{Byte}]]): \\
 \quad \text{transformer_invariant?}(pm' \text{states}, \text{singleton}(\text{expr_2_super}(f(e)))) \\
 \quad (l)) \\
 \supset \\
 \text{data}(\text{apply_side_effects}(l, f)(s)) = \\
 \text{reduce}(\text{null}, \\
 \quad \lambda (e: [\text{Address}, \text{list}[\text{Byte}], \text{tail}: \text{list}[\text{Byte}]]): \\
 \quad \quad \text{LET head} = e'2 \text{ IN append}(\text{head}, \text{tail})) \\
 \quad (l)
 \end{array}$$

Inducting on l on formula 1,

we get 4 subgoals:

apply_side_effects_same_result.1:

{-1}	$ \begin{aligned} & \forall (\text{pm}: \text{Plain_Memory}[\text{Linear_memory}], s: \text{Linear_memory}[\text{Physical_memory}, \text{pm_phy}], \\ & \quad f: \\ & \quad \quad [\text{Memory_Address_4G}, \text{list}[\text{Byte}] \rightarrow \\ & \quad \quad \quad [\text{Linear_memory} \rightarrow \text{ExprResult}[\text{Linear_memory}, \text{list}[\text{Byte}]]]): \\ & \text{is_linear_plain_memory?}(\text{pm}) \wedge \\ & \text{pm}'\text{states}(s) \wedge \\ & (\forall (s_1: (\text{pm}'\text{states})): \\ & \quad \text{every}(\lambda (e: [\text{Memory_Address_4G}, \text{list}[\text{Byte}]]) : \text{OK?}(f(e)(s_1)))(l')) \\ & \quad \wedge \\ & (\forall (s_1: (\text{pm}'\text{states})): \\ & \quad \text{every}(\lambda (e: [\text{Memory_Address_4G}, \text{list}[\text{Byte}]]) : \\ & \quad \quad \text{OK?}(f(e)(s_1)) \supset \text{data}(f(e)(s_1)) = e'2) \\ & \quad \quad (l')) \\ & \quad \wedge \\ & \text{every}(\lambda (e: [\text{Memory_Address_4G}, \text{list}[\text{Byte}]]) : \\ & \quad \text{transformer_invariant?}(\text{pm}'\text{states}, \text{singleton}(\text{expr_2_super}(f(e)))) \\ & \quad (l')) \\ & \supset \\ & \text{data}(\text{apply_side_effects}(l', f)(s)) = \\ & \quad \text{reduce}(\text{null}, \lambda (e: [\text{Address}, \text{list}[\text{Byte}]], \text{tail}: \text{list}[\text{Byte}]) : \text{append}(e'2, \text{tail})) \\ & \quad (l') \end{aligned} $
{1}	$ \begin{aligned} & \forall (\text{pm}: \text{Plain_Memory}[\text{Linear_memory}], s: \text{Linear_memory}[\text{Physical_memory}, \text{pm_phy}], \\ & \quad f: \\ & \quad \quad [\text{Memory_Address_4G}, \text{list}[\text{Byte}] \rightarrow \\ & \quad \quad \quad [\text{Linear_memory} \rightarrow \text{ExprResult}[\text{Linear_memory}, \text{list}[\text{Byte}]]]): \\ & \text{is_linear_plain_memory?}(\text{pm}) \wedge \\ & \text{pm}'\text{states}(s) \wedge \\ & (\forall (s_1: (\text{pm}'\text{states})): \\ & \quad \text{every}(\lambda (e: [\text{Memory_Address_4G}, \text{list}[\text{Byte}])]) : \text{OK?}(f(e)(s_1)))(l')) \\ & \quad \wedge \\ & (\forall (s_1: (\text{pm}'\text{states})): \\ & \quad \text{every}(\lambda (e: [\text{Memory_Address_4G}, \text{list}[\text{Byte}])]) : \\ & \quad \quad \text{OK?}(f(e)(s_1)) \supset \text{data}(f(e)(s_1)) = e'2) \\ & \quad \quad (l')) \\ & \quad \wedge \\ & \text{every}(\lambda (e: [\text{Memory_Address_4G}, \text{list}[\text{Byte}])]) : \\ & \quad \text{transformer_invariant?}(\text{pm}'\text{states}, \text{singleton}(\text{expr_2_super}(f(e)))) \\ & \quad (l')) \\ & \supset \\ & \text{data}(\text{apply_side_effects}(l', f)(s)) = \\ & \quad \text{reduce}(\text{null}, \\ & \quad \quad \lambda (e: [\text{Address}, \text{list}[\text{Byte}]], \text{tail}: \text{list}[\text{Byte}]) : \\ & \quad \quad \quad \text{LET head} = e'2 \text{ IN } \text{append}(\text{head}, \text{tail})) \\ & \quad (l') \end{aligned} $

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of apply_side_effects_same_result.1.

apply_side_effects_same_result.2:

<pre> {1} ∃ (pm: Plain_Memory[Linear_memory], s: Linear_memory[Physical_memory, pm_phy], f: [Memory_Address_4G, list[Byte] → [Linear_memory → ExprResult[Linear_memory, list[Byte]]]): is_linear_plain_memory?(pm) ∧ pm'states(s) ∧ (∃ (s₁: (pm'states)): every(λ (e: [Memory_Address_4G, list[Byte]]): OK?(f(e)(s₁))(null)) ∧ (∃ (s₁: (pm'states)): every(λ (e: [Memory_Address_4G, list[Byte]]): OK?(f(e)(s₁)) ⊃ data(f(e)(s₁)) = e'2 (null)) ∧ every(λ (e: [Memory_Address_4G, list[Byte]]): transformer_invariant?(pm'states, singleton(expr_2_super(f(e)))) (null)) ⊃ data(apply_side_effects(null, f)(s)) = reduce(null, λ (e: [Address, list[Byte]], tail: list[Byte]): append(e'2, tail)) (null) </pre>

Repeatedly Skolemizing and flattening,

Expanding the definition of apply_side_effects,

Expanding the definition of reduce,

Expanding the definition of ok_result,

which is trivially true.

This completes the proof of apply_side_effects_same_result.2.

apply_side_effects_same_result.3:

```

{1}  ∀ (cons1_var: [Memory_Address_4G, list[Byte]],
      cons2_var: list[[Memory_Address_4G, list[Byte]]]):
  (∀ (pm: Plain_Memory[Linear_memory], s: Linear_memory[Physical_memory, pm_phy],
      f:
        [Memory_Address_4G, list[Byte] →
         [Linear_memory → ExprResult[Linear_memory, list[Byte]]]]):
    is_linear_plain_memory?(pm) ∧
    pm'states(s) ∧
    (∀ (s1: (pm'states)):
      every(λ (e: [Memory_Address_4G, list[Byte]]): OK?(f(e)(s1)))
        (cons2_var))
    ∧
    (∀ (s1: (pm'states)):
      every(λ (e: [Memory_Address_4G, list[Byte]]):
        OK?(f(e)(s1)) ⊃ data(f(e)(s1)) = e'2)
        (cons2_var))
    ∧
    every(λ (e: [Memory_Address_4G, list[Byte]]):
      transformer_invariant?(pm'states, singleton(expr_2_super(f(e))))
        (cons2_var))
    ⊃
    data(apply_side_effects(cons2_var, f)(s)) =
      reduce(null,
        λ (e: [Address, list[Byte]], tail: list[Byte]): append(e'2, tail))
        (cons2_var))
    ⊃
    (∀ (pm: Plain_Memory[Linear_memory], s: Linear_memory[Physical_memory, pm_phy],
      f:
        [Memory_Address_4G, list[Byte] →
         [Linear_memory → ExprResult[Linear_memory, list[Byte]]]]):
      is_linear_plain_memory?(pm) ∧
      pm'states(s) ∧
      (∀ (s1: (pm'states)):
        every(λ (e: [Memory_Address_4G, list[Byte]]): OK?(f(e)(s1)))
          (cons(cons1_var, cons2_var)))
      ∧
      (∀ (s1: (pm'states)):
        every(λ (e: [Memory_Address_4G, list[Byte]]):
          OK?(f(e)(s1)) ⊃ data(f(e)(s1)) = e'2)
          (cons(cons1_var, cons2_var)))
      ∧
      every(λ (e: [Memory_Address_4G, list[Byte]]):
        transformer_invariant?(pm'states, single-
ton(expr_2_super(f(e))))
          (cons(cons1_var, cons2_var)))
      ⊃
      data(apply_side_effects(cons(cons1_var, cons2_var), f)(s)) =
        reduce(null,
          λ (e: [Address, list[Byte]], tail: list[Byte]): append(e'2, tail))
          (cons(cons1_var, cons2_var)))
  )

```

C Proof scripts

Repeatedly Skolemizing and flattening,

Instantiating quantified variables,

Expanding the definition of every,

Case splitting on FORALL (s_1 : (pm!1'states)): every(LAMBDA (e: [Memory_Address_4G, list[Byte]]): OK?(f!1(e)(s1))) (cons2_var!1),

we get 2 subgoals:

`apply_side_effects_same_result.3.1:`

<div style="display: flex; flex-direction: column; gap: 5px;"> <div style="display: flex; align-items: flex-start;"> {-1} $\forall (s_1: (\text{pm}'\text{'states})): \text{every}(\lambda (e: [\text{Memory_Address_4G}, \text{list}[\text{Byte}]]): \text{OK?}(f'(e)(s_1)))(\text{cons2_var}'$ </div> <div style="display: flex; align-items: flex-start;"> {-2} $\text{is_linear_plain_memory?}(\text{pm}') \wedge$ </div> <div style="display: flex; align-items: flex-start;"> $\text{pm}'\text{'states}(s') \wedge$ </div> <div style="display: flex; align-items: flex-start;"> $(\forall (s_1: (\text{pm}'\text{'states})): \text{every}(\lambda (e: [\text{Memory_Address_4G}, \text{list}[\text{Byte}]]): \text{OK?}(f'(e)(s_1))$ </div> <div style="display: flex; align-items: flex-start;"> $(\text{cons2_var}'))$ </div> <div style="display: flex; align-items: center;"> \wedge </div> <div style="display: flex; align-items: flex-start;"> $(\forall (s_1: (\text{pm}'\text{'states})): \text{every}(\lambda (e: [\text{Memory_Address_4G}, \text{list}[\text{Byte}]]):$ </div> <div style="display: flex; align-items: flex-start;"> $\text{OK?}(f'(e)(s_1)) \supset \text{data}(f'(e)(s_1)) = e'2$ </div> <div style="display: flex; align-items: flex-start;"> $(\text{cons2_var}'))$ </div> <div style="display: flex; align-items: center;"> \wedge </div> <div style="display: flex; align-items: flex-start;"> $\text{every}(\lambda (e: [\text{Memory_Address_4G}, \text{list}[\text{Byte}]]):$ </div> <div style="display: flex; align-items: flex-start;"> $\text{transformer_invariant?}(\text{pm}'\text{'states}, \text{singleton}(\text{expr_2_super}(f'(e))))$ </div> <div style="display: flex; align-items: flex-start;"> $(\text{cons2_var}')$ </div> <div style="display: flex; align-items: center;"> \supset </div> <div style="display: flex; align-items: flex-start;"> $\text{data}(\text{apply_side_effects}(\text{cons2_var}', f')(s')) =$ </div> <div style="display: flex; align-items: flex-start;"> $\text{reduce}(\text{null}, \lambda (e: [\text{Address}, \text{list}[\text{Byte}]], \text{tail}: \text{list}[\text{Byte}]): \text{append}(e'2, \text{tail}))$ </div> <div style="display: flex; align-items: flex-start;"> $(\text{cons2_var}')$ </div> <div style="display: flex; align-items: flex-start;"> {-3} $\text{is_linear_plain_memory?}(\text{pm}')$ </div> <div style="display: flex; align-items: flex-start;"> {-4} $\text{pm}'\text{'states}(s')$ </div> <div style="display: flex; align-items: flex-start;"> {-5} $\forall (s_1: (\text{pm}'\text{'states})): \text{every}(\lambda (e: [\text{Memory_Address_4G}, \text{list}[\text{Byte}]]): \text{OK?}(f'(e)(s_1))$ </div> <div style="display: flex; align-items: flex-start;"> $(\text{cons}(\text{cons1_var}', \text{cons2_var}'))$ </div> <div style="display: flex; align-items: flex-start;"> {-6} $\forall (s_1: (\text{pm}'\text{'states})): \text{every}(\lambda (e: [\text{Memory_Address_4G}, \text{list}[\text{Byte}]]):$ </div> <div style="display: flex; align-items: flex-start;"> $\text{OK?}(f'(e)(s_1)) \supset \text{data}(f'(e)(s_1)) = e'2$ </div> <div style="display: flex; align-items: flex-start;"> $(\text{cons}(\text{cons1_var}', \text{cons2_var}'))$ </div> <div style="display: flex; align-items: flex-start;"> {-7} $\text{transformer_invariant?}(\text{pm}'\text{'states}, \text{singleton}(\text{expr_2_super}(f'(\text{cons1_var}')))) \wedge$ </div> <div style="display: flex; align-items: flex-start;"> $\text{every}(\lambda (e: [\text{Memory_Address_4G}, \text{list}[\text{Byte}]]):$ </div> <div style="display: flex; align-items: flex-start;"> $\text{transformer_invariant?}(\text{pm}'\text{'states}, \text{singleton}(\text{expr_2_super}(f'(e))))$ </div> <div style="display: flex; align-items: flex-start;"> $(\text{cons2_var}')$ </div> <hr style="border: 0.5px solid black; margin: 5px 0;"/> <div style="display: flex; align-items: flex-start;"> {1} $\text{data}(\text{apply_side_effects}(\text{cons}(\text{cons1_var}', \text{cons2_var}'), f')(s')) =$ </div> <div style="display: flex; align-items: flex-start;"> $\text{reduce}(\text{null}, \lambda (e: [\text{Address}, \text{list}[\text{Byte}]], \text{tail}: \text{list}[\text{Byte}]): \text{append}(e'2, \text{tail}))$ </div> <div style="display: flex; align-items: flex-start;"> $(\text{cons}(\text{cons1_var}', \text{cons2_var}'))$ </div> </div>	
---	--

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 2 subgoals:

apply_side_effects_same_result.3.1.1:

{-1}	$\forall (s_1: (\text{pm}'\text{'states})): \text{every}(\lambda (e: [\text{Memory_Address_4G}, \text{list}[\text{Byte}]]): \text{OK?}(f'(e)(s_1)))(\text{cons2_var}')$
{-2}	$\text{is_linear_plain_memory?}(\text{pm}')$
{-3}	$\text{pm}'\text{'states}(s')$
{-4}	$\text{every}(\lambda (e: [\text{Memory_Address_4G}, \text{list}[\text{Byte}]]): \text{transformer_invariant?}(\text{pm}'\text{'states}, \text{singleton}(\text{expr_2_super}(f'(e))))(\text{cons2_var}'))$
{-5}	$\text{data}(\text{apply_side_effects}(\text{cons2_var}', f')(s')) = \text{reduce}(\text{null}, \lambda (e: [\text{Address}, \text{list}[\text{Byte}]], \text{tail}: \text{list}[\text{Byte}]): \text{append}(e'2, \text{tail}))(\text{cons2_var}')$
{-6}	$\forall (s_1: (\text{pm}'\text{'states})): \text{every}(\lambda (e: [\text{Memory_Address_4G}, \text{list}[\text{Byte}]]): \text{OK?}(f'(e)(s_1)))(\text{cons}(\text{cons1_var}', \text{cons2_var}'))$
{-7}	$\forall (s_1: (\text{pm}'\text{'states})): \text{every}(\lambda (e: [\text{Memory_Address_4G}, \text{list}[\text{Byte}]]): \text{OK?}(f'(e)(s_1)) \supset \text{data}(f'(e)(s_1)) = e'2)(\text{cons}(\text{cons1_var}', \text{cons2_var}'))$
{-8}	$\text{transformer_invariant?}(\text{pm}'\text{'states}, \text{singleton}(\text{expr_2_super}(f'(\text{cons1_var}'))))$
{1}	$\text{data}(\text{apply_side_effects}(\text{cons}(\text{cons1_var}', \text{cons2_var}'), f')(s')) = \text{reduce}(\text{null}, \lambda (e: [\text{Address}, \text{list}[\text{Byte}]], \text{tail}: \text{list}[\text{Byte}]): \text{append}(e'2, \text{tail}))(\text{cons}(\text{cons1_var}', \text{cons2_var}'))$

Using lemma apply_side_effects_ok,

Using lemma apply_side_effects_pm_states,

Expanding the definition of apply_side_effects,

Expanding the definition of reduce,

Expanding the definition of ##,

Expanding the definition of ##,

Simplifying, rewriting, and recording with decision procedures,

Expanding the definition of every,

Rewriting using forall_and, matching in *,

Rewriting using forall_and, matching in *,

Applying disjunctive simplification to flatten sequent,

Replacing using formula -9,

Expanding the definition of apply_side_effects,

Simplifying, rewriting, and recording with decision procedures,

Case splitting on pm!1'states (state(reduce(ok_result(null), LAMBDA (e: [Memory_Address_4G, list[Byte]], r: [Linear_memory[Physical_memory, pm_phy] -> ExprResult [Linear_memory [Physical_memory, pm_phy], list[Byte]]]: (r ## (LAMBDA (tail: list[Byte]): (f!1(e) ## (LAMBDA (head: list[Byte]): ok_result(append(head, tail)))))) (cons2_var!1)(s!1))),

we get 2 subgoals:

C.117 Proofs for Linear_Memory_Properties (challenge-linear.pvs)

Instantiating the top quantifier in -8 with the terms: (state(reduce(ok_result(null), LAMBDA (e: [Memory_Address_4G, list[Byte]], r: [Linear_memory[Physical_memory, pm_phy] -> ExprResult [Linear_memory[Physical_memory, pm_phy], list[Byte]]]): (r ## (LAMBDA (tail: list[Byte]): (f!1(e) ## (LAMBDA (head: list[Byte]): ok_result(append(head, tail))))))) (cons2_var!1)(s!1))),

we get 2 subgoals:

apply_side_effects_same_result.3.1.1.1.1:

```

{-1} pm' `states
      (state(reduce(ok_result(null),
                    λ (e: [Memory_Address_4G, list[Byte]],
                      r:
                        [Linear_memory[Physical_memory, pm_phy] →
                          ExprResult
                            [Linear_memory[Physical_memory, pm_phy], list[Byte]]
                        (r ##
                          (λ (tail: list[Byte]):
                            (f'(e) ##
                              (λ (head: list[Byte]):
                                ok_result(append(head, tail)))))))
                    (cons2_var')(s')))
{-2} transformer_invariant?(pm' `states,
                              singleton(expr_2_super(λ (s:
                                                        Lin-
ear_memory[Physical_memory, pm_phy]):
                                                    reduce(ok_result(null),
                                                          λ (e:
                                                            [Memory_Address_4G, list[Byte]],
                                                              r:
                                                                [Linear_memory
                                                                  [Physical_memory
                                                                    ExprResult
                                                                      [Linear_memory
                                                                        [Physical_memory
                                                                          list[Byte]]]):
                                                                (r
                                                                  ##
                                                                  (λ
                                                                    (tail: list[Byte]):
                                                                      (f'(e)
                                                                        ##
                                                                        (λ
                                                                          (head: list[Byte]
                                                                            ok_result
                                                                              (append
                                                                                (head, tail))
                                                                              (cons2_var')(s)))))))
{-3} OK?(reduce(ok_result(null),
                 λ (e: [Memory_Address_4G, list[Byte]],
                   r:
                     [Linear_memory[Physical_memory, pm_phy] →
                       ExprResult [Linear_memory[Physical_memory, pm_phy], list[Byte]]
                     (r ##
                       (λ (tail: list[Byte]):
                         (f'(e) ##
                           (λ (head: list[Byte]): ok_result(append(head, tail)))))))
                 (cons2_var')(s')))
{-4} is_linear_plain_memory?(pm')
{-5} pm' `states(s')
{-6} every(λ (e: [Memory_Address_4G, list[Byte]]):
           transformer_invariant?(pm' `states, singleton(expr_2_super(f'(e))))
           (cons2_var'))
1598 {-7} data(reduce(ok_result(null),
                    λ (e: [Memory_Address_4G, list[Byte]],
                      r:
                        [Linear_memory[Physical_memory, pm_phy] →
                          ExprResult [Linear_memory[Physical_memory, pm_phy], list[Byte]]
                        (r ##
                          (λ (tail: list[Byte]):
                            (f'(e) ##
                              (λ (head: list[Byte]): ok_result(append(head, tail)))))))

```

C.117 Proofs for Linear_Memory_Properties (challenge-linear.pvs)

Instantiating the top quantifier in -10 with the terms: (state(reduce(ok_result(null), LAMBDA (e: [Memory_Address_4G, list[Byte]], r: [Linear_memory[Physical_memory, pm_phy] -> ExprResult [Linear_memory[Physical_memory, pm_phy], list[Byte]]]): (r ## (LAMBDA (tail: list[Byte]): (f!1(e) ## (LAMBDA (head: list[Byte]): ok_result(append(head, tail))))))) (cons2_var!1)(s!1))),

we get 2 subgoals:


```
apply_side_effects_same_result.3.1.1.1.1.1:
```

```
{-1} pm' `states
      (state(reduce(ok_result(null),
                    λ (e: [Memory_Address_4G, list[Byte]],
                      r:
                        [Linear_memory[Physical_memory, pm_phy] →
                          ExprResult
                            [Linear_memory[Physical_memory, pm_phy], list[Byte]]
                        (r ##
                          (λ (tail: list[Byte]):
                            (f'(e) ##
                              (λ (head: list[Byte]):
                                ok_result(append(head, tail)))))))
                    (cons2_var')(s')))
{-2} transformer_invariant?(pm' `states,
                              singleton(expr_2_super(λ (s:
                                                        Lin-
                                                        ear_memory[Physical_memory, pm_phy]):
                                                                reduce(ok_result(null),
                                                                    λ (e:
                                                                      [Memory_Address_4G, list[Byte]],
                                                                        r:
                                                                          [Linear_memory
                                                                            [Physical_memory
                                                                              ExprResult
                                                                                [Linear_memory
                                                                                  [Physical_memory
                                                                                    list[Byte]]]):
                                                                            (r
                                                                              ##
                                                                              (λ
                                                                                (tail: list[Byte]):
                                                                                  (f'(e)
                                                                                    ##
                                                                                    (λ
                                                                                      (head: list[Byte]
                                                                                        ok_result
                                                                                          (append
                                                                                            (head, tail))
                                                                                          (cons2_var')(s)))))))
{-3} OK?(reduce(ok_result(null),
                λ (e: [Memory_Address_4G, list[Byte]],
                  r:
                    [Linear_memory[Physical_memory, pm_phy] →
                      ExprResult [Linear_memory[Physical_memory, pm_phy], list[Byte]]
                    (r ##
                      (λ (tail: list[Byte]):
                        (f'(e) ##
                          (λ (head: list[Byte]): ok_result(append(head, tail)))))))
                (cons2_var')(s')))
{-4} is_linear_plain_memory?(pm')
{-5} pm' `states(s')
{-6} every(λ (e: [Memory_Address_4G, list[Byte]]):
           transformer_invariant?(pm' `states, singleton(expr_2_super(f'(e))))
           (cons2_var'))
1600 {-7} data(reduce(ok_result(null),
                    λ (e: [Memory_Address_4G, list[Byte]],
                      r:
                        [Linear_memory[Physical_memory, pm_phy] →
                          ExprResult [Linear_memory[Physical_memory, pm_phy], list[Byte]]
                        (r ##
                          (λ (tail: list[Byte]):
                            (f'(e) ##
                              (λ (head: list[Byte]): ok_result(append(head, tail)))))))
```

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Rewriting using `ok_result_data`, matching in `*`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `apply_side_effects_same_result.3.1.1.1.1.1`.

apply_side_effects_same_result.3.1.1.1.1.2:

```

{-1}  pm' `states
      (state(reduce(ok_result(null),
                    λ (e: [Memory_Address_4G, list[Byte]],
                      r:
                        [Linear_memory[Physical_memory, pm_phy] →
                          ExprResult
                            [Linear_memory[Physical_memory, pm_phy], list[Byte]]
                        (r ##
                          (λ (tail: list[Byte]):
                            (f'(e) ##
                              (λ (head: list[Byte]):
                                ok_result(append(head, tail)))))))
                        (cons2_var')(s')))
{-2}  transformer_invariant?(pm' `states,
      singleton(expr_2_super(λ (s:
                                Lin-
                                ear_memory[Physical_memory, pm_phy]):
                                reduce(ok_result(null),
                                    λ (e:
                                        [Memory_Address_4G, list[Byte]],
                                        r:
                                          [Linear_memory
                                            [Physical_memory
                                              ExprResult
                                                [Linear_memory
                                                  [Physical_memory
                                                    list[Byte]]]):
                                          (r
                                            ##
                                            (λ
                                              (tail: list[Byte]):
                                              (f'(e)
                                                ##
                                                (λ
                                                  (head: list[Byte]
                                                    ok_result
                                                      (append
                                                        (head, tail))
                                                    (cons2_var')(s))))))
{-3}  OK?(reduce(ok_result(null),
                λ (e: [Memory_Address_4G, list[Byte]],
                  r:
                    [Linear_memory[Physical_memory, pm_phy] →
                      ExprResult [Linear_memory[Physical_memory, pm_phy], list[Byte]]
                    (r ##
                      (λ (tail: list[Byte]):
                        (f'(e) ##
                          (λ (head: list[Byte]): ok_result(append(head, tail)))))))
                    (cons2_var')(s')))
{-4}  is_linear_plain_memory?(pm')
{-5}  pm' `states(s')
{-6}  every(λ (e: [Memory_Address_4G, list[Byte]]):
        transformer_invariant?(pm' `states, singleton(expr_2_super(f'(e))))
        (cons2_var'))
1602 {-7}  data(reduce(ok_result(null),
                    λ (e: [Memory_Address_4G, list[Byte]],
                      r:
                        [Linear_memory[Physical_memory, pm_phy] →
                          ExprResult [Linear_memory[Physical_memory, pm_phy], list[Byte]]
                        (r ##
                          (λ (tail: list[Byte]):
                            (f'(e) ##
                              (λ (head: list[Byte]): ok_result(append(head, tail)))))))

```

which is trivially true.

This completes the proof of `apply_side_effects_same_result.3.1.1.1.1.2`.

apply_side_effects_same_result.3.1.1.1.2:

```

{-1}  pm' `states
      (state(reduce(ok_result(null),
                    λ (e: [Memory_Address_4G, list[Byte]],
                      r:
                        [Linear_memory[Physical_memory, pm_phy] →
                          ExprResult
                            [Linear_memory[Physical_memory, pm_phy], list[Byte]]
                        (r ##
                          (λ (tail: list[Byte]):
                            (f'(e) ##
                              (λ (head: list[Byte]):
                                ok_result(append(head, tail)))))))
                    (cons2_var')(s')))
{-2}  transformer_invariant?(pm' `states,
      singleton(expr_2_super(λ (s:
        Lin-
        ear_memory[Physical_memory, pm_phy]):
          reduce(ok_result(null),
                λ (e:
                  [Memory_Address_4G, list[Byte]],
                  r:
                    [Linear_memory
                      [Physical_memory
                        ExprResult
                          [Linear_memory
                            [Physical_memory
                              list[Byte]]]):
                    (r
                      ##
                      (λ
                        (tail: list[Byte]):
                          (f'(e)
                            ##
                            (λ
                              (head: list[Byte]
                                ok_result
                                  (append
                                    (head, tail))
                                  (cons2_var')(s)))))))
{-3}  OK?(reduce(ok_result(null),
                λ (e: [Memory_Address_4G, list[Byte]],
                  r:
                    [Linear_memory[Physical_memory, pm_phy] →
                      ExprResult [Linear_memory[Physical_memory, pm_phy], list[Byte]]
                    (r ##
                      (λ (tail: list[Byte]):
                        (f'(e) ##
                          (λ (head: list[Byte]): ok_result(append(head, tail)))))))
        (cons2_var')(s')))
{-4}  is_linear_plain_memory?(pm')
{-5}  pm' `states(s')
{-6}  every(λ (e: [Memory_Address_4G, list[Byte]]):
        transformer_invariant?(pm' `states, singleton(expr_2_super(f'(e))))
        (cons2_var'))
1604 {-7}  data(reduce(ok_result(null),
                λ (e: [Memory_Address_4G, list[Byte]],
                  r:
                    [Linear_memory[Physical_memory, pm_phy] →
                      ExprResult [Linear_memory[Physical_memory, pm_phy], list[Byte]]
                    (r ##
                      (λ (tail: list[Byte]):
                        (f'(e) ##
                          (λ (head: list[Byte]): ok_result(append(head, tail)))))))

```

which is trivially true.

This completes the proof of `apply_side_effects_same_result.3.1.1.1.2`.

apply_side_effects_same_result.3.1.1.2:

```

{-1}  transformer_invariant?(pm' 'states,
      singleton(expr_2_super(λ (s:
                              Lin-
                              reduce(ok_result(null),
                                      λ (e:
                                          [Memory_Address_4G, list[Byte]],
                                          r:
                                          [Linear_memory
                                           [Physical_memory
                                            ExprResult
                                             [Linear_memory
                                              [Physical_memory
                                               list[Byte]]]):
                                          (r
                                           ##
                                           (λ
                                            (tail: list[Byte]):
                                            (f'(e)
                                             ##
                                             (λ
                                              (head: list[Byte]
                                               ok_result
                                               (append
                                                (head, tail)))
                                              (cons2_var')(s))))))
      OK?(reduce(ok_result(null),
                λ (e: [Memory_Address_4G, list[Byte]],
                    r:
                    [Linear_memory [Physical_memory, pm_phy] →
                     ExprResult [Linear_memory [Physical_memory, pm_phy], list[Byte]]
                    (r ##
                     (λ (tail: list[Byte]):
                       (f'(e) ##
                        (λ (head: list[Byte]): ok_result(append(head, tail))))))
                    (cons2_var')(s'))
      {-3}  is_linear_plain_memory?(pm')
      {-4}  pm' 'states(s')
      {-5}  every(λ (e: [Memory_Address_4G, list[Byte]]):
                transformer_invariant?(pm' 'states, singleton(expr_2_super(f'(e))))
                (cons2_var'))
      {-6}  data(reduce(ok_result(null),
                      λ (e: [Memory_Address_4G, list[Byte]],
                          r:
                          [Linear_memory [Physical_memory, pm_phy] →
                           ExprResult [Linear_memory [Physical_memory, pm_phy], list[Byte]]
                          (r ##
                           (λ (tail: list[Byte]):
                             (f'(e) ##
                              (λ (head: list[Byte]): ok_result(append(head, tail))))))
                          (cons2_var')(s'))
          =
          reduce(null, λ (e: [Address, list[Byte]], tail: list[Byte]): append(e'2, tail))
          (cons2_var'))
      {-7}  ∀ (x: (pm' 'states)): OK?(f'(cons1_var')(x))
      {-8}  ∀ (x: (pm' 'states)):
            every(λ (e: [Memory_Address_4G, list[Byte]]): OK?(f'(e)(x)))(cons2_var')
      {-9}  ∀ (x: (pm' 'states)):
            OK?(f'(cons1_var')(x)) ⊃ data(f'(cons1_var')(x)) = cons1_var' '2
      {-10} ∀ (x: (pm' 'states)):
            every(λ (e: [Memory_Address_4G, list[Byte]]):
                OK?(f'(e)(x)) ⊃ data(f'(e)(x)) = e'2)

```

Hiding formulas: 2,

Using lemma `expr_transformer_invariant_next_ok`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 2 subgoals:

apply_side_effects_same_result.3.1.1.2.1:

```

{-1}  transformer_invariant?(pm' 'states,
      singleton(expr_2_super(λ (s:
                                Lin-
                                reduce(ok_result(null),
                                      λ (e:
                                          [Memory_Address_4G,
                                          list[Byte]],
                                          r:
                                          [Linear_memory
                                          [Physical_memory
                                          ExprResult
                                          [Linear_memory
                                          [Physical_memory
                                          list[Byte]]]):
                                          (r
                                          ##
                                          (λ
                                          (tail: list[Byte]):
                                          (f'(e)
                                          ##
                                          (λ
                                          (head: list[Byte],
                                          ok_result
                                          (append
                                          (head, tail)))
                                          (cons2_var')(s))))))
      pm' 'states(s')
{-2}  OK?(reduce(ok_result(null),
{-3}  λ (e: [Memory_Address_4G, list[Byte]],
      r:
      [Linear_memory[Physical_memory, pm_phy] →
      ExprResult[Linear_memory[Physical_memory, pm_phy], list[Byte]]
      (r ##
      (λ (tail: list[Byte]):
      (f'(e) ##
      (λ (head: list[Byte]): ok_result(append(head, tail))))))
      (cons2_var')(s'))
{-4}  is_linear_plain_memory?(pm')
{-5}  every(λ (e: [Memory_Address_4G, list[Byte]]):
      transformer_invariant?(pm' 'states, singleton(expr_2_super(f'(e))))
      (cons2_var')
{-6}  data(reduce(ok_result(null),
      λ (e: [Memory_Address_4G, list[Byte]],
      r:
      [Linear_memory[Physical_memory, pm_phy] →
      ExprResult[Linear_memory[Physical_memory, pm_phy], list[Byte]]
      (r ##
      (λ (tail: list[Byte]):
      (f'(e) ##
      (λ (head: list[Byte]): ok_result(append(head, tail))))))
      (cons2_var')(s'))
      =
      reduce(null, λ (e: [Address, list[Byte]], tail: list[Byte]): append(e'2, tail))
      (cons2_var')
1608 {-7}  ∀ (x: (pm' 'states)): OK?(f'(cons1_var')(x))
      {-8}  ∀ (x: (pm' 'states)):
      every(λ (e: [Memory_Address_4G, list[Byte]]): OK?(f'(e)(x)))(cons2_var')
      {-9}  ∀ (x: (pm' 'states)):
      OK?(f'(cons1_var')(x)) ⊃ data(f'(cons1_var')(x)) = cons1_var' '2
      {-10}  ∀ (x: (pm' 'states)):
      every(λ (e: [Memory_Address_4G, list[Byte]]):
      OK?(f'(e)(x)) ⊃ data(f'(e)(x)) = e'2)

```

Expanding the definition of `has_next_state`,

which is trivially true.

This completes the proof of `apply_side_effects_same_result.3.1.1.2.1`.

apply_side_effects_same_result.3.1.1.2.2:

```

{-1}  transformer_invariant?(pm' 'states,
      singleton(expr_2_super( $\lambda$  (s:
                                Lin-
                                reduce(ok_result(null),
                                       $\lambda$  (e:
                                          [Memory_Address_4G,
                                           list[Byte]],
                                          r:
                                          [Linear_memory
                                           [Physical_memory
                                           ExprResult
                                           [Linear_memory
                                           [Physical_memory
                                           list[Byte]]]):
                                          (r
                                           ##
                                           ( $\lambda$ 
                                           (tail: list[Byte]):
                                           (f'(e)
                                           ##
                                           ( $\lambda$ 
                                           (head: list[Byte]:
                                           ok_result
                                           (append
                                           (head, tail))))
                                           (cons2_var')(s))))))
      ear_memory[Physical_memory, pm_phy]):
{-2}  pm' 'states(s')
{-3}  OK?(reduce(ok_result(null),
                $\lambda$  (e: [Memory_Address_4G, list[Byte]],
                   r:
                   [Linear_memory[Physical_memory, pm_phy]  $\rightarrow$ 
                    ExprResult[Linear_memory[Physical_memory, pm_phy], list[Byte]]
                   (r ##
                    ( $\lambda$  (tail: list[Byte]):
                     (f'(e) ##
                      ( $\lambda$  (head: list[Byte]): ok_result(append(head, tail))))))
                   (cons2_var')(s')))
{-4}  is_linear_plain_memory?(pm')
{-5}  every( $\lambda$  (e: [Memory_Address_4G, list[Byte]]):
          transformer_invariant?(pm' 'states, singleton(expr_2_super(f'(e))))
          (cons2_var'))
{-6}  data(reduce(ok_result(null),
                $\lambda$  (e: [Memory_Address_4G, list[Byte]],
                   r:
                   [Linear_memory[Physical_memory, pm_phy]  $\rightarrow$ 
                    ExprResult[Linear_memory[Physical_memory, pm_phy], list[Byte]]
                   (r ##
                    ( $\lambda$  (tail: list[Byte]):
                     (f'(e) ##
                      ( $\lambda$  (head: list[Byte]): ok_result(append(head, tail))))))
                   (cons2_var')(s')))
      =
      reduce(null,  $\lambda$  (e: [Address, list[Byte]], tail: list[Byte]): append(e'2, tail))
      (cons2_var'))
1610 {-7}   $\forall$  (x: (pm' 'states)): OK?(f'(cons1_var')(x))
      {-8}   $\forall$  (x: (pm' 'states)):
            every( $\lambda$  (e: [Memory_Address_4G, list[Byte]]): OK?(f'(e)(x))(cons2_var'))
      {-9}   $\forall$  (x: (pm' 'states)):
            OK?(f'(cons1_var')(x))  $\supset$  data(f'(cons1_var')(x)) = cons1_var' '2
      {-10}  $\forall$  (x: (pm' 'states)):
            every( $\lambda$  (e: [Memory_Address_4G, list[Byte]]):
                OK?(f'(e)(x))  $\supset$  data(f'(e)(x)) = e'2)

```

Expanding the definition of singleton,

Applying decompose-equality,

Keeping (1) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `apply_side_effects_same_result.3.1.1.2.2`.

`apply_side_effects_same_result.3.1.2`:

<div style="display: flex; align-items: flex-start;"> <div style="margin-right: 10px;">{-1}</div> <div> $\forall (s_1: (\text{pm}'\text{'states})): \text{every}(\lambda (e: [\text{Memory_Address_4G}, \text{list}[\text{Byte}]]): \text{OK?}(f'(e)(s_1))(\text{cons2_var}'))$ </div> </div>	
<div style="display: flex; align-items: flex-start;"> <div style="margin-right: 10px;">{-2}</div> <div> $\text{is_linear_plain_memory?}(\text{pm}')$ </div> </div>	
<div style="display: flex; align-items: flex-start;"> <div style="margin-right: 10px;">{-3}</div> <div> $\text{pm}'\text{'states}(s')$ </div> </div>	
<div style="display: flex; align-items: flex-start;"> <div style="margin-right: 10px;">{-4}</div> <div> $\text{every}(\lambda (e: [\text{Memory_Address_4G}, \text{list}[\text{Byte}]]): \text{transformer_invariant?}(\text{pm}'\text{'states}, \text{singleton}(\text{expr_2_super}(f'(e))))(\text{cons2_var}'))$ </div> </div>	
<div style="display: flex; align-items: flex-start;"> <div style="margin-right: 10px;">{-5}</div> <div> $\forall (s_1: (\text{pm}'\text{'states})): \text{every}(\lambda (e: [\text{Memory_Address_4G}, \text{list}[\text{Byte}]]): \text{OK?}(f'(e)(s_1))(\text{cons}(\text{cons1_var}', \text{cons2_var}'))$ </div> </div>	
<div style="display: flex; align-items: flex-start;"> <div style="margin-right: 10px;">{-6}</div> <div> $\forall (s_1: (\text{pm}'\text{'states})): \text{every}(\lambda (e: [\text{Memory_Address_4G}, \text{list}[\text{Byte}]]): \text{OK?}(f'(e)(s_1)) \supset \text{data}(f'(e)(s_1)) = e'2)(\text{cons}(\text{cons1_var}', \text{cons2_var}'))$ </div> </div>	
<div style="display: flex; align-items: flex-start;"> <div style="margin-right: 10px;">{-7}</div> <div> $\text{transformer_invariant?}(\text{pm}'\text{'states}, \text{singleton}(\text{expr_2_super}(f'(\text{cons1_var}'))))$ </div> </div>	
<div style="display: flex; align-items: flex-start;"> <div style="margin-right: 10px;">{1}</div> <div> $\forall (s_1: (\text{pm}'\text{'states})): \text{every}(\lambda (e: [\text{Memory_Address_4G}, \text{list}[\text{Byte}]]): \text{OK?}(f'(e)(s_1)) \supset \text{data}(f'(e)(s_1)) = e'2)(\text{cons2_var}')$ </div> </div>	
<div style="display: flex; align-items: flex-start;"> <div style="margin-right: 10px;">{2}</div> <div> $\text{data}(\text{apply_side_effects}(\text{cons}(\text{cons1_var}', \text{cons2_var}'), f')(s')) = \text{reduce}(\text{null}, \lambda (e: [\text{Address}, \text{list}[\text{Byte}]], \text{tail}: \text{list}[\text{Byte}]): \text{append}(e'2, \text{tail}))(\text{cons}(\text{cons1_var}', \text{cons2_var}'))$ </div> </div>	

Repeatedly Skolemizing and flattening,

Instantiating quantified variables,

Expanding the definition of every,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `apply_side_effects_same_result.3.1.2`.

apply_side_effects_same_result.3.2:

<pre> {-1} is_linear_plain_memory?(pm') ∧ pm' `states(s') ∧ (∀ (s₁: (pm' `states)): every(λ (e: [Memory_Address_4G, list[Byte]]): OK?(f'(e)(s₁))) (cons2_var')) ∧ (∀ (s₁: (pm' `states)): every(λ (e: [Memory_Address_4G, list[Byte]]): OK?(f'(e)(s₁) ⊃ data(f'(e)(s₁)) = e'2) (cons2_var')) ∧ every(λ (e: [Memory_Address_4G, list[Byte]]): transformer_invariant?(pm' `states, singleton(expr_2_super(f'(e)))) (cons2_var')) ⊃ data(apply_side_effects(cons2_var', f')(s')) = reduce(null, λ (e: [Address, list[Byte]], tail: list[Byte]): append(e'2, tail)) (cons2_var')) {-2} is_linear_plain_memory?(pm') {-3} pm' `states(s') {-4} ∀ (s₁: (pm' `states)): every(λ (e: [Memory_Address_4G, list[Byte]]): OK?(f'(e)(s₁))) (cons(cons1_var', cons2_var')) {-5} ∀ (s₁: (pm' `states)): every(λ (e: [Memory_Address_4G, list[Byte]]): OK?(f'(e)(s₁) ⊃ data(f'(e)(s₁)) = e'2) (cons(cons1_var', cons2_var')) {-6} transformer_invariant?(pm' `states, singleton(expr_2_super(f'(cons1_var')))) ∧ every(λ (e: [Memory_Address_4G, list[Byte]]): transformer_invariant?(pm' `states, singleton(expr_2_super(f'(e)))) (cons2_var')) </pre>	<hr style="border: 0.5px solid black;"/> <pre> {1} ∀ (s₁: (pm' `states)): every(λ (e: [Memory_Address_4G, list[Byte]]): OK?(f'(e)(s₁)))(cons2_var') {2} data(apply_side_effects(cons(cons1_var', cons2_var'), f')(s')) = reduce(null, λ (e: [Address, list[Byte]], tail: list[Byte]): append(e'2, tail)) (cons(cons1_var', cons2_var')) </pre>
--	--

Repeatedly Skolemizing and flattening,

Expanding the definition of every,

Instantiating quantified variables,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `apply_side_effects_same_result.3.2`.

apply_side_effects_same_result.4:

```

{1}  ∀ (l: list[[Memory_Address_4G, list[Byte]]],
      pm: Plain_Memory[Linear_memory[Physical_memory, pm_phy]],
      s: Linear_memory[Physical_memory, pm_phy],
      f:
        [Memory_Address_4G, list[Byte] →
         [Linear_memory[Physical_memory, pm_phy] →
          ExprResult[Linear_memory[Physical_memory, pm_phy], list[Byte]]]]):
is_linear_plain_memory?(pm) ∧
pm' states(s) ∧
(∀ (s1: (pm' states)):
  every(λ (e: [Memory_Address_4G, list[Byte]]): OK?(f(e)(s1))(l))
  ∧
  (∀ (s1: (pm' states)):
    every(λ (e: [Memory_Address_4G, list[Byte]]):
      OK?(f(e)(s1) ⊃ data(f(e)(s1) = e'2)
      (l))
    ∧
    every(λ (e: [Memory_Address_4G, list[Byte]]):
      transformer_invariant?(pm' states, singleton(expr_2_super(f(e))))
      (l))
    ⊃
    OK?[Physical_memory, list[Byte]]
    (apply_side_effects[Physical_memory, pm_phy](l, f)(s))
{2}  ∀ (pm: Plain_Memory[Linear_memory], s: Linear_memory[Physical_memory, pm_phy],
      f:
        [Memory_Address_4G, list[Byte] →
         [Linear_memory → ExprResult[Linear_memory, list[Byte]]]]):
is_linear_plain_memory?(pm) ∧
pm' states(s) ∧
(∀ (s1: (pm' states)):
  every(λ (e: [Memory_Address_4G, list[Byte]]): OK?(f(e)(s1))(l'))
  ∧
  (∀ (s1: (pm' states)):
    every(λ (e: [Memory_Address_4G, list[Byte]]):
      OK?(f(e)(s1) ⊃ data(f(e)(s1) = e'2)
      (l'))
    ∧
    every(λ (e: [Memory_Address_4G, list[Byte]]):
      transformer_invariant?(pm' states, singleton(expr_2_super(f(e))))
      (l'))
    ⊃
    data(apply_side_effects(l', f)(s)) =
    reduce(null,
      λ (e: [Address, list[Byte]], tail: list[Byte]):
        LET head = e'2 IN append(head, tail)
      (l'))

```

Hiding formulas: 2,
 Repeatedly Skolemizing and flattening,
 Using lemma apply_side_effects_ok,

C Proof scripts

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `apply_side_effects_same_result.4`.

Q.E.D.

C.117.27 Linear_Memory_Properties.apply_side_effects_unchanged

Terse proof for `apply_side_effects_unchanged`.

`apply_side_effects_unchanged`:

```
{1}  ∃ (pm: Plain_Memory [Linear_memory], l: list [Memory_Address_4G, list [Byte]]),
      f:
        [Memory_Address_4G, list [Byte] →
         [Linear_memory → ExprResult [Linear_memory, list [Byte]]],
        addresses: PRED [Memory_Address_4G]:
is_linear_plain_memory?(pm) ∧
(addresses ⊆ restrict [Address, Memory_Address_4G, boolean] ((pm_phy'ro_addr ∪ pm_phy'rw-
^
every(λ (e: [Memory_Address_4G, list [Byte]]):
      unchanged_memory_invariant?(pm_phy'mem, pm_phy'states,
                                  singleton(expr_2_super(f(e))),
                                  extend [Address, Mem-
ory_Address_4G, bool, FALSE]
                                  (addresses)))
      (l)
    )
    ⊃
    unchanged_memory_invariant?(pm_phy'mem, pm_phy'states,
                                singleton(expr_2_super(apply_side_effects(l, f))),
                                extend [Address, Mem-
ory_Address_4G, bool, FALSE] (addresses))
```

Inducting on l on formula 1,

we get 2 subgoals:

apply_side_effects_unchanged.1:

```

{1}  ∃ (pm: Plain_Memory [Linear_memory],
      f: [Memory_Address_4G, list[Byte] →
          [Linear_memory → ExprResult [Linear_memory, list[Byte]]]),
      addresses: PRED [Memory_Address_4G]):
  is_linear_plain_memory?(pm) ∧
  (addresses ⊆ restrict [Address, Memory_Address_4G, boolean]((pm_phy'ro_addr ∪ pm_phy'rw_addr)))
  ∧
  every(λ (e: [Memory_Address_4G, list[Byte]]):
        unchanged_memory_invariant?(pm_phy'mem, pm_phy'states,
                                     singleton(expr_2_super(f(e))),
                                     extend[Address, Mem-
ory_Address_4G, bool, FALSE]
                                     (addresses)))
    (null)
  ⊃
  unchanged_memory_invariant?(pm_phy'mem, pm_phy'states,
                               singleton(expr_2_super(apply_side_effects(null, f))),
                               extend[Address, Mem-
ory_Address_4G, bool, FALSE](addresses))

```

Repeatedly Skolemizing and flattening,

Keeping (1) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of apply_side_effects_unchanged.1.

apply_side_effects_unchanged.2:

```

{1}  ∃ (cons1_var: [Memory_Address_4G, list[Byte]],
      cons2_var: list[[Memory_Address_4G, list[Byte]]]):
  (∃ (pm: Plain_Memory [Linear_memory],
    f:
      [Memory_Address_4G, list[Byte] →
        [Linear_memory → ExprResult [Linear_memory, list[Byte]]]],
    addresses: PRED [Memory_Address_4G]):
  is_linear_plain_memory?(pm) ∧
  (addresses ⊆ restrict [Address, Memory_Address_4G, boolean] ((pm_phy'ro_addr ∪ pm_phy'
  ∧
  every(λ (e: [Memory_Address_4G, list[Byte]]):
    unchanged_memory_invariant?(pm_phy' mem, pm_phy' states,
      singleton(expr_2_super(f(e))),
      extend[Address, Mem-
  ory_Address_4G, bool, FALSE]
      (addresses)))
    (cons2_var)
  ⊃
  unchanged_memory_invariant?(pm_phy' mem, pm_phy' states,
    singleton(expr_2_super(apply_side_effects
      (cons2_var, f))),
    extend[Address, Mem-
  ory_Address_4G, bool, FALSE]
    (addresses)))
  ⊃
  (∃ (pm: Plain_Memory [Linear_memory],
    f:
      [Memory_Address_4G, list[Byte] →
        [Linear_memory → ExprResult [Linear_memory, list[Byte]]]],
    addresses: PRED [Memory_Address_4G]):
  is_linear_plain_memory?(pm) ∧
  (addresses ⊆ restrict [Address, Memory_Address_4G, boolean] ((pm_phy'ro_addr ∪ pm_phy'
  ∧
  every(λ (e: [Memory_Address_4G, list[Byte]]):
    unchanged_memory_invariant?(pm_phy' mem, pm_phy' states,
      singleton(expr_2_super(f(e))),
      extend[Address, Mem-
  ory_Address_4G, bool, FALSE]
      (addresses)))
    (cons(cons1_var, cons2_var))
  ⊃
  unchanged_memory_invariant?(pm_phy' mem, pm_phy' states,
    singleton(expr_2_super(apply_side_effects
      (cons(cons1_var, cons2_v
      f))),
    extend[Address, Mem-
  ory_Address_4G, bool, FALSE]
    (addresses)))

```

Repeatedly Skolemizing and flattening,

C.117 Proofs for Linear_Memory_Properties (challenge-linear.pvs)

Instantiating quantified variables,

Expanding the definition of every,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Expanding the definition of apply_side_effects,

Expanding the definition of reduce,

Using lemma pm_plain_phy,

Using lemma pm_states,

Using lemma plain_memory_unchanged_composition [Linear_memory, list [Byte], list [Byte]],

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 4 subgoals:

```
apply_side_effects_unchanged.2.1:
```

```
{-1} plain_memory?(pm_phy)
{-2} unchanged_memory_invariant?(pm_phy' mem, pm_phy' states,
                                  singleton(expr_2_super(reduce(ok_result(null),
                                                              λ
                                                              (e:
                                                                [Memory_Address_4G
                                                                list[Byte]]),
                                                              r:
                                                                [Linear_memory
                                                                [Physical_memory,
                                                                ExprResult
                                                                [Linear_memory
                                                                [Physical_memory
                                                                list[Byte]]]):
                                                              (r
                                                                ##
                                                                (λ
                                                                  (tail: list[Byte]):
                                                                    (f'(e)
                                                                      ##
                                                                      (λ
                                                                        (head: list[Byte]
                                                                        ok_result
                                                                        (append
                                                                        (head, tail)))
                                                                      (cons2_var'))
                                                                ##
                                                                (λ (tail_1: list[Byte]):
                                                                  (f'(cons1_var'))
                                                                ##
                                                                (λ
                                                                  (head_1: list[Byte]):
                                                                    ok_result
                                                                    (append
                                                                    (head_1, tail_1))))))
                                                                extend[Address, Memory_Address_4G, bool, FALSE](addresses'
                                                                (cons2_var'))
                                                                (cons2_var'))
{-3} is_linear_plain_memory?(pm')
{-4} pm' states = pm_phy' states
{-5} (addresses' ⊆ restrict [Address, Memory_Address_4G, boolean]((pm_phy' ro_addr ∪ pm_phy' rw_addr)
{-6} every(λ (e: [Memory_Address_4G, list[Byte]]):
          unchanged_memory_invariant?(pm_phy' mem, pm_phy' states,
                                      singleton(expr_2_super(f'(e))),
                                      extend[Address, Memory_Address_4G, bool, FALSE]
                                      (addresses')))
          (cons2_var'))
{-7} unchanged_memory_invariant?(pm_phy' mem, pm_phy' states,
                                  singleton(expr_2_super(apply_side_effects(cons2_var', f'))),
                                  extend[Address, Memory_Address_4G, bool, FALSE](addresses'
                                  (cons2_var')))
{-8} unchanged_memory_invariant?(pm_phy' mem, pm_phy' states,
                                  singleton(expr_2_super(f'(cons1_var'))),
                                  extend[Address, Memory_Address_4G, bool, FALSE](addresses'
                                  (cons1_var')))
```

1618

```
Linear_memory
[Physical_memory, pm_p
(reduce(ok_result(null),
λ
(e:
[Memory_Address
list[Byte]]),
r:
[Physical_memory, pm_p
```

Expanding the definition of `expr_2_super`,

which is trivially true.

This completes the proof of `apply_side_effects_unchanged.2.1`.

apply_side_effects_unchanged.2.2:

```

{-1} plain_memory?(pm_phy)
{-2} is_linear_plain_memory?(pm')
{-3} pm' `states = pm_phy `states
{-4} (addresses' ⊆ restrict [Address, Memory_Address_4G, boolean] ((pm_phy `ro_addr ∪ pm_phy `rw_addr)))
{-5} every(λ (e: [Memory_Address_4G, list[Byte]]):
      unchanged_memory_invariant?(pm_phy `mem, pm_phy `states,
      singleton(expr_2_super(f'(e))),
      extend[Address, Memory_Address_4G, bool, FALSE]
      (addresses')))
      (cons2_var')
{-6} unchanged_memory_invariant?(pm_phy `mem, pm_phy `states,
      singleton(expr_2_super(apply_side_effects(cons2_var', f'))),
      extend[Address, Memory_Address_4G, bool, FALSE](addresses'))
{-7} unchanged_memory_invariant?(pm_phy `mem, pm_phy `states,
      singleton(expr_2_super(f'(cons1_var'))),
      extend[Address, Memory_Address_4G, bool, FALSE](addresses'))


---


{1} ∀ (d: (λ (l: list[Byte]): TRUE)):
      unchanged_memory_invariant?(pm_phy `mem, pm_phy `states,
      singleton(expr_2_super((f'(cons1_var') ##
      (λ (head_1: list[Byte]):
      ok_result
      (append(head_1, d)))))),
      extend[Address, Memory_Address_4G, bool, FALSE](addresses'))
{2} unchanged_memory_invariant?(pm_phy `mem, pm_phy `states,
      singleton(expr_2_super(λ (s:
      Linear_memory
      [Physical_memory, pm_phy]
      (reduce(ok_result(null),
      λ
      (e:
      [Memory_Address_4G,
      list[Byte]],
      r:
      [Linear_memory
      [Physical_memory,
      pm_phy] →
      Ex-
      [Linear_memory
      [Physical_memory,
      pm_phy],
      list[Byte]]))
      (r
      ##
      (λ
      (tail: list[Byte]
      (f'(e)
      ##
      (λ
      (head: list[Byte]
      ok_result
      (append
      (head,
      tail))))
      (cons2_var')
      ##
      (λ
      (tail_1: list[Byte]):
      (f'(cons1_var')
      ##

```

Hiding formulas: 2,

Repeatedly Skolemizing and flattening,

Using lemma plain_memory_unchanged_composition [Linear_memory, list[Byte], list[Byte]],

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 2 subgoals:

apply_side_effects_unchanged.2.2.1:

{-1}	plain_memory?(pm_phy)
{-2}	unchanged_memory_invariant?(pm_phy'mem, pm_phy'states, singleton(expr_2_super(f'(cons1_var'))), extend[Address, Memory_Address_4G, bool, FALSE](addresses'))
{-3}	every($\lambda (x: \text{number})$: number_field_pred(x) \wedge real_pred(x) \wedge rational_pred(x) \wedge integer_pred(x) \wedge $x \geq 0 \wedge x < \text{max_byte}$ (d')
{-4}	is_linear_plain_memory?(pm')
{-5}	pm' 'states = pm_phy 'states
{-6}	(addresses' \subseteq restrict [Address, Memory_Address_4G, boolean]((pm_phy'ro_addr \cup pm_phy'rw_addr)))
{-7}	every($\lambda (e: [\text{Memory_Address_4G}, \text{list}[\text{Byte}]])$): unchanged_memory_invariant?(pm_phy'mem, pm_phy'states, singleton(expr_2_super(f'(e))), extend[Address, Memory_Address_4G, bool, FALSE](addresses')) (cons2_var')
{-8}	unchanged_memory_invariant?(pm_phy'mem, pm_phy'states, singleton(expr_2_super(apply_side_effects(cons2_var', f'))), extend[Address, Memory_Address_4G, bool, FALSE](addresses'))
{1}	$\forall (d: (\lambda (l: \text{list}[\text{Byte}]): \text{TRUE}))$: unchanged_memory_invariant?(pm_phy'mem, pm_phy'states, singleton(expr_2_super(ok_result(append(d, d')))), extend[Address, Memory_Address_4G, bool, FALSE](addresses'))
{2}	unchanged_memory_invariant?(pm_phy'mem, pm_phy'states, singleton(expr_2_super((f'(cons1_var') ## $(\lambda (\text{head}_1: \text{list}[\text{Byte}]): \text{ok_result}(\text{append}(\text{head}_1, d'))$))), extend[Address, Memory_Address_4G, bool, FALSE](addresses'))

Repeatedly Skolemizing and flattening,

Keeping (1) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of apply_side_effects_unchanged.2.2.1.

C Proof scripts

`apply_side_effects_unchanged.2.2.2:`

{-1}	<code>plain_memory?(pm_phy)</code>
{-2}	<code>unchanged_memory_invariant?(pm_phy'mem, pm_phy'states, singleton(expr_2_super(f'(cons1_var'))), extend[Address, Memory_Address_4G, bool, FALSE](addresses'))</code>
{-3}	<code>every(λ (x: number): number_field_pred(x) ∧ real_pred(x) ∧ rational_pred(x) ∧ integer_pred(x) ∧ x ≥ 0 ∧ x < max_byte) (d')</code>
{-4}	<code>is_linear_plain_memory?(pm')</code>
{-5}	<code>pm'states = pm_phy'states</code>
{-6}	<code>(addresses' ⊆ restrict [Address, Memory_Address_4G, boolean]((pm_phy'ro_addr ∪ pm_phy'rw_addr)))</code>
{-7}	<code>every(λ (e: [Memory_Address_4G, list[Byte]]): unchanged_memory_invariant?(pm_phy'mem, pm_phy'states, singleton(expr_2_super(f'(e))), extend[Address, Memory_Address_4G, bool, FALSE] (addresses')) (cons2_var'))</code>
{-8}	<code>unchanged_memory_invariant?(pm_phy'mem, pm_phy'states, singleton(expr_2_super(apply_side_effects(cons2_var', f'))), extend[Address, Memory_Address_4G, bool, FALSE](addresses'))</code>
{1}	<code>(extend[Address, Memory_Address_4G, bool, FALSE](addresses') ⊆ (pm_phy'ro_addr ∪ pm_phy'rw_addr))</code>
{2}	<code>unchanged_memory_invariant?(pm_phy'mem, pm_phy'states, singleton(expr_2_super((f'(cons1_var') ## (λ (head_1: list[Byte]): ok_result (append(head_1, d'))))))), extend[Address, Memory_Address_4G, bool, FALSE](addresses'))</code>

Keeping (-6 1) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `apply_side_effects_unchanged.2.2.2`.

C.117 Proofs for Linear_Memory_Properties (challenge-linear.pvs)

apply_side_effects_unchanged.2.3:

```

{-1} plain_memory?(pm_phy)
{-2} is_linear_plain_memory?(pm')
{-3} pm' states = pm_phy states
{-4} (addresses' ⊆ restrict [Address, Memory_Address_4G, boolean]((pm_phy ro_addr ∪ pm_phy rw_addr)))
{-5} every(λ (e: [Memory_Address_4G, list[Byte]]):
    unchanged_memory_invariant?(pm_phy mem, pm_phy states,
    singleton(expr_2_super(f'(e))),
    extend[Address, Memory_Address_4G, bool, FALSE]
    (addresses'))
    (cons2_var'))
{-6} unchanged_memory_invariant?(pm_phy mem, pm_phy states,
    singleton(expr_2_super(apply_side_effects(cons2_var', f'))),
    extend[Address, Memory_Address_4G, bool, FALSE](addresses'))
{-7} unchanged_memory_invariant?(pm_phy mem, pm_phy states,
    singleton(expr_2_super(f'(cons1_var'))),
    extend[Address, Memory_Address_4G, bool, FALSE](addresses'))


---


{1} unchanged_memory_invariant?(pm_phy mem, pm_phy states,
    singleton(expr_2_super(reduce(ok_result(null),
    λ
    (e:
    [Memory_Address_4G,
    list[Byte]],
    r:
    [Linear_memory
    [Physical_memory, pm_phy] -
    ExprResult
    [Linear_memory
    [Physical_memory, pm_phy]
    list[Byte]])):
    (r
    ##
    (λ
    (tail: list[Byte]):
    (f'(e)
    ##
    (λ
    (head: list[Byte]):
    ok_result
    (append
    (head, tail)))))))
    (cons2_var'))),
    extend[Address, Memory_Address_4G, bool, FALSE](addresses'))
{2} unchanged_memory_invariant?(pm_phy mem, pm_phy states,
    singleton(expr_2_super(λ (s:
    Linear_memory
    [Physical_memory, pm_phy]):
    (reduce(ok_result(null),
    λ
    (e:
    [Memory_Address_4G,
    list[Byte]],
    r:
    [Linear_memory
    [Physical_memory, pm_phy] →
    ExprResult
    [Linear_memory
    [Physical_memory, pm_phy],
    list[Byte]])):
    prResult
    [Linear_memory
    [Physical_memory, pm_phy],
    list[Byte]])):

```


C Proof scripts

Hiding formulas: 2,

Expanding the definition of `apply_side_effects`,

Expanding the definition of `expr_2_super`,

which is trivially true.

This completes the proof of `apply_side_effects_unchanged.2.3`.

apply_side_effects_unchanged.2.4:

<pre> {-1} plain_memory?(pm_phy) {-2} is_linear_plain_memory?(pm') {-3} pm'`states = pm_phy`states {-4} (addresses' ⊆ restrict [Address, Memory_Address_4G, boolean]((pm_phy`ro_addr ∪ pm_phy`rw_addr))) {-5} every(λ (e: [Memory_Address_4G, list[Byte]]): unchanged_memory_invariant?(pm_phy`mem, pm_phy`states, singleton(expr_2_super(f'(e))), extend[Address, Mem- ory_Address_4G, bool, FALSE] (addresses')) (cons2_var')) {-6} unchanged_memory_invariant?(pm_phy`mem, pm_phy`states, singleton(expr_2_super(apply_side_effects(cons2_var', f'))), extend[Address, Memory_Address_4G, bool, FALSE](addresses')) {-7} unchanged_memory_invariant?(pm_phy`mem, pm_phy`states, singleton(expr_2_super(f'(cons1_var'))), extend[Address, Memory_Address_4G, bool, FALSE](addresses')) </pre>	<pre> {1} (extend[Address, Memory_Address_4G, bool, FALSE](addresses') ⊆ (pm_phy`ro_addr ∪ pm_phy`rw_addr)) {2} unchanged_memory_invariant?(pm_phy`mem, pm_phy`states, singleton(expr_2_super(λ (s: </pre>
<pre> prResult </pre>	<pre> Linear_memory [Physical_memory, pm_phy]): (reduce(ok_result(null), λ (e: [Memory_Address_4G, list[Byte]], r: [Linear_memory [Physical_memory, pm_phy] → Ex- [Linear_memory [Physical_memory, pm_phy], list[Byte]]]): (r ## (λ (tail: list[Byte]): (f'(e) ## (λ (head: list[Byte]): ok_result (append (head, tail)))))) (cons2_var')) ## (λ (tail_1: list[Byte]): (f'(cons1_var')) ## 1625 (λ (head_1: list[Byte]): ok_result (append (head_1, tail_1)))))) (s))), extend[Address, Memory_Address_4G, bool, FALSE](addresses')) </pre>

C Proof scripts

Keeping (-4 1) and hiding *,
 Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of `apply_side_effects_unchanged.2.4`.
 Q.E.D.

C.117.28 `Linear_Memory_Properties.subset_ptab_address_rw_addr`

Terse proof for `subset_ptab_address_rw_addr`.

`subset_ptab_address_rw_addr`:

$\{1\} \quad \forall (pm: \text{Plain_Memory}[\text{Linear_memory}], s: \text{Linear_memory}[\text{Physical_memory}, pm_phy], \text{base}: \text{Memory_Address_4G}):$ $(\text{address_in_pt_range?}(s, (\text{restrict} [\text{Address}, \text{Memory_Address_4G}, \text{bool}](pm'ro_addr) \cup \text{restrict} [Address, \text{Memory_Address_4G}, \text{bool}](pm'rw_addr))) \wedge$ $(\text{address_block}(\text{base}, \text{expt}(2, pe_size)) \subseteq \text{extend} [\text{Address}, \text{Memory_Address_4G}, \text{bool}, \text{FALSE}](\text{address_block}(\text{base}, \text{expt}(2, pe_size)), pm_phy'rw_addr))) \subseteq pm_phy'rw_addr$

Repeatedly Skolemizing and flattening,

Case splitting on `union(restrict[Address, Memory_Address_4G, boolean](pm!1'ro_addr), restrict[Address, Memory_Address_4G, boolean](pm!1'rw_addr)) = restrict[Address, Memory_Address_4G, boolean](union(pm!1'ro_addr, pm!1'rw_addr))`,

we get 2 subgoals:

`subset_ptab_address_rw_addr.1`:

$\{-1\} \quad (\text{restrict} [Address, \text{Memory_Address_4G}, \text{bool}](pm'ro_addr) \cup \text{restrict} [Address, \text{Memory_Address_4G}, \text{bool}](pm'rw_addr)) = \text{restrict} [Address, \text{Memory_Address_4G}, \text{bool}](\text{union}(pm'ro_addr, pm'rw_addr))$ $\{-2\} \quad \text{Mem?}(\text{base}'\text{type_of})$ $\{-3\} \quad 0 \leq \text{base}'\text{offset}$ $\{-4\} \quad \text{base}'\text{offset} < \text{max_linear_offset}$ $\{-5\} \quad (\text{address_in_pt_range?}(s', (\text{restrict} [Address, \text{Memory_Address_4G}, \text{bool}](pm'ro_addr) \cup \text{restrict} [Address, \text{Memory_Address_4G}, \text{bool}](pm'rw_addr)))) \wedge$ $\{-6\} \quad (\text{address_block}(\text{base}', \text{expt}(2, pe_size)) \subseteq \text{extend} [Address, \text{Memory_Address_4G}, \text{bool}, \text{FALSE}](\text{address_block}(\text{base}', \text{expt}(2, pe_size)), pm_phy'rw_addr))) \subseteq pm_phy'rw_addr$
--

Replacing using formula -1,

Using lemma `subset_transitive`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Keeping (-6 1) and hiding *,

Expanding the definition of `subset?`,

Repeatedly Skolemizing and flattening,

Instantiating the top quantifier in -2 with the terms: `(x!1)`,

we get 2 subgoals:

`subset_ptab_address_rw_addr.1.1`:

$\{-1\} \quad (x' \in \text{extend} [Address, \text{Memory_Address_4G}, \text{bool}, \text{FALSE}](\text{address_in_pt_range?}(s', \text{restrict} [Address, \text{Memory_Address_4G}, \text{bool}](pm'ro_addr) \cup \text{restrict} [Address, \text{Memory_Address_4G}, \text{bool}](pm'rw_addr)))) \wedge$ $\{-2\} \quad (x' \in \text{address_in_pt_range?}(s', \text{restrict} [Address, \text{Memory_Address_4G}, \text{bool}](\text{union}(pm'ro_addr, pm'rw_addr)))) \Rightarrow (x' \in \text{restrict} [Address, \text{Memory_Address_4G}, \text{bool}](pm_phy'rw_addr))$
$\{1\} \quad (x' \in pm_phy'rw_addr)$

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `subset_ptab_address_rw_addr.1.1`.


```
translate_transformer_invariant.1:
```

```

{-1} is_linear_plain_memory?(pm')
{-2} pm'`states = pm_phy`states
{-3} lvl' < max_level
{-4} Mem?(base'`type_of)
{-5} 0 ≤ base'`offset
{-6} base'`offset < max_linear_offset
{-7} aligned?(bits_per_level[Physical_memory, pm_phy] + pe_size)(offset(base'))
{-8} Mem?(lin_a'`type_of)
{-9} 0 ≤ lin_a'`offset
{-10} lin_a'`offset < max_linear_offset
{-11} union(pm'`ro_addr, pm'`rw_addr)(lin_a')
{-12} (address_block(xlat_idx(lvl', base', lin_a'), expt(2, pe_size)) ⊆ pm_phy`rw_addr)
-----
{1} transformer_invariant?(pm_phy`states,
    singleton(expr_2_super(read_data(pm_phy, pag-
        ing_data_type(lvl'))
                                (xlat_idx
                                    (lvl', base', lin_a')))))
{2} transformer_invariant?(pm_phy`states,
    singleton(expr_2_super((read_data(pm_phy, pag-
        ing_data_type(lvl'))
                                (xlat_idx(lvl', base', lin_a')))))
    ##
    (λ (pe: (range_pt(lvl'))):
        IF pag-
            ing_type?(lvl', pe)
                THEN IF present?(pe)
                    THEN IF ac-
                        cessible?(pe, access')
                            ∧
                            privi-
                                leged?(pe, priv')
                                    THEN write_data
                                        (pm_phy,
                                            pag-
                                                ing_data_type
                                                    (lvl'))
                                        (xlat_idx
                                            (lvl',
                                                base',
                                                lin_a'),
                                            set_refer-
                                                (pe, ac-
                                                    cessible?
                                                        (pe, access'),
                                                        TRUE,
                                                        lin_a'))
                                        ENDIF
                                    ELSE raise_fault
                                        (priv',
                                            access',
                                            TRUE,
                                            lin_a')
                                        ENDIF
                                ELSE raise_fault
                                    (priv',
                                        access',
                                        FALSE,
                                        lin_a')
                                    ENDIF
                            ELSE fatal_result
                                ENDIF))))))

```

Hiding formulas: 2,

Using lemma `pm_plain_phy`,

Rewriting using `plain_memory_transformer_invariant_read_data`, matching in `*`,

Expanding the definition of `in_blessed_memory?`,

Rewriting using `paging_data_type_size`, matching in `*`,

Keeping `(-13 1)` and hiding `*`,

Rewriting using `subset_transitive`, matching in `*` where `a` gets `address_block(xlat_idx(lvl!1, base!1, lin_a!1), expt(2, pe_size))`, `b` gets `pm_phy'rw_addr`,

Rewriting using `union_subset3`, matching in `*`,

This completes the proof of `translate_transformer_invariant.1`.

```
translate_transformer_invariant.2:
```

```

{-1} is_linear_plain_memory?(pm')
{-2} pm' 'states = pm_phy 'states
{-3} lvl' < max_level
{-4} Mem?(base' 'type_of)
{-5} 0 ≤ base' 'offset
{-6} base' 'offset < max_linear_offset
{-7} aligned?(bits_per_level[Physical_memory, pm_phy] + pe_size)(offset(base'))
{-8} Mem?(lin_a' 'type_of)
{-9} 0 ≤ lin_a' 'offset
{-10} lin_a' 'offset < max_linear_offset
{-11} union(pm' 'ro_addr, pm' 'rw_addr)(lin_a')
{-12} (address_block(xlat_idx(lvl', base', lin_a'), expt(2, pe_size)) ⊆ pm_phy 'rw_addr)
-----
{1} ∀ (d: (λ (p: (range_pt(lvl'))): TRUE)):
      transformer_invariant?(pm_phy 'states,
                             IF paging_type?(lvl', d)
                             THEN IF present?(d)
                             THEN IF accessible?(d, access') ∧ privi-
leged?(d, priv')
                             THEN singleton(expr_2_super(write_data
                                                         (pm_phy,
                                                         pag-
ing_data_type
                                                         (lvl'))
                                                         (xlat_idx
                                                         (lvl',
                                                         base',
                                                         lin_a'),
                                                         set_refere
                                                         (d, acces
                                                         ##
                                                         ok_result
                                                         (is_leaf?(d)
                                                         base(d))))
                             ELSE singleton(expr_2_super(raise_fault
                                                         (priv',
                                                         access',
                                                         TRUE,
                                                         lin_a'))))
                             ENDIF
                             ELSE singleton(expr_2_super(raise_fault
                                                         (priv',
                                                         access',
                                                         FALSE,
                                                         lin_a'))))
                             ENDIF
                             ELSE singleton(expr_2_super(fatal_result))
                             ENDIF)
{2} transformer_invariant?(pm_phy 'states,
                             singleton(expr_2_super((read_data(pm_phy, pag-
ing_data_type(lvl'))
                                                         (xlat_idx(lvl', base', lin_
##
(λ (pe: (range_pt(lvl'))):
IF pag-
ing_type?(lvl', pe)
THEN IF present?(pe)
THEN IF ac-
cessible?(pe, access')
^
privi-
leged?(pe, priv')
THEN write_data
(pm_phy

```

Hiding formulas: 2,

Repeatedly Skolemizing and flattening,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 5 subgoals:

translate_transformer_invariant.2.1:

<pre> {-1} range_pt(lvl')(d') {-2} paging_type?(lvl', d') {-3} present?(d') {-4} accessible?(d', access') {-5} privileged?(d', priv') {-6} is_linear_plain_memory?(pm') {-7} pm' states = pm_phy states {-8} lvl' < max_level {-9} Mem?(base' type_of) {-10} 0 ≤ base' offset {-11} base' offset < max_linear_offset {-12} aligned?(bits_per_level[Physical_memory, pm_phy] + pe_size)(offset(base')) {-13} Mem?(lin_a' type_of) {-14} 0 ≤ lin_a' offset {-15} lin_a' offset < max_linear_offset {-16} union(pm' ro_addr, pm' rw_addr)(lin_a') {-17} (address_block(xlat_idx(lvl', base', lin_a'), expt(2, pe_size)) ⊆ pm_phy rw_addr) </pre>	<pre> {1} transformer_invariant?(pm_phy states, singleton(expr_2_super(write_data(pm_phy, pag- ing_data_type(lvl'))) (xlat_idx(lvl', base', lin_a'), set_reference(d', access'))) ## ok_result(is_leaf?(d'), base(d')))) </pre>
--	--

Rewriting using expr_composition_transformer_invariant, matching in *,

we get 2 subgoals:

C Proof scripts

translate_transformer_invariant.2.1.1:

{-1}	range_pt(lvl')(d')
{-2}	paging_type?(lvl', d')
{-3}	present?(d')
{-4}	accessible?(d', access')
{-5}	privileged?(d', priv')
{-6}	is_linear_plain_memory?(pm')
{-7}	pm' 'states = pm_phy' 'states
{-8}	lvl' < max_level
{-9}	Mem?(base' 'type_of)
{-10}	0 ≤ base' 'offset
{-11}	base' 'offset < max_linear_offset
{-12}	aligned?(bits_per_level[Physical_memory, pm_phy] + pe_size)(offset(base'))
{-13}	Mem?(lin_a' 'type_of)
{-14}	0 ≤ lin_a' 'offset
{-15}	lin_a' 'offset < max_linear_offset
{-16}	union(pm' 'ro_addr, pm' 'rw_addr)(lin_a')
{-17}	(address_block(xlat_idx(lvl', base', lin_a'), expt(2, pe_size)) ⊆ pm_phy'rw_addr)
{1}	transformer_invariant?(pm_phy' 'states, singleton(expr_2_super(write_data(pm_phy, pag- ing_data_type(lvl')) (xlat_idx(lvl', base', lin_a' set_reference(d', access'))
{2}	transformer_invariant?(pm_phy' 'states, singleton(expr_2_super(write_data(pm_phy, pag- ing_data_type(lvl')) (xlat_idx(lvl', base', lin_a' set_reference(d', access')) ## ok_result(is_leaf?(d'), base(d')))

Hiding formulas: 2,

Using lemma pm_plain_phy,

Rewriting using plain_memory_transformer_invariant_write_data, matching in *,

Expanding the definition of in_blessed_memory?,

Rewriting using paging_data_type_size, matching in *,

This completes the proof of translate_transformer_invariant.2.1.1.

translate_transformer_invariant.2.1.2:

{-1}	range_pt(lvl')(d')	
{-2}	paging_type?(lvl', d')	
{-3}	present?(d')	
{-4}	accessible?(d', access')	
{-5}	privileged?(d', priv')	
{-6}	is_linear_plain_memory?(pm')	
{-7}	pm' 'states = pm_phy' 'states	
{-8}	lvl' < max_level	
{-9}	Mem?(base' 'type_of)	
{-10}	0 ≤ base' 'offset	
{-11}	base' 'offset < max_linear_offset	
{-12}	aligned?(bits_per_level[Physical_memory, pm_phy] + pe_size)(offset(base'))	
{-13}	Mem?(lin_a' 'type_of)	
{-14}	0 ≤ lin_a' 'offset	
{-15}	lin_a' 'offset < max_linear_offset	
{-16}	union(pm' 'ro_addr, pm' 'rw_addr)(lin_a')	
{-17}	(address_block(xlat_idx(lvl', base', lin_a'), expt(2, pe_size)) ⊆ pm_phy'rw_addr)	
{1}	transformer_invariant?(pm_phy' 'states, singleton(expr_2_super(ok_result(is_leaf?(d'), base(d')))))	
{2}	transformer_invariant?(pm_phy' 'states, singleton(expr_2_super(write_data(pm_phy, paging_data_type(lvl'), (xlat_idx(lvl', base', lin_a'), set_reference(d', access')))) # # ok_result(is_leaf?(d'), base(d'))))	

Keeping (1) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of translate_transformer_invariant.2.1.2.

translate_transformer_invariant.2.2:

{-1}	range_pt(lvl')(d')	
{-2}	paging_type?(lvl', d')	
{-3}	present?(d')	
{-4}	is_linear_plain_memory?(pm')	
{-5}	pm' 'states = pm_phy' 'states	
{-6}	lvl' < max_level	
{-7}	Mem?(base' 'type_of)	
{-8}	0 ≤ base' 'offset	
{-9}	base' 'offset < max_linear_offset	
{-10}	aligned?(bits_per_level[Physical_memory, pm_phy] + pe_size)(offset(base'))	
{-11}	Mem?(lin_a' 'type_of)	
{-12}	0 ≤ lin_a' 'offset	
{-13}	lin_a' 'offset < max_linear_offset	
{-14}	union(pm' 'ro_addr, pm' 'rw_addr)(lin_a')	
{-15}	(address_block(xlat_idx(lvl', base', lin_a'), expt(2, pe_size)) ⊆ pm_phy'rw_addr)	
{1}	privileged?(d', priv')	
{2}	transformer_invariant?(pm_phy' 'states, singleton(expr_2_super(raise_fault(priv', access', TRUE, lin_a'))))	

C Proof scripts

Rewriting using `raise_fault_transformer_invariant`, matching in `*` where `pm` gets `pm!1`,

This completes the proof of `translate_transformer_invariant.2.2`.

`translate_transformer_invariant.2.3`:

{-1}	<code>range_pt(lvl')(d')</code>	
{-2}	<code>paging_type?(lvl', d')</code>	
{-3}	<code>present?(d')</code>	
{-4}	<code>is_linear_plain_memory?(pm')</code>	
{-5}	<code>pm' 'states = pm_phy' 'states</code>	
{-6}	<code>lvl' < max_level</code>	
{-7}	<code>Mem?(base' 'type_of)</code>	
{-8}	<code>0 ≤ base' 'offset</code>	
{-9}	<code>base' 'offset < max_linear_offset</code>	
{-10}	<code>aligned?(bits_per_level[Physical_memory, pm_phy] + pe_size)(offset(base'))</code>	
{-11}	<code>Mem?(lin_a' 'type_of)</code>	
{-12}	<code>0 ≤ lin_a' 'offset</code>	
{-13}	<code>lin_a' 'offset < max_linear_offset</code>	
{-14}	<code>union(pm' 'ro_addr, pm' 'rw_addr)(lin_a')</code>	
{-15}	<code>(address_block(xlat_idx(lvl', base', lin_a'), expt(2, pe_size))) ⊆ pm_phy'rw_addr</code>	
{1}	<code>accessible?(d', access')</code>	
{2}	<code>transformer_invariant?(pm_phy' 'states,</code> <code>singleton(expr_2_super(raise_fault(priv',</code> <code>access',</code> <code>TRUE,</code> <code>lin_a'))))</code>	

Rewriting using `raise_fault_transformer_invariant`, matching in `*` where `pm` gets `pm!1`,

This completes the proof of `translate_transformer_invariant.2.3`.

`translate_transformer_invariant.2.4`:

{-1}	<code>range_pt(lvl')(d')</code>	
{-2}	<code>paging_type?(lvl', d')</code>	
{-3}	<code>is_linear_plain_memory?(pm')</code>	
{-4}	<code>pm' 'states = pm_phy' 'states</code>	
{-5}	<code>lvl' < max_level</code>	
{-6}	<code>Mem?(base' 'type_of)</code>	
{-7}	<code>0 ≤ base' 'offset</code>	
{-8}	<code>base' 'offset < max_linear_offset</code>	
{-9}	<code>aligned?(bits_per_level[Physical_memory, pm_phy] + pe_size)(offset(base'))</code>	
{-10}	<code>Mem?(lin_a' 'type_of)</code>	
{-11}	<code>0 ≤ lin_a' 'offset</code>	
{-12}	<code>lin_a' 'offset < max_linear_offset</code>	
{-13}	<code>union(pm' 'ro_addr, pm' 'rw_addr)(lin_a')</code>	
{-14}	<code>(address_block(xlat_idx(lvl', base', lin_a'), expt(2, pe_size))) ⊆ pm_phy'rw_addr</code>	
{1}	<code>present?(d')</code>	
{2}	<code>transformer_invariant?(pm_phy' 'states,</code> <code>singleton(expr_2_super(raise_fault(priv',</code> <code>access',</code> <code>FALSE,</code> <code>lin_a'))))</code>	

Rewriting using `raise_fault_transformer_invariant`, matching in `*` where `pm` gets `pm!1`,

This completes the proof of `translate_transformer_invariant.2.4`.

translate_transformer_invariant.2.5:

{-1}	range_pt(lvl')(d')
{-2}	is_linear_plain_memory?(pm')
{-3}	pm' 'states = pm_phy' 'states
{-4}	lvl' < max_level
{-5}	Mem?(base' 'type_of)
{-6}	0 ≤ base' 'offset
{-7}	base' 'offset < max_linear_offset
{-8}	aligned?(bits_per_level[Physical_memory, pm_phy] + pe_size)(offset(base'))
{-9}	Mem?(lin_a' 'type_of)
{-10}	0 ≤ lin_a' 'offset
{-11}	lin_a' 'offset < max_linear_offset
{-12}	union(pm' 'ro_addr, pm' 'rw_addr)(lin_a')
{-13}	(address_block(xlat_idx(lvl', base', lin_a'), expt(2, pe_size)) ⊆ pm_phy' 'rw_addr)
{1}	paging_type?(lvl', d')
{2}	transformer_invariant?(pm_phy' 'states, singleton(expr_2_super(fatal_result)))

Keeping (2) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of translate_transformer_invariant.2.5.

```
translate_transformer_invariant.3:
```

```

{-1} is_linear_plain_memory?(pm')
{-2} pm'`states = pm_phy`states
{-3} lvl' < max_level
{-4} Mem?(base'`type_of)
{-5} 0 ≤ base'`offset
{-6} base'`offset < max_linear_offset
{-7} aligned?(bits_per_level[Physical_memory, pm_phy] + pe_size)(offset(base'))
{-8} Mem?(lin_a'`type_of)
{-9} 0 ≤ lin_a'`offset
{-10} lin_a'`offset < max_linear_offset
{-11} union(pm'`ro_addr, pm'`rw_addr)(lin_a')
{-12} (address_block(xlat_idx(lvl', base', lin_a'), expt(2, pe_size)) ⊆ pm_phy`rw_addr)


---


{1} ∀ (pe: (range_pt(lvl'))):
      accessible?(pe, access') ∧
      privileged?(pe, priv') ∧ present?(pe) ∧ paging_type?(lvl', pe)
      ⊃ range_pt(lvl')(set_reference(pe, access'))
{2} transformer_invariant?(pm_phy`states,
      singleton(expr_2_super((read_data(pm_phy, pag-
      ing_data_type(lvl'))
      (xlat_idx(lvl', base', lin_
      ##
      (λ (pe: (range_pt(lvl'))):
      IF pag-
      THEN IF present?(pe)
      THEN IF ac-
      cessible?(pe, access')
      ∧
      privi-
      THEN write_data
      (pm_phy,
      pag-
      (lvl'))
      (xlat_idx
      (lvl',
      base',
      lin_a')
      set_refer
      (pe, ac-
      ##
      ok_result
      (is_leaf?(p
      base(pe)
      ELSE raise_fault
      (priv',
      access',
      TRUE,
      lin_a')
      ENDIF
      ELSE raise_fault
      (priv',
      access',
      FALSE,
      lin_a')
      ENDIF
      ELSE fatal_result
      ENDIF))))))

```

Keeping (1) and hiding *,
 Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of `translate_transformer_invariant.3`.
 Q.E.D.

C.117.30 Lin-ear_Memory_Properties.translate_unchanged_pm_phy_except_pe_TCC1

Terse proof for `translate_unchanged_pm_phy_except_pe_TCC1`.
`translate_unchanged_pm_phy_except_pe_TCC1:`

$ \begin{aligned} &\{1\} \quad \forall (lvl: \text{Level}, pm: \text{Plain_Memory}[\text{Linear_memory}], base: \text{PTab_Address}[\text{Physical_memory}, pm_phy], \\ &\quad \quad \quad lin_a: \text{Memory_Address_4G}): \\ &\quad (\forall (s: (pm'states))): \\ &\quad \quad \text{OK?}(\text{read_data}(pm_phy, \text{paging_data_type}(lvl))(\text{xlat_idx}(lvl, base, lin_a))(s)) \\ &\quad \quad \wedge \\ &\quad (\text{address_block}(\text{xlat_idx}(lvl, base, lin_a), \text{expt}(2, pe_size)) \subseteq pm_phy'rw_addr) \wedge \\ &\quad \quad \text{is_linear_plain_memory?}(pm) \wedge \text{union}(pm'ro_addr, pm'rw_addr)(lin_a) \\ &\quad \quad \supset \\ &\quad (\forall (s_1, s_2: (pm'states))): \\ &\quad \quad \text{OK?}[\text{Physical_memory}, ((\text{range_pt}(lvl)))] \\ &\quad \quad \quad (\text{read_data}[\text{Physical_memory}, ((\text{range_pt}(lvl)))] \\ &\quad \quad \quad \quad (pm_phy, \text{paging_data_type}(lvl)) \\ &\quad \quad \quad \quad (\text{xlat_idx}[\text{Physical_memory}, pm_phy](lvl, base, lin_a))(s_1)) \end{aligned} $
--

Repeatedly Skolemizing and flattening,
 Instantiating quantified variables,
 This completes the proof of `translate_unchanged_pm_phy_except_pe_TCC1`.
 Q.E.D.

C.117.31 Lin-ear_Memory_Properties.translate_unchanged_pm_phy_except_pe_TCC2

Terse proof for `translate_unchanged_pm_phy_except_pe_TCC2`.
`translate_unchanged_pm_phy_except_pe_TCC2:`

$ \begin{aligned} &\{1\} \quad \forall (lvl: \text{Level}, pm: \text{Plain_Memory}[\text{Linear_memory}], base: \text{PTab_Address}[\text{Physical_memory}, pm_phy], \\ &\quad \quad \quad lin_a: \text{Memory_Address_4G}): \\ &\quad (\forall (s: (pm'states))): \\ &\quad \quad \text{OK?}(\text{read_data}(pm_phy, \text{paging_data_type}(lvl))(\text{xlat_idx}(lvl, base, lin_a))(s)) \\ &\quad \quad \wedge \\ &\quad (\text{address_block}(\text{xlat_idx}(lvl, base, lin_a), \text{expt}(2, pe_size)) \subseteq pm_phy'rw_addr) \wedge \\ &\quad \quad \text{is_linear_plain_memory?}(pm) \wedge \text{union}(pm'ro_addr, pm'rw_addr)(lin_a) \\ &\quad \quad \supset \\ &\quad (\forall (s_1, s_2: (pm'states))): \\ &\quad \quad \text{OK?}[\text{Physical_memory}, ((\text{range_pt}(lvl)))] \\ &\quad \quad \quad (\text{read_data}[\text{Physical_memory}, ((\text{range_pt}(lvl)))] \\ &\quad \quad \quad \quad (pm_phy, \text{paging_data_type}(lvl)) \\ &\quad \quad \quad \quad (\text{xlat_idx}[\text{Physical_memory}, pm_phy](lvl, base, lin_a))(s_2)) \end{aligned} $
--

Repeatedly Skolemizing and flattening,

Instantiating quantified variables,

This completes the proof of `translate_unchanged_pm_phy_except_pe_TCC2`.

Q.E.D.

C.117.32

Linear_Memory_Properties.translate_unchanged_pm_phy_except_pe

Terse proof for `translate_unchanged_pm_phy_except_pe`.

`translate_unchanged_pm_phy_except_pe`:

```

{1}  ∃ (lvl: Level, pm: Plain_Memory[Linear_memory], base: PTab_Address, lin_a: Mem-
      ory_Address_4G,
      access: Memory_access, priv: Memory_privilege):
      (union(pm'ro_addr, pm'rw_addr)(lin_a) ∧
       is_linear_plain_memory?(pm) ∧
       (address_block(xlat_idx(lvl, base, lin_a), expt(2, pe_size)) ⊆ pm_phy'rw_addr) ∧
       (∃ (s: (pm'states)):
         OK?(read_data(pm_phy, paging_data_type(lvl))(xlat_idx(lvl, base, lin_a))
              (s)))
        ∧
        (∃ (s1, s2: (pm'states)):
          set_reference(data(read_data(pm_phy, paging_data_type(lvl))
                             (xlat_idx(lvl, base, lin_a))(s1)),
                       Write)
          =
          set_reference(data(read_data(pm_phy, paging_data_type(lvl))
                             (xlat_idx(lvl, base, lin_a))(s2)),
                       Write)))
       ⊃
      LET pe_addr = xlat_idx(lvl, base, lin_a) IN
      unchanged_memory_invariant?(pm_phy'mem, pm_phy'states,
                                  singleton(expr_2_super(translate(lvl,
                                                                    base,
                                                                    lin_a,
                                                                    ac-
                                                                    cess,
                                                                    priv))),
                                  ((pm_phy'ro_addr ∪ pm_phy'rw_addr) \ address_block(pe_a

```

Repeatedly Skolemizing and flattening,

Letting `addr_set` name `difference(union(pm_phy'ro_addr, pm_phy'rw_addr), address_block(xlat_idx(lvl!1, base!1, lin_a!1), expt(2, pe_size)))`,

Using lemma `pm_states`,

Using lemma `pm_plain_phy`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Replacing using formula -4,

Case splitting on `subset?(addr_set, union(pm_phy'ro_addr, pm_phy'rw_addr))`,

we get 2 subgoals:

translate_unchanged_pm_phy_except_pe.1:

<pre> {-1} (addr_set ⊆ (pm_phy'ro_addr ∪ pm_phy'rw_addr)) {-2} is_linear_plain_memory?(pm') {-3} plain_memory?(pm_phy) {-4} pm' states = pm_phy states {-5} ((pm_phy'ro_addr ∪ pm_phy'rw_addr) \ address_block(xlat_idx(lvl', base', lin_a'), expt(2, pe_size))) = addr_set {-6} lvl' < max_level {-7} Mem?(base' type_of) {-8} 0 ≤ base' offset {-9} base' offset < max_linear_offset {-10} aligned?(bits_per_level[Physical_memory, pm_phy] + pe_size)(offset(base')) {-11} Mem?(lin_a' type_of) {-12} 0 ≤ lin_a' offset {-13} lin_a' offset < max_linear_offset {-14} union(pm' ro_addr, pm' rw_addr)(lin_a') {-15} (address_block(xlat_idx(lvl', base', lin_a'), expt(2, pe_size)) ⊆ pm_phy'rw_addr) {-16} ∀ (s: (pm' states)): OK?(read_data(pm_phy, paging_data_type(lvl'))(xlat_idx(lvl', base', lin_a'))(s)) {-17} ∀ (s1, s2: (pm' states)): set_reference(data(read_data(pm_phy, paging_data_type(lvl')) (xlat_idx(lvl', base', lin_a'))(s1)), Write) = set_reference(data(read_data(pm_phy, paging_data_type(lvl')) (xlat_idx(lvl', base', lin_a'))(s2)), Write) </pre>	<pre> {1} unchanged_memory_invariant?(pm_phy' mem, pm_phy' states, singleton(expr_2_super(translate(lvl', base', lin_a', access', priv'))), addr_set) </pre>
--	---

Expanding the definition of translate,

Rewriting using plain_memory_unchanged_composition [Physical_memory, (range_pt(lvl!1)), [bool, Address]], matching in * where P gets LAMBDA (p: (range_pt(lvl!1))): FORALL (s: (pm_phy' states)): OK?(read_data(pm_phy, paging_data_type(lvl!1)) (xlat_idx(lvl!1, base!1, lin_a!1))(s)) IMPLIES set_reference(data(read_data(pm_phy, paging_data_type(lvl!1)) (xlat_idx(lvl!1, base!1, lin_a!1)) (s)), Write) = set_reference(p, Write), pm gets pm_phy, addresses gets addr_set,

we get 4 subgoals:

translate_unchanged_pm_phy_except_pe.1.1:

```

{-1} (addr_set  $\subseteq$  (pm_phy'ro_addr  $\cup$  pm_phy'rw_addr))
{-2} is_linear_plain_memory?(pm')
{-3} plain_memory?(pm_phy)
{-4} pm' 'states = pm_phy' 'states
{-5} ((pm_phy'ro_addr  $\cup$  pm_phy'rw_addr) \ address_block(xlat_idx(lvl', base', lin_a'), expt(2, pe_size)
    = addr_set
{-6} lvl' < max_level
{-7} Mem?(base' 'type_of)
{-8} 0  $\leq$  base' 'offset
{-9} base' 'offset < max_linear_offset
{-10} aligned?(bits_per_level[Physical_memory, pm_phy] + pe_size)(offset(base'))
{-11} Mem?(lin_a' 'type_of)
{-12} 0  $\leq$  lin_a' 'offset
{-13} lin_a' 'offset < max_linear_offset
{-14} union(pm' 'ro_addr, pm' 'rw_addr)(lin_a')
{-15} (address_block(xlat_idx(lvl', base', lin_a'), expt(2, pe_size))  $\subseteq$  pm_phy'rw_addr)
{-16}  $\forall$  (s: (pm' 'states)):
    OK?(read_data(pm_phy, paging_data_type(lvl'))(xlat_idx(lvl', base', lin_a'))(s))
{-17}  $\forall$  (s1, s2: (pm' 'states)):
    set_reference(data(read_data(pm_phy, paging_data_type(lvl'))
        (xlat_idx(lvl', base', lin_a'))(s1)),
        Write)
    =
    set_reference(data(read_data(pm_phy, paging_data_type(lvl'))
        (xlat_idx(lvl', base', lin_a'))(s2)),
        Write)


---


{1}  $\forall$  (s1: (pm_phy' 'states)):
    OK?(read_data(pm_phy, paging_data_type(lvl'))(xlat_idx(lvl', base', lin_a'))
        (s1))
     $\supset$ 
    ( $\forall$  (s: (pm_phy' 'states)):
        OK?(read_data(pm_phy, paging_data_type(lvl'))(xlat_idx(lvl', base', lin_a'))
            (s))
         $\supset$ 
        set_reference(data(read_data(pm_phy, paging_data_type(lvl'))
            (xlat_idx(lvl', base', lin_a'))(s)),
            Write)
        =
        set_reference(data(read_data(pm_phy, paging_data_type(lvl'))
            (xlat_idx(lvl', base', lin_a'))(s1)),
            Write))
{2} unchanged_memory_invariant?(pm_phy' 'mem, pm_phy' 'states,
    singleton(expr_2_super((read_data(pm_phy,
        pag-
        ing_data_type(lvl'))
        (xlat_idx
            (lvl', base', lin_a')
            ##
            ( $\lambda$  (pe: (range_pt(lvl'))):
                IF pag-
                ing_type?(lvl', pe)
                THEN IF present?(pe)
                THEN IF ac-
                cessible?
                (pe, acc-
                &
                priv-
                THEN write-
                (pm_ph-
                pag-
                ing_data_type
                (lvl'

```

Hiding formulas: 2,

Repeatedly Skolemizing and flattening,

Instantiating the top quantifier in -21 with the terms: (s!1 s!2),

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `translate_unchanged_pm_phy_except_pe.1.1`.

translate_unchanged_pm_phy_except_pe.1.2:

```

{-1} (addr_set ⊆ (pm_phy'ro_addr ∪ pm_phy'rw_addr))
{-2} is_linear_plain_memory?(pm')
{-3} plain_memory?(pm_phy)
{-4} pm' 'states = pm_phy' 'states
{-5} ((pm_phy'ro_addr ∪ pm_phy'rw_addr) \ address_block(xlat_idx(lvl', base', lin_a'), expt(2, pe_size)
    = addr_set
{-6} lvl' < max_level
{-7} Mem?(base' 'type_of)
{-8} 0 ≤ base' 'offset
{-9} base' 'offset < max_linear_offset
{-10} aligned?(bits_per_level[Physical_memory, pm_phy] + pe_size)(offset(base'))
{-11} Mem?(lin_a' 'type_of)
{-12} 0 ≤ lin_a' 'offset
{-13} lin_a' 'offset < max_linear_offset
{-14} union(pm' 'ro_addr, pm' 'rw_addr)(lin_a')
{-15} (address_block(xlat_idx(lvl', base', lin_a'), expt(2, pe_size)) ⊆ pm_phy'rw_addr)
{-16} ∀ (s: (pm' 'states)):
    OK?(read_data(pm_phy, paging_data_type(lvl'))(xlat_idx(lvl', base', lin_a'))(s))
{-17} ∀ (s1, s2: (pm' 'states)):
    set_reference(data(read_data(pm_phy, paging_data_type(lvl'))
        (xlat_idx(lvl', base', lin_a'))(s1)),
        Write)
    =
    set_reference(data(read_data(pm_phy, paging_data_type(lvl'))
        (xlat_idx(lvl', base', lin_a'))(s2)),
        Write)
-----
{1} unchanged_memory_invariant?(pm_phy' 'mem, pm_phy' 'states,
    singleton(expr_2_super(read_data(pm_phy, pag-
        ing_data_type(lvl'))
        (xlat_idx
            (lvl', base', lin_a')
            addr_set)
{2} unchanged_memory_invariant?(pm_phy' 'mem, pm_phy' 'states,
    singleton(expr_2_super((read_data(pm_phy,
        ing_data_type(lvl'))
        (xlat_idx
            (lvl', base', lin_a')
            ##
            (λ (pe: (range_pt(lvl'))):
                IF pag-
                    THEN IF present?(pe)
                        THEN IF ac-
                            cessible?
                                (pe, acc-
                                    ^
                                    priv-
                                THEN write_
                                    (pm_pl
                                        pag-
                                        (lvl')
                                        (xlat_i-
                                            (lvl'
                                                ba
                                                lin
                                                set.L
                                                (pe,
                                                ##
                                                ok_resu
                                                (is_leaf

```

C.117 Proofs for Linear_Memory_Properties (challenge-linear.pvs)

Hiding formulas: 2,

Using lemma `plain_memory_unchanged_memory_invariant_read_data` [`Physical_memory`, `(range_pt(lvl'))`],

Using lemma `unchanged_memory_invariant_mono`,

Expanding the definition of `in_blessed_memory?`,

Rewriting using `subset_reflexive`, matching in `*`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Rewriting using `paging_data_type_size`, matching in `*`,

Simplifying, rewriting, and recording with decision procedures,

Keeping `(-15 3)` and hiding `*`,

Rewriting using `subset_transitive`, matching in `*` where `a` gets `address_block(xlat_idx(lvl!1, base!1, lin_a!1), expt(2, pe_size))`, `b` gets `pm_phy'rw_addr`, `c` gets `union(pm_phy'ro_addr, pm_phy'rw_addr)`,

Rewriting using `union_subset3`, matching in `*`,

This completes the proof of `translate_unchanged_pm_phy_except_pe.1.2`.

translate_unchanged_pm_phy_except_pe.1.3:

```

{-1}  (addr_set  $\subseteq$  (pm_phy'ro_addr  $\cup$  pm_phy'rw_addr))
{-2}  is_linear_plain_memory?(pm')
{-3}  plain_memory?(pm_phy)
{-4}  pm' 'states = pm_phy' 'states
{-5}  ((pm_phy'ro_addr  $\cup$  pm_phy'rw_addr) \ address_block(xlat_idx(lvl', base', lin_a'), expt(2, pe_size)
      = addr_set
{-6}  lvl' < max_level
{-7}  Mem?(base' 'type_of)
{-8}  0  $\leq$  base' 'offset
{-9}  base' 'offset < max_linear_offset
{-10} aligned?(bits_per_level[Physical_memory, pm_phy] + pe_size)(offset(base'))
{-11} Mem?(lin_a' 'type_of)
{-12} 0  $\leq$  lin_a' 'offset
{-13} lin_a' 'offset < max_linear_offset
{-14} union(pm' 'ro_addr, pm' 'rw_addr)(lin_a')
{-15} (address_block(xlat_idx(lvl', base', lin_a'), expt(2, pe_size))  $\subseteq$  pm_phy'rw_addr)
{-16}  $\forall$  (s: (pm' 'states)):
      OK?(read_data(pm_phy, paging_data_type(lvl'))(xlat_idx(lvl', base', lin_a'))(s))
{-17}  $\forall$  (s1, s2: (pm' 'states)):
      set_reference(data(read_data(pm_phy, paging_data_type(lvl'))
                          (xlat_idx(lvl', base', lin_a'))(s1)),
                    Write)
      =
      set_reference(data(read_data(pm_phy, paging_data_type(lvl'))
                          (xlat_idx(lvl', base', lin_a'))(s2)),
                    Write)
-----
{1}   $\forall$  (d:
      ( $\lambda$  (p: (range_pt(lvl'))):
         $\forall$  (s: (pm_phy' 'states)):
          OK?(read_data(pm_phy, paging_data_type(lvl'))
                (xlat_idx(lvl', base', lin_a'))(s))
           $\supset$ 
          set_reference(data(read_data(pm_phy, paging_data_type(lvl'))
                            (xlat_idx(lvl', base', lin_a'))(s)),
                        Write)
          = set_reference(p, Write)))
      unchanged_memory_invariant?(pm_phy' mem, pm_phy' states,
        IF paging_type?(lvl', d)
        THEN IF present?(d)
          THEN IF accessible?(d, access')  $\wedge$ 
                privileged?(d, priv')
            THEN singleton(expr_2_super
                          (write_data
                           (pm_phy,
                            pag-
ing_data_type(lvl'))
                          (xlat_idx
                           (lvl', base', lin_a')
                           set_reference(d, a
##
ok_result
(is_leaf?(d), base(d)
ELSE singleton(expr_2_super
               (raise_fault
                (priv',
                 access',
                 TRUE,
                 lin_a'))))
        ENDIF
      ELSE singleton(expr_2_super(raise_fault
                                (priv',
                                 access',
                                 FALSE,

```

Hiding formulas: 2,

Repeatedly Skolemizing and flattening,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 5 subgoals:

```
translate_unchanged_pm_phy_except_pe.1.3.1:
```

```
{-1} range_pt(lvl')(d')
{-2}  $\forall (s: (\text{pm\_phy}'\text{states})):$ 
      OK?(read_data(pm_phy, paging_data_type(lvl'))(xlat_idx(lvl', base', lin_a'))(s))
       $\supset$ 
      set_reference(data(read_data(pm_phy, paging_data_type(lvl'))
                          (xlat_idx(lvl', base', lin_a'))(s)),
                    Write)
      = set_reference(d', Write)
{-3} paging_type?(lvl', d')
{-4} present?(d')
{-5} accessible?(d', access')
{-6} privileged?(d', priv')
{-7} (addr_set  $\subseteq$  (pm_phy'ro_addr  $\cup$  pm_phy'rw_addr))
{-8} is_linear_plain_memory?(pm')
{-9} plain_memory?(pm_phy)
{-10} pm' 'states = pm_phy' 'states
{-11} ((pm_phy'ro_addr  $\cup$  pm_phy'rw_addr) \ address_block(xlat_idx(lvl', base', lin_a'), expt(2, pe_size)
      = addr_set
{-12} lvl' < max_level
{-13} Mem?(base' 'type_of)
{-14} 0  $\leq$  base' 'offset
{-15} base' 'offset < max_linear_offset
{-16} aligned?(bits_per_level[Physical_memory, pm_phy] + pe_size)(offset(base'))
{-17} Mem?(lin_a' 'type_of)
{-18} 0  $\leq$  lin_a' 'offset
{-19} lin_a' 'offset < max_linear_offset
{-20} union(pm' 'ro_addr, pm' 'rw_addr)(lin_a')
{-21} (address_block(xlat_idx(lvl', base', lin_a'), expt(2, pe_size))  $\subseteq$  pm_phy'rw_addr)
{-22}  $\forall (s: (\text{pm}'\text{states})):$ 
      OK?(read_data(pm_phy, paging_data_type(lvl'))(xlat_idx(lvl', base', lin_a'))(s))
{-23}  $\forall (s_1, s_2: (\text{pm}'\text{states})):$ 
      set_reference(data(read_data(pm_phy, paging_data_type(lvl'))
                          (xlat_idx(lvl', base', lin_a'))(s_1)),
                    Write)
      =
      set_reference(data(read_data(pm_phy, paging_data_type(lvl'))
                          (xlat_idx(lvl', base', lin_a'))(s_2)),
                    Write)
-----
{1} unchanged_memory_invariant?(pm_phy' 'mem, pm_phy' 'states,
                                singleton(expr_2_super(write_data(pm_phy,
                                                                pag-
                                                                ing_data_type(lvl'))
                                                                (xlat_idx
                                                                (lvl', base', lin_a')
                                                                set_reference(d', a
                                                                ##
                                                                ok_result(is_leaf?(d'),
                                                                base(d')))),
                                addr_set)
```

Using lemma `plain_memory_unchanged_composition` `[Physical_memory, Unit, [bool, Address]]`,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 3 subgoals:

Expanding the definition of ##,

Expanding the definition of ##,

which is trivially true.

This completes the proof of `translate_unchanged_pm_phy_except_pe.1.3.1.1`.

translate_unchanged_pm_phy_except_pe.1.3.1.2:

```

{-1} plain_memory?(pm_phy)
{-2} (addr_set  $\subseteq$  (pm_phy'ro_addr  $\cup$  pm_phy'rw_addr))
{-3} range_pt(lvl')(d')
{-4}  $\forall$  (s: (pm_phy'states)):
      OK?(read_data(pm_phy, paging_data_type(lvl'))(xlat_idx(lvl', base', lin_a'))(s))
       $\supset$ 
      set_reference(data(read_data(pm_phy, paging_data_type(lvl'))
                          (xlat_idx(lvl', base', lin_a'))(s)),
                    Write)
      = set_reference(d', Write)
{-5} paging_type?(lvl', d')
{-6} present?(d')
{-7} accessible?(d', access')
{-8} privileged?(d', priv')
{-9} is_linear_plain_memory?(pm')
{-10} pm' 'states = pm_phy 'states
{-11} ((pm_phy'ro_addr  $\cup$  pm_phy'rw_addr) \ address_block(xlat_idx(lvl', base', lin_a'), expt(2, pe_size)
      = addr_set
{-12} lvl' < max_level
{-13} Mem?(base' 'type_of)
{-14} 0  $\leq$  base' 'offset
{-15} base' 'offset < max_linear_offset
{-16} aligned?(bits_per_level[Physical_memory, pm_phy] + pe_size)(offset(base'))
{-17} Mem?(lin_a' 'type_of)
{-18} 0  $\leq$  lin_a' 'offset
{-19} lin_a' 'offset < max_linear_offset
{-20} union(pm' 'ro_addr, pm' 'rw_addr)(lin_a')
{-21} (address_block(xlat_idx(lvl', base', lin_a'), expt(2, pe_size))  $\subseteq$  pm_phy'rw_addr)
{-22}  $\forall$  (s: (pm' 'states)):
      OK?(read_data(pm_phy, paging_data_type(lvl'))(xlat_idx(lvl', base', lin_a'))(s))
{-23}  $\forall$  (s1, s2: (pm' 'states)):
      set_reference(data(read_data(pm_phy, paging_data_type(lvl'))
                          (xlat_idx(lvl', base', lin_a'))(s1)),
                    Write)
      =
      set_reference(data(read_data(pm_phy, paging_data_type(lvl'))
                          (xlat_idx(lvl', base', lin_a'))(s2)),
                    Write)


---


{1}  $\forall$  (d: ( $\lambda$  (u: Unit): TRUE)):
      unchanged_memory_invariant?(pm_phy 'mem, pm_phy 'states,
      singleton(expr_2_super(ok_result
      [Physical_memory, [bool
      dress]])
      (is_leaf?(d'), base(d'))),
      addr_set)
{2} unchanged_memory_invariant?(pm_phy 'mem, pm_phy 'states,
      singleton(expr_2_super(write_data(pm_phy, pag-
      ing_data_type(lvl'))
      (xlat_idx
      (lvl', base', lin_a')
      set_reference(d', a
      ##
      ok_result(is_leaf?(d'),
      base(d')))),
      addr_set)

```

Hiding formulas: 2,

Keeping (1) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `translate_unchanged_pm_phy_except_pe.1.3.1.2`.

translate_unchanged_pm_phy_except_pe.1.3.1.3:

```

{-1} plain_memory?(pm_phy)
{-2} (addr_set  $\subseteq$  (pm_phy'ro_addr  $\cup$  pm_phy'rw_addr))
{-3} range_pt(lvl')(d')
{-4}  $\forall$  (s: (pm_phy'states)):
      OK?(read_data(pm_phy, paging_data_type(lvl'))(xlat_idx(lvl', base', lin_a'))(s))
       $\supset$ 
      set_reference(data(read_data(pm_phy, paging_data_type(lvl'))
                          (xlat_idx(lvl', base', lin_a'))(s)),
                    Write)
      = set_reference(d', Write)
{-5} paging_type?(lvl', d')
{-6} present?(d')
{-7} accessible?(d', access')
{-8} privileged?(d', priv')
{-9} is_linear_plain_memory?(pm')
{-10} pm' 'states = pm_phy' 'states
{-11} ((pm_phy'ro_addr  $\cup$  pm_phy'rw_addr) \ address_block(xlat_idx(lvl', base', lin_a'), expt(2, pe_size)
      = addr_set
{-12} lvl' < max_level
{-13} Mem?(base' 'type_of)
{-14} 0  $\leq$  base' 'offset
{-15} base' 'offset < max_linear_offset
{-16} aligned?(bits_per_level[Physical_memory, pm_phy] + pe_size)(offset(base'))
{-17} Mem?(lin_a' 'type_of)
{-18} 0  $\leq$  lin_a' 'offset
{-19} lin_a' 'offset < max_linear_offset
{-20} union(pm' 'ro_addr, pm' 'rw_addr)(lin_a')
{-21} (address_block(xlat_idx(lvl', base', lin_a'), expt(2, pe_size))  $\subseteq$  pm_phy'rw_addr)
{-22}  $\forall$  (s: (pm' 'states)):
      OK?(read_data(pm_phy, paging_data_type(lvl'))(xlat_idx(lvl', base', lin_a'))(s))
{-23}  $\forall$  (s1, s2: (pm' 'states)):
      set_reference(data(read_data(pm_phy, paging_data_type(lvl'))
                          (xlat_idx(lvl', base', lin_a'))(s1)),
                    Write)
      =
      set_reference(data(read_data(pm_phy, paging_data_type(lvl'))
                          (xlat_idx(lvl', base', lin_a'))(s2)),
                    Write)
-----
{1} unchanged_memory_invariant?(pm_phy'mem, pm_phy'states,
                                singleton(expr_2_super(write_data(pm_phy,
                                                                pag-
                                                                ing_data_type(lvl'))
                                                                (xlat_idx
                                                                (lvl', base', lin_a')
                                                                set_reference
                                                                (d', access')))),
                                addr_set)
{2} unchanged_memory_invariant?(pm_phy'mem, pm_phy'states,
                                singleton(expr_2_super(write_data(pm_phy,
                                                                pag-
                                                                ing_data_type(lvl'))
                                                                (xlat_idx
                                                                (lvl', base', lin_a')
                                                                set_reference(d', a
                                                                ##
                                                                ok_result(is_leaf?(d'),
                                                                base(d')))),
                                addr_set)

```

Hiding formulas: 2,

Replacing using formula -10,

Using lemma plain_memory_unchanged_memory_invariant_write_data,

Expanding the definition of in_blessed_memory?,

Rewriting using paging_data_type_size, matching in *,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of translate_unchanged_pm_phy_except_pe.1.3.1.3.

translate_unchanged_pm_phy_except_pe.1.3.2:

<pre> {-1} range_pt(lvl')(d') {-2} ∀ (s: (pm_phy' states)): OK?(read_data(pm_phy, paging_data_type(lvl'))(xlat_idx(lvl', base', lin_a'))(s)) ⊃ set_reference(data(read_data(pm_phy, paging_data_type(lvl')) (xlat_idx(lvl', base', lin_a'))(s)), Write) = set_reference(d', Write) {-3} paging_type?(lvl', d') {-4} present?(d') {-5} (addr_set ⊆ (pm_phy'ro_addr ∪ pm_phy'rw_addr)) {-6} is_linear_plain_memory?(pm') {-7} plain_memory?(pm_phy) {-8} pm' states = pm_phy' states {-9} ((pm_phy'ro_addr ∪ pm_phy'rw_addr) \ address_block(xlat_idx(lvl', base', lin_a'), expt(2, pe_size))) = addr_set {-10} lvl' < max_level {-11} Mem?(base' type_of) {-12} 0 ≤ base' offset {-13} base' offset < max_linear_offset {-14} aligned?(bits_per_level[Physical_memory, pm_phy] + pe_size)(offset(base')) {-15} Mem?(lin_a' type_of) {-16} 0 ≤ lin_a' offset {-17} lin_a' offset < max_linear_offset {-18} union(pm'ro_addr, pm'rw_addr)(lin_a') {-19} (address_block(xlat_idx(lvl', base', lin_a'), expt(2, pe_size))) ⊆ pm_phy'rw_addr {-20} ∀ (s: (pm' states)): OK?(read_data(pm_phy, paging_data_type(lvl'))(xlat_idx(lvl', base', lin_a'))(s)) {-21} ∀ (s₁, s₂: (pm' states)): set_reference(data(read_data(pm_phy, paging_data_type(lvl')) (xlat_idx(lvl', base', lin_a'))(s₁)), Write) = set_reference(data(read_data(pm_phy, paging_data_type(lvl')) (xlat_idx(lvl', base', lin_a'))(s₂)), Write) </pre>	<hr style="border: 0.5px solid black;"/> <pre> {1} privileged?(d', priv') {2} unchanged_memory_invariant?(pm_phy' mem, pm_phy' states, singleton(expr_2_super(raise_fault(priv', access', TRUE, lin_a'))), addr_set) </pre>
---	---

Using lemma raise_fault_unchanged_memory_invariant,

C Proof scripts

Using lemma `unchanged_memory_invariant_mono`,
 Rewriting using `subset_reflexive`, matching in `*`,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 This completes the proof of `translate_unchanged_pm_phy_except_pe.1.3.2`.
`translate_unchanged_pm_phy_except_pe.1.3.3`:

<pre> {-1} range_pt(lvl')(d') {-2} ∀ (s: (pm_phy' states)): OK?(read_data(pm_phy, paging_data_type(lvl'))(xlat_idx(lvl', base', lin_a'))(s)) ⊃ set_reference(data(read_data(pm_phy, paging_data_type(lvl')) (xlat_idx(lvl', base', lin_a'))(s)), Write) = set_reference(d', Write) {-3} paging_type?(lvl', d') {-4} present?(d') {-5} (addr_set ⊆ (pm_phy'ro_addr ∪ pm_phy'rw_addr)) {-6} is_linear_plain_memory?(pm') {-7} plain_memory?(pm_phy) {-8} pm' states = pm_phy states {-9} ((pm_phy'ro_addr ∪ pm_phy'rw_addr) \ address_block(xlat_idx(lvl', base', lin_a'), expt(2, pe_size) = addr_set {-10} lvl' < max_level {-11} Mem?(base' type_of) {-12} 0 ≤ base' offset {-13} base' offset < max_linear_offset {-14} aligned?(bits_per_level[Physical_memory, pm_phy] + pe_size)(offset(base')) {-15} Mem?(lin_a' type_of) {-16} 0 ≤ lin_a' offset {-17} lin_a' offset < max_linear_offset {-18} union(pm'ro_addr, pm'rw_addr)(lin_a') {-19} (address_block(xlat_idx(lvl', base', lin_a'), expt(2, pe_size)) ⊆ pm_phy'rw_addr) {-20} ∀ (s: (pm' states)): OK?(read_data(pm_phy, paging_data_type(lvl'))(xlat_idx(lvl', base', lin_a'))(s)) {-21} ∀ (s₁, s₂: (pm' states)): set_reference(data(read_data(pm_phy, paging_data_type(lvl')) (xlat_idx(lvl', base', lin_a'))(s₁)), Write) = set_reference(data(read_data(pm_phy, paging_data_type(lvl')) (xlat_idx(lvl', base', lin_a'))(s₂)), Write) </pre>	<pre> {1} accessible?(d', access') {2} unchanged_memory_invariant?(pm_phy' mem, pm_phy' states, singleton(expr_2_super(raise_fault(priv', access', TRUE, lin_a'))), addr_set) </pre>
--	--

Using lemma `raise_fault_unchanged_memory_invariant`,
 Using lemma `unchanged_memory_invariant_mono`,
 Rewriting using `subset_reflexive`, matching in `*`,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `translate_unchanged_pm_phy_except_pe.1.3.3`.

`translate_unchanged_pm_phy_except_pe.1.3.4`:

<pre> {-1} range_pt(lvl')(d') {-2} ∀ (s: (pm_phy'states)): OK?(read_data(pm_phy, paging_data_type(lvl'))(xlat_idx(lvl', base', lin_a'))(s)) ⊃ set_reference(data(read_data(pm_phy, paging_data_type(lvl')) (xlat_idx(lvl', base', lin_a'))(s)), Write) = set_reference(d', Write) {-3} paging_type?(lvl', d') {-4} (addr_set ⊆ (pm_phy'ro_addr ∪ pm_phy'rw_addr)) {-5} is_linear_plain_memory?(pm') {-6} plain_memory?(pm_phy) {-7} pm'states = pm_phy'states {-8} ((pm_phy'ro_addr ∪ pm_phy'rw_addr) \ address_block(xlat_idx(lvl', base', lin_a'), expt(2, pe_size))) = addr_set {-9} lvl' < max_level {-10} Mem?(base' type_of) {-11} 0 ≤ base'offset {-12} base'offset < max_linear_offset {-13} aligned?(bits_per_level[Physical_memory, pm_phy] + pe_size)(offset(base')) {-14} Mem?(lin_a' type_of) {-15} 0 ≤ lin_a'offset {-16} lin_a'offset < max_linear_offset {-17} union(pm'ro_addr, pm'rw_addr)(lin_a') {-18} (address_block(xlat_idx(lvl', base', lin_a'), expt(2, pe_size)) ⊆ pm_phy'rw_addr) {-19} ∀ (s: (pm'states)): OK?(read_data(pm_phy, paging_data_type(lvl'))(xlat_idx(lvl', base', lin_a'))(s)) {-20} ∀ (s₁, s₂: (pm'states)): set_reference(data(read_data(pm_phy, paging_data_type(lvl')) (xlat_idx(lvl', base', lin_a'))(s₁)), Write) = set_reference(data(read_data(pm_phy, paging_data_type(lvl')) (xlat_idx(lvl', base', lin_a'))(s₂)), Write) </pre>	<pre> {1} present?(d') {2} unchanged_memory_invariant?(pm_phy'mem, pm_phy'states, singleton(expr_2_super(raise_fault(priv', access', FALSE, lin_a'))), addr_set) </pre>
---	--

Using lemma `raise_fault_unchanged_memory_invariant`,

Using lemma `unchanged_memory_invariant_mono`,

Rewriting using `subset_reflexive`, matching in `*`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `translate_unchanged_pm_phy_except_pe.1.3.4`.

translate_unchanged_pm_phy_except_pe.1.3.5:

<pre> {-1} range_pt(lvl')(d') {-2} $\forall (s: (\text{pm_phy}'\text{states})):$ OK?(read_data(pm_phy, paging_data_type(lvl'))(xlat_idx(lvl', base', lin_a'))(s)) \supset set_reference(data(read_data(pm_phy, paging_data_type(lvl')) (xlat_idx(lvl', base', lin_a'))(s)), Write) = set_reference(d', Write) {-3} (addr_set \subseteq (pm_phy'ro_addr \cup pm_phy'rw_addr)) {-4} is_linear_plain_memory?(pm') {-5} plain_memory?(pm_phy) {-6} pm' 'states = pm_phy' 'states {-7} ((pm_phy'ro_addr \cup pm_phy'rw_addr) \ address_block(xlat_idx(lvl', base', lin_a'), expt(2, pe_size) = addr_set {-8} lvl' < max_level {-9} Mem?(base' 'type_of) {-10} 0 \leq base' 'offset {-11} base' 'offset < max_linear_offset {-12} aligned?(bits_per_level[Physical_memory, pm_phy] + pe_size)(offset(base')) {-13} Mem?(lin_a' 'type_of) {-14} 0 \leq lin_a' 'offset {-15} lin_a' 'offset < max_linear_offset {-16} union(pm' 'ro_addr, pm' 'rw_addr)(lin_a') {-17} (address_block(xlat_idx(lvl', base', lin_a'), expt(2, pe_size)) \subseteq pm_phy'rw_addr) {-18} $\forall (s: (\text{pm}'\text{states})):$ OK?(read_data(pm_phy, paging_data_type(lvl'))(xlat_idx(lvl', base', lin_a'))(s)) {-19} $\forall (s_1, s_2: (\text{pm}'\text{states})):$ set_reference(data(read_data(pm_phy, paging_data_type(lvl')) (xlat_idx(lvl', base', lin_a'))(s_1)), Write) = set_reference(data(read_data(pm_phy, paging_data_type(lvl')) (xlat_idx(lvl', base', lin_a'))(s_2)), Write) </pre>	<pre> {1} paging_type?(lvl', d') {2} unchanged_memory_invariant?(pm_phy' 'mem, pm_phy' 'states, singleton(expr_2_super(fatal_result)), addr_set) </pre>
--	---

Keeping (2) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of translate_unchanged_pm_phy_except_pe.1.3.5.

translate_unchanged_pm_phy_except_pe.1.4:

```

{-1} (addr_set ⊆ (pm_phy'ro_addr ∪ pm_phy'rw_addr))
{-2} is_linear_plain_memory?(pm')
{-3} plain_memory?(pm_phy)
{-4} pm' states = pm_phy states
{-5} ((pm_phy'ro_addr ∪ pm_phy'rw_addr) \ address_block(xlat_idx(lvl', base', lin_a'), expt(2, pe_size)))
    = addr_set
{-6} lvl' < max_level
{-7} Mem?(base' type_of)
{-8} 0 ≤ base' offset
{-9} base' offset < max_linear_offset
{-10} aligned?(bits_per_level[Physical_memory, pm_phy] + pe_size)(offset(base'))
{-11} Mem?(lin_a' type_of)
{-12} 0 ≤ lin_a' offset
{-13} lin_a' offset < max_linear_offset
{-14} union(pm' ro_addr, pm' rw_addr)(lin_a')
{-15} (address_block(xlat_idx(lvl', base', lin_a'), expt(2, pe_size)) ⊆ pm_phy'rw_addr)
{-16} ∀ (s: (pm' states)):
    OK?(read_data(pm_phy, paging_data_type(lvl'))(xlat_idx(lvl', base', lin_a'))(s))
{-17} ∀ (s1, s2: (pm' states)):
    set_reference(data(read_data(pm_phy, paging_data_type(lvl'))
        (xlat_idx(lvl', base', lin_a'))(s1)),
        Write)
    =
    set_reference(data(read_data(pm_phy, paging_data_type(lvl'))
        (xlat_idx(lvl', base', lin_a'))(s2)),
        Write)
-----
{1}  ∀ (pe: (range_pt(lvl'))):
    accessible?(pe, access') ∧
    privileged?(pe, priv') ∧ present?(pe) ∧ paging_type?(lvl', pe)
    ⊃ range_pt(lvl')(set_reference(pe, access'))
{2}  unchanged_memory_invariant?(pm_phy mem, pm_phy states,
    singleton(expr_2_super((read_data(pm_phy,
    pag-
    ing_data_type(lvl'))
    (xlat_idx
    (lvl', base', lin_a'))
    ##
    (λ (pe: (range_pt(lvl'))):
    IF pag-
    ing_type?(lvl', pe)
    THEN IF present?(pe)
    THEN IF ac-
    cessible?
    (pe, access')
    ∧
    priv-
    iledged?(pe, priv')
    THEN write_data
    (pm_phy,
    pag-
    ing_data_type
    (lvl'))
    (xlat_idx
    (lvl',
    base',
    lin_a')),
    1657 set_reference
    (pe, access'))
    ##
    ok_result
    (is_leaf?(pe),
    base(pe))
    ELSE raise_fault
    (priv')

```

C Proof scripts

Keeping (1) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `translate_unchanged_pm_phy_except_pe.1.4`.

`translate_unchanged_pm_phy_except_pe.2`:

<pre> {-1} is_linear_plain_memory?(pm') {-2} plain_memory?(pm_phy) {-3} pm'`states = pm_phy`states {-4} ((pm_phy`ro_addr ∪ pm_phy`rw_addr) \ address_block(xlat_idx(lvl', base', lin_a'), expt(2, pe_size) = addr_set {-5} lvl' < max_level {-6} Mem?(base'`type_of) {-7} 0 ≤ base'`offset {-8} base'`offset < max_linear_offset {-9} aligned?(bits_per_level[Physical_memory, pm_phy] + pe_size)(offset(base')) {-10} Mem?(lin_a'`type_of) {-11} 0 ≤ lin_a'`offset {-12} lin_a'`offset < max_linear_offset {-13} union(pm'`ro_addr, pm'`rw_addr)(lin_a') {-14} (address_block(xlat_idx(lvl', base', lin_a'), expt(2, pe_size)) ⊆ pm_phy`rw_addr) {-15} ∀ (s: (pm'`states)): OK?(read_data(pm_phy, paging_data_type(lvl'))(xlat_idx(lvl', base', lin_a'))(s)) {-16} ∀ (s₁, s₂: (pm'`states)): set_reference(data(read_data(pm_phy, paging_data_type(lvl')) (xlat_idx(lvl', base', lin_a'))(s₁)), Write) = set_reference(data(read_data(pm_phy, paging_data_type(lvl')) (xlat_idx(lvl', base', lin_a'))(s₂)), Write) </pre>	<pre> {1} (addr_set ⊆ (pm_phy`ro_addr ∪ pm_phy`rw_addr)) {2} unchanged_memory_invariant?(pm_phy`mem, pm_phy`states, singleton(expr_2_super(translate(lvl', base', lin_a', access', priv'))), addr_set) </pre>
--	---

Replacing using formula -4,

Rewriting using `difference_subset`, matching in *,

This completes the proof of `translate_unchanged_pm_phy_except_pe.2`.

Q.E.D.

C.117.33 Linear_Memory_Properties.translate_result_ok_TCC1

Terse proof for `translate_result_ok_TCC1`.

translate_result_ok_TCC1:

$\{1\} \quad \forall (lvl: \text{Level}, pm: \text{Plain_Memory}[\text{Linear_memory}], s_1, s_2: (\text{pm}'\text{states}),$ $\text{base: PTab_Address}[\text{Physical_memory}, pm_phy], \text{lin_a: Memory_Address_4G}):$ $\text{is_linear_plain_memory?}(pm) \wedge \text{union}(pm'\text{ro_addr}, pm'\text{rw_addr})(\text{lin_a}) \supset$ $\text{Mem?}(xlat_idx[\text{Physical_memory}, pm_phy](lvl, \text{base}, \text{lin_a})'\text{type_of}) \wedge$ $0 \leq xlat_idx[\text{Physical_memory}, pm_phy](lvl, \text{base}, \text{lin_a})'\text{offset} \wedge$ $xlat_idx[\text{Physical_memory}, pm_phy](lvl, \text{base}, \text{lin_a})'\text{offset} < \text{max_linear_offset}$

Repeatedly Skolemizing and flattening,
 Installing automatic rewrites from: (Mem! max_linear! <!)
 Using lemma xlat_idx_memory_address,
 Using lemma xlat_idx_memory_address2,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 This completes the proof of translate_result_ok_TCC1.
 Q.E.D.

C.117.34 Linear_Memory_Properties.translate_result_ok

Terse proof for translate_result_ok.

translate_result_ok:

$\{1\} \quad \forall (lvl: \text{Level}, pm: \text{Plain_Memory}[\text{Linear_memory}], s_1, s_2: (\text{pm}'\text{states}), \text{base: PTab_Address},$ $\text{lin_a: Memory_Address_4G}, \text{access: Memory_access}, \text{priv: Memory_privilege}):$ $\text{union}(pm'\text{ro_addr}, pm'\text{rw_addr})(\text{lin_a}) \wedge$ $\text{is_linear_plain_memory?}(pm) \wedge$ $\text{pe_in_pt_range?}(s_1,$ $\quad \text{restrict}[\text{Address}, \text{Memory_Address_4G}, \text{boolean}]$ $\quad \quad ((pm'\text{ro_addr} \cup pm'\text{rw_addr}),$ $\quad \quad \text{lvl})$ $\quad \quad (\text{xlat_idx}(lvl, \text{base}, \text{lin_a}))$ $\quad \wedge \text{OK?}(\text{translate}(lvl, \text{base}, \text{lin_a}, \text{access}, \text{priv})(s_1))$ $\quad \supset \text{OK?}(\text{translate}(lvl, \text{base}, \text{lin_a}, \text{access}, \text{priv})(s_2))$
--

Repeatedly Skolemizing and flattening,
 Expanding the definition of translate,
 Expanding the definition of is_linear_plain_memory?,
 Applying disjunctive simplification to flatten sequent,
 Hiding formulas: (-13 -14 -15 -16),
 Expanding the definition of ##,
 Expanding the definition of raise_fault,
 Expanding the definition of exception_result,
 Expanding the definition of fatal_result,
 Expanding the definition of ##,
 Expanding the definition of ##,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 Instantiating the top quantifier in -15 with the terms: (s1!1 s2!1 lvl!1 xlat_idx(lvl!1, base!1, lin_a!1)),
 Instantiating the top quantifier in -13 with the terms: (s1!1 s2!1),
 Applying disjunctive simplification to flatten sequent,
 Instantiating the top quantifier in -15 with the terms: (s2!1 lvl!1 xlat_idx(lvl!1, base!1, lin_a!1)),
 we get 2 subgoals:

```
translate_result_ok.1:
```

```

{-1}  lvl' < max_level
{-2}  pm' 'states(s'_1)
{-3}  pm' 'states(s'_2)
{-4}  Mem?(base' 'type_of)
{-5}  0 ≤ base' 'offset
{-6}  base' 'offset < max_linear_offset
{-7}  aligned?(bits_per_level[Physical_memory, pm_phy] + pe_size)(offset(base'))
{-8}  Mem?(lin_a' 'type_of)
{-9}  0 ≤ lin_a' 'offset
{-10} lin_a' 'offset < max_linear_offset
{-11} union(pm' 'ro_addr, pm' 'rw_addr)(lin_a')
{-12} pm' 'mem = linear_pm
{-13} pe_in_pdir_range?(s'_1,
                                restrict[Address, Memory_Address_4G, boolean]
                                  ((pm' 'ro_addr ∪ pm' 'rw_addr)))
    =
    pe_in_pdir_range?(s'_2,
                                restrict[Address, Memory_Address_4G, boolean]
                                  ((pm' 'ro_addr ∪ pm' 'rw_addr)))
{-14} pe_in_ptab_range?(s'_1,
                                restrict[Address, Memory_Address_4G, boolean]
                                  ((pm' 'ro_addr ∪ pm' 'rw_addr)))
    =
    pe_in_ptab_range?(s'_2,
                                restrict[Address, Memory_Address_4G, boolean]
                                  ((pm' 'ro_addr ∪ pm' 'rw_addr)))
{-15} OK?(read_data(pm_phy, paging_data_type(lvl'))(xlat_idx(lvl', base', lin_a'))(s'_2))
{-16} set_reference(data(read_data(pm_phy, paging_data_type(lvl'))
                                (xlat_idx(lvl', base', lin_a'))(s'_1)),
                    Write)
    =
    set_reference(data(read_data(pm_phy, paging_data_type(lvl'))
                                (xlat_idx(lvl', base', lin_a'))(s'_2)),
                    Write)
{-17} plain_memory?(pm_phy)
{-18} pm' 'states = pm_phy 'states
{-19} (pm' 'other_actions ⊆ pm_phy 'other_actions)
{-20} (linear_resolve_register_transformers ⊆ pm_phy 'other_actions)
{-21} ∀ (a: ((pm' 'ro_addr ∪ pm' 'rw_addr))):
    Mem?(type_of(a)) ∧ 0 ≤ offset(a) ∧ offset(a) < max_linear_offset
{-22} ∀ (s: (pm' 'states)):
    linear_blessed?(s, restrict[Address, Memory_Address_4G, boolean](pm' 'ro_addr),
                    restrict[Address, Memory_Address_4G, boolean](pm' 'rw_addr))
{-23} pe_in_pt_range?(s'_1,
                                restrict[Address, Memory_Address_4G, boolean]
                                  ((pm' 'ro_addr ∪ pm' 'rw_addr)),
                                lvl')
                                (xlat_idx(lvl', base', lin_a'))
{-24} OK?(read_data(pm_phy, paging_data_type(lvl'))(xlat_idx(lvl', base', lin_a'))(s'_1))
{-25} paging_type?(lvl',
                                data(read_data(pm_phy, paging_data_type(lvl'))
                                        (xlat_idx(lvl', base', lin_a'))(s'_1)))
{-26} present?(data(read_data(pm_phy, paging_data_type(lvl'))(xlat_idx(lvl', base', lin_a'))
                                (s'_1)))
{-27} accessible?(data(read_data(pm_phy, paging_data_type(lvl'))
                                (xlat_idx(lvl', base', lin_a'))(s'_1)),
                    access')
1660
{-28} privileged?(data(read_data(pm_phy, paging_data_type(lvl'))
                                (xlat_idx(lvl', base', lin_a'))(s'_1)),
                    priv')
{-29} OK?((write_data(pm_phy, paging_data_type(lvl'))
                                (xlat_idx(lvl', base', lin_a')),
                    set_reference(data(read_data(pm_phy, paging_data_type(lvl'))
                                (xlat_idx(lvl', base', lin_a'))(s'_1))

```

Expanding the definition of ##,

Using lemma set_reference_present,

Using lemma set_reference_base,

Using lemma set_reference_accessible,

Using lemma set_reference_privileged,

Using lemma set_reference_leaf,

Using lemma set_reference_paging_type,

Case splitting on $OK?(write_data(pm_phy, paging_data_type(lvl!1)) (xlat_idx(lvl!1, base!1, lin_a!1), set_reference(data(read_data (pm_phy, paging_data_type(lvl!1)) (xlat_idx(lvl!1, base!1, lin_a!1)) (s2!1)), access!1)) (state(read_data(pm_phy, paging_data_type(lvl!1)) (xlat_idx(lvl!1, base!1, lin_a!1)) (s2!1))))$,

we get 4 subgoals:

translate_result_ok.1.1:

```

{-1} OK?(write_data(pm_phy, paging_data_type(lvl')
                (xlat_idx(lvl', base', lin_a'),
                 set_reference(data(read_data(pm_phy, paging_data_type(lvl')
                                (xlat_idx(lvl', base', lin_a'))(s'_2)),
                                access'))
                (state(read_data(pm_phy, paging_data_type(lvl')
                                (xlat_idx(lvl', base', lin_a'))(s'_2))))))
{-2} set_reference(data(read_data(pm_phy, paging_data_type(lvl')
                (xlat_idx(lvl', base', lin_a'))(s'_1)),
                Write)
    =
    set_reference(data(read_data(pm_phy, paging_data_type(lvl')
                (xlat_idx(lvl', base', lin_a'))(s'_2)),
                Write)
    ⊃
    paging_type?(lvl',
                data(read_data(pm_phy, paging_data_type(lvl')
                (xlat_idx(lvl', base', lin_a'))(s'_1)))
    =
    paging_type?(lvl',
                data(read_data(pm_phy, paging_data_type(lvl')
                (xlat_idx(lvl', base', lin_a'))(s'_2)))
{-3} set_reference(data(read_data(pm_phy, paging_data_type(lvl')
                (xlat_idx(lvl', base', lin_a'))(s'_1)),
                Write)
    =
    set_reference(data(read_data(pm_phy, paging_data_type(lvl')
                (xlat_idx(lvl', base', lin_a'))(s'_2)),
                Write)
    ∧
    present?(data(read_data(pm_phy, paging_data_type(lvl')
                (xlat_idx(lvl', base', lin_a'))(s'_1)))
    ⊃
    is_leaf?(data(read_data(pm_phy, paging_data_type(lvl')
                (xlat_idx(lvl', base', lin_a'))(s'_1)))
    =
    is_leaf?(data(read_data(pm_phy, paging_data_type(lvl')
                (xlat_idx(lvl', base', lin_a'))(s'_2)))
{-4} set_reference(data(read_data(pm_phy, paging_data_type(lvl')
                (xlat_idx(lvl', base', lin_a'))(s'_1)),
                Write)
    =
    set_reference(data(read_data(pm_phy, paging_data_type(lvl')
                (xlat_idx(lvl', base', lin_a'))(s'_2)),
                Write)
    ∧
    present?(data(read_data(pm_phy, paging_data_type(lvl')
                (xlat_idx(lvl', base', lin_a'))(s'_1)))
    ⊃
    privileged?(data(read_data(pm_phy, paging_data_type(lvl')
                (xlat_idx(lvl', base', lin_a'))(s'_1)),
                priv')
    =
    privileged?(data(read_data(pm_phy, paging_data_type(lvl')
                (xlat_idx(lvl', base', lin_a'))(s'_2)),
                priv')
1662 {-5} set_reference(data(read_data(pm_phy, paging_data_type(lvl')
                (xlat_idx(lvl', base', lin_a'))(s'_1)),
                Write)
    =
    set_reference(data(read_data(pm_phy, paging_data_type(lvl')
                (xlat_idx(lvl', base', lin_a'))(s'_2)),
                Write)

```

Installing automatic rewrites from: ## ok_result

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `translate_result_ok.1.1`.

translate_result_ok.1.2:

```

{-1} set_reference(data(read_data(pm_phy, paging_data_type(lvl'))
                        (xlat_idx(lvl', base', lin_a'))(s'_1)),
                  Write)
=
set_reference(data(read_data(pm_phy, paging_data_type(lvl'))
                  (xlat_idx(lvl', base', lin_a'))(s'_2)),
              Write)
⊃
paging_type?(lvl',
              data(read_data(pm_phy, paging_data_type(lvl'))
                  (xlat_idx(lvl', base', lin_a'))(s'_1)))
=
paging_type?(lvl',
              data(read_data(pm_phy, paging_data_type(lvl'))
                  (xlat_idx(lvl', base', lin_a'))(s'_2)))
{-2} set_reference(data(read_data(pm_phy, paging_data_type(lvl'))
                        (xlat_idx(lvl', base', lin_a'))(s'_1)),
                  Write)
=
set_reference(data(read_data(pm_phy, paging_data_type(lvl'))
                  (xlat_idx(lvl', base', lin_a'))(s'_2)),
              Write)
^
present?(data(read_data(pm_phy, paging_data_type(lvl'))
              (xlat_idx(lvl', base', lin_a'))(s'_1)))
⊃
is_leaf?(data(read_data(pm_phy, paging_data_type(lvl'))
              (xlat_idx(lvl', base', lin_a'))(s'_1)))
=
is_leaf?(data(read_data(pm_phy, paging_data_type(lvl'))
              (xlat_idx(lvl', base', lin_a'))(s'_2)))
{-3} set_reference(data(read_data(pm_phy, paging_data_type(lvl'))
                        (xlat_idx(lvl', base', lin_a'))(s'_1)),
                  Write)
=
set_reference(data(read_data(pm_phy, paging_data_type(lvl'))
                  (xlat_idx(lvl', base', lin_a'))(s'_2)),
              Write)
^
present?(data(read_data(pm_phy, paging_data_type(lvl'))
              (xlat_idx(lvl', base', lin_a'))(s'_1)))
⊃
privileged?(data(read_data(pm_phy, paging_data_type(lvl'))
                  (xlat_idx(lvl', base', lin_a'))(s'_1)),
             priv')
=
privileged?(data(read_data(pm_phy, paging_data_type(lvl'))
                  (xlat_idx(lvl', base', lin_a'))(s'_2)),
             priv')
{-4} set_reference(data(read_data(pm_phy, paging_data_type(lvl'))
                        (xlat_idx(lvl', base', lin_a'))(s'_1)),
                  Write)
=
set_reference(data(read_data(pm_phy, paging_data_type(lvl'))
                  (xlat_idx(lvl', base', lin_a'))(s'_2)),
              Write)
^
present?(data(read_data(pm_phy, paging_data_type(lvl'))
              (xlat_idx(lvl', base', lin_a'))(s'_1)))
⊃
accessible?(data(read_data(pm_phy, paging_data_type(lvl'))
                  (xlat_idx(lvl', base', lin_a'))(s'_1)),
            access')

```

Hiding formulas: 2,

Using lemma plain_memory_write_data_ok,

Case splitting on `in_blessed_memory?(paging_data_type(lvl!1), xlat_idx(lvl!1, base!1, lin_a!1), pm_phy'rw_addr)`,

we get 2 subgoals:

translate_result_ok.1.2.1:

```

{-1}  in_blessed_memory?(paging_data_type(lvl'), xlat_idx(lvl', base', lin_a'), pm_phy'rw_addr)
{-2}  plain_memory?(pm_phy) ^
      pm_phy'states
      (state(read_data(pm_phy, paging_data_type(lvl'))(xlat_idx(lvl', base', lin_a'))
              (s'_2)))

      ^
      in_blessed_memory?(paging_data_type(lvl'), xlat_idx(lvl', base', lin_a'),
                          pm_phy'rw_addr)
      ⊃
      OK?(write_data(pm_phy, paging_data_type(lvl'))
           (xlat_idx(lvl', base', lin_a'),
            set_reference(data(read_data(pm_phy, paging_data_type(lvl'))
                              (xlat_idx(lvl', base', lin_a'))(s'_2)),
                          access'))
           (state(read_data(pm_phy, paging_data_type(lvl'))
                   (xlat_idx(lvl', base', lin_a'))(s'_2))))))
{-3}  set_reference(data(read_data(pm_phy, paging_data_type(lvl'))
                          (xlat_idx(lvl', base', lin_a'))(s'_1)),
                    Write)
      =
      set_reference(data(read_data(pm_phy, paging_data_type(lvl'))
                          (xlat_idx(lvl', base', lin_a'))(s'_2)),
                    Write)
      ⊃
      paging_type?(lvl',
                    data(read_data(pm_phy, paging_data_type(lvl'))
                          (xlat_idx(lvl', base', lin_a'))(s'_1)))
      =
      paging_type?(lvl',
                    data(read_data(pm_phy, paging_data_type(lvl'))
                          (xlat_idx(lvl', base', lin_a'))(s'_2)))
{-4}  set_reference(data(read_data(pm_phy, paging_data_type(lvl'))
                          (xlat_idx(lvl', base', lin_a'))(s'_1)),
                    Write)
      =
      set_reference(data(read_data(pm_phy, paging_data_type(lvl'))
                          (xlat_idx(lvl', base', lin_a'))(s'_2)),
                    Write)
      ^
      present?(data(read_data(pm_phy, paging_data_type(lvl'))
                     (xlat_idx(lvl', base', lin_a'))(s'_1)))
      ⊃
      is_leaf?(data(read_data(pm_phy, paging_data_type(lvl'))
                    (xlat_idx(lvl', base', lin_a'))(s'_1)))
      =
      is_leaf?(data(read_data(pm_phy, paging_data_type(lvl'))
                    (xlat_idx(lvl', base', lin_a'))(s'_2)))
{-5}  set_reference(data(read_data(pm_phy, paging_data_type(lvl'))
                          (xlat_idx(lvl', base', lin_a'))(s'_1)),
                    Write)
      =
      set_reference(data(read_data(pm_phy, paging_data_type(lvl'))
                          (xlat_idx(lvl', base', lin_a'))(s'_2)),
                    Write)
      ^
1666  present?(data(read_data(pm_phy, paging_data_type(lvl'))
                   (xlat_idx(lvl', base', lin_a'))(s'_1)))
      ⊃
      privileged?(data(read_data(pm_phy, paging_data_type(lvl'))
                       (xlat_idx(lvl', base', lin_a'))(s'_1)),
                  priv')
      =
      privileged?(data(read_data(pm_phy, paging_data_type(lvl'))

```

C.117 Proofs for Linear_Memory_Properties (challenge-linear.pvs)

Using lemma `plain_memory_transformer_invariant_read_data`,

Using lemma `expr_transformer_invariant_next_ok`,

Expanding the definition of `singleton`,

Installing automatic rewrites from: `has_next_state in_blessed_memory_rw_ro`

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `translate_result_ok.1.2.1`.

translate_result_ok.1.2.2:

```

{-1} plain_memory?(pm_phy) ∧
      pm_phy'states
        (state(read_data(pm_phy, paging_data_type(lvl'))(xlat_idx(lvl', base', lin_a'))
                (s'_2)))
      ∧
      in_blessed_memory?(paging_data_type(lvl'), xlat_idx(lvl', base', lin_a'),
                          pm_phy'rw_addr)
    ⊃
    OK?(write_data(pm_phy, paging_data_type(lvl'))
         (xlat_idx(lvl', base', lin_a'),
          set_reference(data(read_data(pm_phy, paging_data_type(lvl'))
                             (xlat_idx(lvl', base', lin_a'))(s'_2)),
                        access'))
         (state(read_data(pm_phy, paging_data_type(lvl'))
                (xlat_idx(lvl', base', lin_a'))(s'_2))))))
{-2} set_reference(data(read_data(pm_phy, paging_data_type(lvl'))
                        (xlat_idx(lvl', base', lin_a'))(s'_1)),
                  Write)
      =
      set_reference(data(read_data(pm_phy, paging_data_type(lvl'))
                        (xlat_idx(lvl', base', lin_a'))(s'_2)),
                  Write)
    ⊃
      paging_type?(lvl',
                  data(read_data(pm_phy, paging_data_type(lvl'))
                        (xlat_idx(lvl', base', lin_a'))(s'_1)))
      =
      paging_type?(lvl',
                  data(read_data(pm_phy, paging_data_type(lvl'))
                        (xlat_idx(lvl', base', lin_a'))(s'_2)))
{-3} set_reference(data(read_data(pm_phy, paging_data_type(lvl'))
                        (xlat_idx(lvl', base', lin_a'))(s'_1)),
                  Write)
      =
      set_reference(data(read_data(pm_phy, paging_data_type(lvl'))
                        (xlat_idx(lvl', base', lin_a'))(s'_2)),
                  Write)
      ∧
      present?(data(read_data(pm_phy, paging_data_type(lvl'))
                    (xlat_idx(lvl', base', lin_a'))(s'_1)))
    ⊃
      is_leaf?(data(read_data(pm_phy, paging_data_type(lvl'))
                    (xlat_idx(lvl', base', lin_a'))(s'_1)))
      =
      is_leaf?(data(read_data(pm_phy, paging_data_type(lvl'))
                    (xlat_idx(lvl', base', lin_a'))(s'_2)))
{-4} set_reference(data(read_data(pm_phy, paging_data_type(lvl'))
                        (xlat_idx(lvl', base', lin_a'))(s'_1)),
                  Write)
      =
      set_reference(data(read_data(pm_phy, paging_data_type(lvl'))
                        (xlat_idx(lvl', base', lin_a'))(s'_2)),
                  Write)
      ∧
      present?(data(read_data(pm_phy, paging_data_type(lvl'))
                    (xlat_idx(lvl', base', lin_a'))(s'_1)))
    ⊃
      privileged?(data(read_data(pm_phy, paging_data_type(lvl'))
                       (xlat_idx(lvl', base', lin_a'))(s'_1)),
                  priv')
      =
      privileged?(data(read_data(pm_phy, paging_data_type(lvl'))
                       (xlat_idx(lvl', base', lin_a'))(s'_2)),

```

C.117 Proofs for Linear_Memory_Properties (challenge-linear.pvs)

Expanding the definition of `in_blessed_memory?`,

Rewriting using `paging_data_type_size`, matching in `*`,

Hiding formulas: `(-1 -2 -3 -4 -5 -6 -7 2)`,

Using lemma `pe_in_pt_address_in_pt`,

Using lemma `subset_ptab_address_rw_addr`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Instantiating the top quantifier in `-24` with the terms: `(s1!1)`,

Expanding the definition of `linear_blessed?`,

which is trivially true.

This completes the proof of `translate_result_ok.1.2.2`.

translate_result_ok.1.3:

```

{-1} set_reference(data(read_data(pm_phy, paging_data_type(lvl'))
                        (xlat_idx(lvl', base', lin_a'))(s'_1)),
                  Write)
=
set_reference(data(read_data(pm_phy, paging_data_type(lvl'))
                  (xlat_idx(lvl', base', lin_a'))(s'_2)),
              Write)
⊃
paging_type?(lvl',
              data(read_data(pm_phy, paging_data_type(lvl'))
                  (xlat_idx(lvl', base', lin_a'))(s'_1)))
=
paging_type?(lvl',
              data(read_data(pm_phy, paging_data_type(lvl'))
                  (xlat_idx(lvl', base', lin_a'))(s'_2)))
{-2} set_reference(data(read_data(pm_phy, paging_data_type(lvl'))
                        (xlat_idx(lvl', base', lin_a'))(s'_1)),
                  Write)
=
set_reference(data(read_data(pm_phy, paging_data_type(lvl'))
                  (xlat_idx(lvl', base', lin_a'))(s'_2)),
              Write)
^
present?(data(read_data(pm_phy, paging_data_type(lvl'))
              (xlat_idx(lvl', base', lin_a'))(s'_1)))
⊃
is_leaf?(data(read_data(pm_phy, paging_data_type(lvl'))
              (xlat_idx(lvl', base', lin_a'))(s'_1)))
=
is_leaf?(data(read_data(pm_phy, paging_data_type(lvl'))
              (xlat_idx(lvl', base', lin_a'))(s'_2)))
{-3} set_reference(data(read_data(pm_phy, paging_data_type(lvl'))
                        (xlat_idx(lvl', base', lin_a'))(s'_1)),
                  Write)
=
set_reference(data(read_data(pm_phy, paging_data_type(lvl'))
                  (xlat_idx(lvl', base', lin_a'))(s'_2)),
              Write)
^
present?(data(read_data(pm_phy, paging_data_type(lvl'))
              (xlat_idx(lvl', base', lin_a'))(s'_1)))
⊃
privileged?(data(read_data(pm_phy, paging_data_type(lvl'))
                  (xlat_idx(lvl', base', lin_a'))(s'_1)),
             priv')
=
privileged?(data(read_data(pm_phy, paging_data_type(lvl'))
                  (xlat_idx(lvl', base', lin_a'))(s'_2)),
             priv')
{-4} set_reference(data(read_data(pm_phy, paging_data_type(lvl'))
                        (xlat_idx(lvl', base', lin_a'))(s'_1)),
                  Write)
=
set_reference(data(read_data(pm_phy, paging_data_type(lvl'))
                  (xlat_idx(lvl', base', lin_a'))(s'_2)),
              Write)
^
present?(data(read_data(pm_phy, paging_data_type(lvl'))
              (xlat_idx(lvl', base', lin_a'))(s'_1)))
⊃
accessible?(data(read_data(pm_phy, paging_data_type(lvl'))
                  (xlat_idx(lvl', base', lin_a'))(s'_1)),
            access')

```

C.117 Proofs for Linear_Memory_Properties (challenge-linear.pvs)

Rewriting using `set_reference_range`, matching in `*`,

This completes the proof of `translate_result_ok.1.3`.

translate_result_ok.1.4:

```

{-1} set_reference(data(read_data(pm_phy, paging_data_type(lvl'))
                        (xlat_idx(lvl', base', lin_a'))(s'_1)),
                  Write)
=
set_reference(data(read_data(pm_phy, paging_data_type(lvl'))
                  (xlat_idx(lvl', base', lin_a'))(s'_2)),
              Write)
⊃
paging_type?(lvl',
              data(read_data(pm_phy, paging_data_type(lvl'))
                  (xlat_idx(lvl', base', lin_a'))(s'_1)))
=
paging_type?(lvl',
              data(read_data(pm_phy, paging_data_type(lvl'))
                  (xlat_idx(lvl', base', lin_a'))(s'_2)))
{-2} set_reference(data(read_data(pm_phy, paging_data_type(lvl'))
                        (xlat_idx(lvl', base', lin_a'))(s'_1)),
                  Write)
=
set_reference(data(read_data(pm_phy, paging_data_type(lvl'))
                  (xlat_idx(lvl', base', lin_a'))(s'_2)),
              Write)
^
present?(data(read_data(pm_phy, paging_data_type(lvl'))
              (xlat_idx(lvl', base', lin_a'))(s'_1)))
⊃
is_leaf?(data(read_data(pm_phy, paging_data_type(lvl'))
              (xlat_idx(lvl', base', lin_a'))(s'_1)))
=
is_leaf?(data(read_data(pm_phy, paging_data_type(lvl'))
              (xlat_idx(lvl', base', lin_a'))(s'_2)))
{-3} set_reference(data(read_data(pm_phy, paging_data_type(lvl'))
                        (xlat_idx(lvl', base', lin_a'))(s'_1)),
                  Write)
=
set_reference(data(read_data(pm_phy, paging_data_type(lvl'))
                  (xlat_idx(lvl', base', lin_a'))(s'_2)),
              Write)
^
present?(data(read_data(pm_phy, paging_data_type(lvl'))
              (xlat_idx(lvl', base', lin_a'))(s'_1)))
⊃
privileged?(data(read_data(pm_phy, paging_data_type(lvl'))
                 (xlat_idx(lvl', base', lin_a'))(s'_1)),
             priv')
=
privileged?(data(read_data(pm_phy, paging_data_type(lvl'))
                 (xlat_idx(lvl', base', lin_a'))(s'_2)),
             priv')
{-4} set_reference(data(read_data(pm_phy, paging_data_type(lvl'))
                        (xlat_idx(lvl', base', lin_a'))(s'_1)),
                  Write)
=
set_reference(data(read_data(pm_phy, paging_data_type(lvl'))
                  (xlat_idx(lvl', base', lin_a'))(s'_2)),
              Write)
^
present?(data(read_data(pm_phy, paging_data_type(lvl'))
              (xlat_idx(lvl', base', lin_a'))(s'_1)))
⊃
accessible?(data(read_data(pm_phy, paging_data_type(lvl'))
                 (xlat_idx(lvl', base', lin_a'))(s'_1)),
            access')

```

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `translate_result_ok.1.4`.

translate_result_ok.2:

```

{-1}  lvl' < max_level
{-2}  pm' 'states(s'_1)
{-3}  pm' 'states(s'_2)
{-4}  Mem?(base' 'type_of)
{-5}  0 ≤ base' 'offset
{-6}  base' 'offset < max_linear_offset
{-7}  aligned?(bits_per_level[Physical_memory, pm_phy] + pe_size)(offset(base'))
{-8}  Mem?(lin_a' 'type_of)
{-9}  0 ≤ lin_a' 'offset
{-10} lin_a' 'offset < max_linear_offset
{-11} union(pm' 'ro_addr, pm' 'rw_addr)(lin_a')
{-12} pm' 'mem = linear_pm
{-13} pe_in_pdir_range?(s'_1,
                                restrict[Address, Memory_Address_4G, boolean]
                                  ((pm' 'ro_addr ∪ pm' 'rw_addr)))
    =
    pe_in_pdir_range?(s'_2,
                                restrict[Address, Memory_Address_4G, boolean]
                                  ((pm' 'ro_addr ∪ pm' 'rw_addr)))
{-14} pe_in_ptab_range?(s'_1,
                                restrict[Address, Memory_Address_4G, boolean]
                                  ((pm' 'ro_addr ∪ pm' 'rw_addr)))
    =
    pe_in_ptab_range?(s'_2,
                                restrict[Address, Memory_Address_4G, boolean]
                                  ((pm' 'ro_addr ∪ pm' 'rw_addr)))
{-15} set_reference(data(read_data(pm_phy, paging_data_type(lvl'))
                                (xlat_idx(lvl', base', lin_a'))(s'_1)),
                    Write)
    =
    set_reference(data(read_data(pm_phy, paging_data_type(lvl'))
                                (xlat_idx(lvl', base', lin_a'))(s'_2)),
                    Write)
{-16} plain_memory?(pm_phy)
{-17} pm' 'states = pm_phy 'states
{-18} (pm' 'other_actions ⊆ pm_phy 'other_actions)
{-19} (linear_resolve_register_transformers ⊆ pm_phy 'other_actions)
{-20} ∀ (a: ((pm' 'ro_addr ∪ pm' 'rw_addr))):
    Mem?(type_of(a)) ∧ 0 ≤ offset(a) ∧ offset(a) < max_linear_offset
{-21} ∀ (s: (pm' 'states)):
    linear_blessed?(s, restrict[Address, Memory_Address_4G, boolean](pm' 'ro_addr),
                    restrict[Address, Memory_Address_4G, boolean](pm' 'rw_addr))
{-22} pe_in_pt_range?(s'_1,
                                restrict[Address, Memory_Address_4G, boolean]
                                  ((pm' 'ro_addr ∪ pm' 'rw_addr)),
                                lvl')
                                (xlat_idx(lvl', base', lin_a'))
{-23} OK?(read_data(pm_phy, paging_data_type(lvl'))(xlat_idx(lvl', base', lin_a'))(s'_1))
{-24} paging_type?(lvl',
                    data(read_data(pm_phy, paging_data_type(lvl'))
                                (xlat_idx(lvl', base', lin_a'))(s'_1)))
{-25} present?(data(read_data(pm_phy, paging_data_type(lvl'))(xlat_idx(lvl', base', lin_a'))
                    (s'_1)))
{-26} accessible?(data(read_data(pm_phy, paging_data_type(lvl'))
                    (xlat_idx(lvl', base', lin_a'))(s'_1)),
                    access')
1674 {-27} privileged?(data(read_data(pm_phy, paging_data_type(lvl'))
                    (xlat_idx(lvl', base', lin_a'))(s'_1)),
                    priv')
{-28} OK?((write_data(pm_phy, paging_data_type(lvl'))
                    (xlat_idx(lvl', base', lin_a')),
                    set_reference(data(read_data(pm_phy, paging_data_type(lvl'))
                    (xlat_idx(lvl', base', lin_a'))(s'_1)),
                    access'))

```

Expanding the definition of `pe_in_pt_range?`,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 This completes the proof of `translate_result_ok.2`.
 Q.E.D.

C.117.35 Linear_Memory_Properties.translate_result_as_read_TCC1

Terse proof for `translate_result_as_read_TCC1`.

`translate_result_as_read_TCC1`:

```
{1}  ∀ (lvl: Level, pm: Plain_Memory[Linear_memory], s: Lin-
      ear_memory[Physical_memory, pm_phy],
      base: PTab_Address[Physical_memory, pm_phy], lin_a: Memory_Address_4G,
      access: Memory_access, priv: Memory_privilege):
      union(pm'ro_addr, pm'rw_addr)(lin_a) ∧
      is_linear_plain_memory?(pm) ∧ OK?(translate(lvl, base, lin_a, access, priv)(s))
      ⊃
      (∀ (res: ExprResult[Physical_memory, ((range_pt(lvl))))):
       res = read_data(pm_phy, paging_data_type(lvl))(xlat_idx(lvl, base, lin_a))(s) ⊃
        OK?[Physical_memory, ((range_pt(lvl)))](res))
```

Repeatedly Skolemizing and flattening,
 Expanding the definition of `translate`,
 Expanding the definition of `##`,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 This completes the proof of `translate_result_as_read_TCC1`.
 Q.E.D.

C.117.36 Linear_Memory_Properties.translate_result_as_read_TCC2

Terse proof for `translate_result_as_read_TCC2`.

`translate_result_as_read_TCC2`:

```
{1}  ∀ (lvl: Level, pm: Plain_Memory[Linear_memory], s: Lin-
      ear_memory[Physical_memory, pm_phy],
      base: PTab_Address[Physical_memory, pm_phy], lin_a: Memory_Address_4G,
      access: Memory_access, priv: Memory_privilege):
      union(pm'ro_addr, pm'rw_addr)(lin_a) ∧
      is_linear_plain_memory?(pm) ∧ OK?(translate(lvl, base, lin_a, access, priv)(s))
      ⊃
      (∀ (res: ExprResult[Physical_memory, ((range_pt(lvl))))):
       res = read_data(pm_phy, paging_data_type(lvl))(xlat_idx(lvl, base, lin_a))(s) ⊃
        present?(data[Physical_memory, ((range_pt(lvl)))](res)))
```

Repeatedly Skolemizing and flattening,
 Hiding formulas: -10,
 Expanding the definition of `translate`,
 Installing automatic rewrites from: (##! raise_fault! fatal_result! exception_result!)
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 This completes the proof of `translate_result_as_read_TCC2`.
 Q.E.D.

C.117.37 Linear_Memory_Properties.translate_result_as_readTerse proof for `translate_result_as_read`.`translate_result_as_read:`

```
{1}  ∃ (lvl: Level, pm: Plain_Memory[Linear_memory], s: Linear_memory[Physical_memory, pm_phy],
      base: PTab_Address, lin_a: Memory_Address_4G, access: Memory_access, priv: Memory_privilege):
  union(pm'ro_addr, pm'rw_addr)(lin_a) ∧
  is_linear_plain_memory?(pm) ∧ OK?(translate(lvl, base, lin_a, access, priv)(s))
  ⊃
  LET res = read_data(pm_phy, paging_data_type(lvl))(xlat_idx(lvl, base, lin_a))(s) IN
  data(translate(lvl, base, lin_a, access, priv)(s)) =
  (is_leaf?(data(res)), base(data(res)))
```

Repeatedly Skolemizing and flattening,

Expanding the definition of `translate`,Installing automatic rewrites from: (###! expr_2_super! expr_2_super_res! ok_result! fatal_result!
raise_fault! exception_result!)

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `translate_result_as_read`.

Q.E.D.

C.117.38 Linear_Memory_Properties.translate_same_resultTerse proof for `translate_same_result`.`translate_same_result:`

```
{1}  ∃ (lvl: Level, pm: Plain_Memory[Linear_memory], s1, s2: (pm'states), base: PTab_Address,
      lin_a: Memory_Address_4G, ac1, ac2: Memory_access, priv: Memory_privilege):
  union(pm'ro_addr, pm'rw_addr)(lin_a) ∧
  is_linear_plain_memory?(pm) ∧
  pe_in_pt_range?(s1,
                  restrict[Address, Memory_Address_4G, boolean]
                    ((pm'ro_addr ∪ pm'rw_addr),
                     lvl)
                    (xlat_idx(lvl, base, lin_a)))
  ∧
  OK?(translate(lvl, base, lin_a, ac1, priv)(s1)) ∧
  OK?(translate(lvl, base, lin_a, ac2, priv)(s2))
  ⊃
  data(translate(lvl, base, lin_a, ac1, priv)(s1)) =
  data(translate(lvl, base, lin_a, ac2, priv)(s2))
```

Repeatedly Skolemizing and flattening,

Using lemma `translate_result_as_read`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Using lemma `translate_result_as_read`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Expanding the definition of `is_linear_plain_memory?`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Instantiating the top quantifier in -9 with the terms: (s1!1 s2!1 lvl!1 xlat_idx(lvl!1, base!1, lin_a!1)),
 Using lemma set_reference_leaf,
 Using lemma set_reference_base,
 Using lemma set_reference_present,
 Keeping (-1 -2 -3 -12 -19 -20 -21 -22 1) and hiding *,
 Replacing using formula -6,
 Replacing using formula -8,
 Hiding formulas: (-6 -8),
 Expanding the definition of translate,
 Installing automatic rewrites from: (##! expr_2_super! raise_fault! expr_2_super_res! excep-
 tion_result! ok_result! fatal_result!)
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 This completes the proof of translate_same_result.
 Q.E.D.

C.117.39 Linear_Memory_Properties.xlat_pe_in_pdir_range_TCC1

Terse proof for xlat_pe_in_pdir_range_TCC1.

xlat_pe_in_pdir_range_TCC1:

$\{1\} \quad \forall (pm: \text{Plain_Memory}[\text{Linear_memory}], \text{addr}: \text{Memory_Address_4G},$ $s_1, s_2: \text{Linear_memory}[\text{Physical_memory}, pm_phy]):$ $\text{union}(pm'ro_addr, pm'rw_addr)(\text{addr}) \wedge$ $\text{is_linear_plain_memory?}(pm) \wedge pm'states(s_1) \wedge pm'states(s_2)$ $\supset \text{pdir_lvl}[\text{Physical_memory}, pm_phy] < \text{max_level}$
--

Repeatedly Skolemizing and flattening,
 Keeping (1) and hiding *,
 Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of xlat_pe_in_pdir_range_TCC1.
 Q.E.D.

C.117.40 Linear_Memory_Properties.xlat_pe_in_pdir_range_TCC2

Terse proof for xlat_pe_in_pdir_range_TCC2.

xlat_pe_in_pdir_range_TCC2:

$\{1\} \quad \forall (pm: \text{Plain_Memory}[\text{Linear_memory}], \text{addr}: \text{Memory_Address_4G},$ $s_1, s_2: \text{Linear_memory}[\text{Physical_memory}, pm_phy]):$ $\text{union}(pm'ro_addr, pm'rw_addr)(\text{addr}) \wedge$ $\text{is_linear_plain_memory?}(pm) \wedge pm'states(s_1) \wedge pm'states(s_2)$ \supset $\text{OK?}[\text{Physical_memory}, \text{Pdbr_type}]$ $(\text{read_data}[\text{Physical_memory}, \text{Pdbr_type}](pm_phy, \text{pdbr_data_type})(\text{PDBR})(s_2))$
--

Repeatedly Skolemizing and flattening,
 Expanding the definition of is_linear_plain_memory?,
 Applying disjunctive simplification to flatten sequent,
 Instantiating quantified variables,
 Instantiating quantified variables,
 This completes the proof of xlat_pe_in_pdir_range_TCC2.
 Q.E.D.

C.117.41 Linear_Memory_Properties.xlat_pe_in_pdir_range_TCC3

Terse proof for xlat_pe_in_pdir_range_TCC3.

xlat_pe_in_pdir_range_TCC3:

$$\begin{aligned}
 & \{1\} \quad \forall (pm: \text{Plain_Memory}[\text{Linear_memory}], \text{addr}: \text{Memory_Address_4G}, \\
 & \quad s_1, s_2: \text{Linear_memory}[\text{Physical_memory}, pm_phy]): \\
 & \quad \text{union}(pm'ro_addr, pm'rw_addr)(\text{addr}) \wedge \\
 & \quad \text{is_linear_plain_memory?}(pm) \wedge pm'states(s_1) \wedge pm'states(s_2) \\
 & \quad \supset \\
 & \quad \text{Mem?}(xlat_idx[\text{Physical_memory}, pm_phy] \\
 & \quad \quad (\text{pdir_lvl}[\text{Physical_memory}, pm_phy], \\
 & \quad \quad \text{data}[\text{Physical_memory}, \text{Pdbr_type}] \\
 & \quad \quad \quad (\text{read_data}[\text{Physical_memory}, \text{Pdbr_type}] \\
 & \quad \quad \quad \quad (pm_phy, \text{pdbr_data_type})(\text{PDBR})(s_2))'base_addr, \\
 & \quad \quad \text{addr})'type_of) \\
 & \quad \wedge \\
 & \quad 0 \leq \\
 & \quad \quad xlat_idx[\text{Physical_memory}, pm_phy] \\
 & \quad \quad \quad (\text{pdir_lvl}[\text{Physical_memory}, pm_phy], \\
 & \quad \quad \quad \text{data}[\text{Physical_memory}, \text{Pdbr_type}] \\
 & \quad \quad \quad \quad (\text{read_data}[\text{Physical_memory}, \text{Pdbr_type}] \\
 & \quad \quad \quad \quad \quad (pm_phy, \text{pdbr_data_type})(\text{PDBR})(s_2))'base_addr, \\
 & \quad \quad \quad \text{addr})'offset \\
 & \quad \wedge \\
 & \quad \quad xlat_idx[\text{Physical_memory}, pm_phy] \\
 & \quad \quad \quad (\text{pdir_lvl}[\text{Physical_memory}, pm_phy], \\
 & \quad \quad \quad \text{data}[\text{Physical_memory}, \text{Pdbr_type}] \\
 & \quad \quad \quad \quad (\text{read_data}[\text{Physical_memory}, \text{Pdbr_type}] \\
 & \quad \quad \quad \quad \quad (pm_phy, \text{pdbr_data_type})(\text{PDBR})(s_2))'base_addr, \\
 & \quad \quad \quad \text{addr})'offset \\
 & \quad < \text{max_linear_offset}
 \end{aligned}$$

Installing automatic rewrites from: (bits_per_level pe_size Mem max_linear < max_level)

Repeatedly Skolemizing and flattening,

Case splitting on $\text{OK?}[\text{Physical_memory}, \text{Pdbr_type}] (\text{read_data}[\text{Physical_memory}, \text{Pdbr_type}] (pm_phy, \text{pdbr_data_type})(\text{PDBR})(s_2!1))$,

we get 2 subgoals:

xlat_pe_in_pdir_range_TCC3.1:

<pre> {-1} OK?[Physical_memory, Pdbr_type] (read_data[Physical_memory, Pdbr_type](pm_phy, pdbr_data_type)(PDBR)(s'_2)) {-2} Mem?(addr' 'type_of) {-3} 0 ≤ addr' 'offset {-4} addr' 'offset < max_linear_offset {-5} union(pm' 'ro_addr, pm' 'rw_addr)(addr') {-6} is_linear_plain_memory?(pm') {-7} pm' 'states(s'_1) {-8} pm' 'states(s'_2) </pre>	<hr/> <pre> {1} Mem?(xlat_idx[Physical_memory, pm_phy] (pdir_lvl[Physical_memory, pm_phy], data[Physical_memory, Pdbr_type] (read_data[Physical_memory, Pdbr_type] (pm_phy, pdbr_data_type)(PDBR)(s'_2)) 'base_addr, addr') 'type_of) ^ 0 ≤ xlat_idx[Physical_memory, pm_phy] (pdir_lvl[Physical_memory, pm_phy], data[Physical_memory, Pdbr_type] (read_data[Physical_memory, Pdbr_type] (pm_phy, pdbr_data_type)(PDBR)(s'_2)) 'base_addr, addr') 'offset ^ xlat_idx[Physical_memory, pm_phy] (pdir_lvl[Physical_memory, pm_phy], data[Physical_memory, Pdbr_type] (read_data[Physical_memory, Pdbr_type] (pm_phy, pdbr_data_type)(PDBR)(s'_2)) 'base_addr, addr') 'offset < max_linear_offset </pre>
--	--

Using lemma `xlat_idx_memory_address`,

we get 2 subgoals:

xlat_pe_in_pdir_range_TCC3.1.1:

```

{-1}  aligned?(bits_per_level + pe_size)
      (offset
        (data[Physical_memory, Pdbr_type]
          (read_data[Physical_memory, Pdbr_type]
            (pm_phy, pdbr_data_type)(PDBR)(s'_2))'base_addr))
      ⊃
      xlat_idx(pdir_lvl[Physical_memory, pm_phy],
        data[Physical_memory, Pdbr_type]
          (read_data[Physical_memory, Pdbr_type]
            (pm_phy, pdbr_data_type)(PDBR)(s'_2))'base_addr,
        addr')
{-2}  < max_linear
      OK?[Physical_memory, Pdbr_type]
        (read_data[Physical_memory, Pdbr_type](pm_phy, pdbr_data_type)(PDBR)(s'_2))
{-3}  Mem?(addr' 'type_of)
{-4}  0 ≤ addr' 'offset
{-5}  addr' 'offset < max_linear_offset
{-6}  union(pm' 'ro_addr, pm' 'rw_addr)(addr')
{-7}  is_linear_plain_memory?(pm')
{-8}  pm' 'states(s'_1)
{-9}  pm' 'states(s'_2)
-----
{1}  Mem?(xlat_idx[Physical_memory, pm_phy]
        (pdir_lvl[Physical_memory, pm_phy],
          data[Physical_memory, Pdbr_type]
            (read_data[Physical_memory, Pdbr_type]
              (pm_phy, pdbr_data_type)(PDBR)(s'_2))'base_addr,
          addr') 'type_of)
      ^
      0 ≤
      xlat_idx[Physical_memory, pm_phy]
        (pdir_lvl[Physical_memory, pm_phy],
          data[Physical_memory, Pdbr_type]
            (read_data[Physical_memory, Pdbr_type]
              (pm_phy, pdbr_data_type)(PDBR)(s'_2))'base_addr,
          addr') 'offset
      ^
      xlat_idx[Physical_memory, pm_phy]
        (pdir_lvl[Physical_memory, pm_phy],
          data[Physical_memory, Pdbr_type]
            (read_data[Physical_memory, Pdbr_type]
              (pm_phy, pdbr_data_type)(PDBR)(s'_2))'base_addr,
          addr') 'offset
      < max_linear_offset

```

Using lemma xlat_idx_memory_address2,

we get 2 subgoals:

xlat_pe_in_pdir_range_TCC3.1.1.1:

```

{-1}  aligned?(bits_per_level + pe_size)
      (offset
       (data[Physical_memory, Pdbr_type]
        (read_data[Physical_memory, Pdbr_type]
         (pm_phy, pdbr_data_type)(PDBR)(s'_2))'base_addr))
      ⊃
      0 ≤
      offset
      (xlat_idx(pdir_lvl[Physical_memory, pm_phy],
               data[Physical_memory, Pdbr_type]
               (read_data[Physical_memory, Pdbr_type]
                (pm_phy, pdbr_data_type)(PDBR)(s'_2))'base_addr,
               addr'))
{-2}  aligned?(bits_per_level + pe_size)
      (offset
       (data[Physical_memory, Pdbr_type]
        (read_data[Physical_memory, Pdbr_type]
         (pm_phy, pdbr_data_type)(PDBR)(s'_2))'base_addr))
      ⊃
      xlat_idx(pdir_lvl[Physical_memory, pm_phy],
               data[Physical_memory, Pdbr_type]
               (read_data[Physical_memory, Pdbr_type]
                (pm_phy, pdbr_data_type)(PDBR)(s'_2))'base_addr,
               addr')
      < max_linear
{-3}  OK?[Physical_memory, Pdbr_type]
      (read_data[Physical_memory, Pdbr_type](pm_phy, pdbr_data_type)(PDBR)(s'_2))
{-4}  Mem?(addr' 'type_of)
{-5}  0 ≤ addr' 'offset
{-6}  addr' 'offset < max_linear_offset
{-7}  union(pm' 'ro_addr, pm' 'rw_addr)(addr')
{-8}  is_linear_plain_memory?(pm')
{-9}  pm' 'states(s'_1)
{-10} pm' 'states(s'_2)
-----
{1}  Mem?(xlat_idx[Physical_memory, pm_phy]
          (pdir_lvl[Physical_memory, pm_phy],
           data[Physical_memory, Pdbr_type]
           (read_data[Physical_memory, Pdbr_type]
            (pm_phy, pdbr_data_type)(PDBR)(s'_2))'base_addr,
           addr')'type_of)
      ∧
      0 ≤
      xlat_idx[Physical_memory, pm_phy]
      (pdir_lvl[Physical_memory, pm_phy],
       data[Physical_memory, Pdbr_type]
       (read_data[Physical_memory, Pdbr_type]
        (pm_phy, pdbr_data_type)(PDBR)(s'_2))'base_addr,
       addr')'offset
      ∧
      xlat_idx[Physical_memory, pm_phy]
      (pdir_lvl[Physical_memory, pm_phy],
       data[Physical_memory, Pdbr_type]
       (read_data[Physical_memory, Pdbr_type]
        (pm_phy, pdbr_data_type)(PDBR)(s'_2))'base_addr,
       addr')'offset
      < max_linear_offset

```

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `xlat_pe_in_pdir_range_TCC3.1.1.1`.

`xlat_pe_in_pdir_range_TCC3.1.1.2`:

```

{-1}  aligned?(bits_per_level + pe_size)
      (offset
       (data[Physical_memory, Pdbr_type]
        (read_data[Physical_memory, Pdbr_type]
         (pm_phy, pdbr_data_type)(PDBR)(s'_2))'base_addr))
      ⊃
      xlat_idx(pdir_lvl[Physical_memory, pm_phy],
              data[Physical_memory, Pdbr_type]
              (read_data[Physical_memory, Pdbr_type]
               (pm_phy, pdbr_data_type)(PDBR)(s'_2))'base_addr,
              addr')
      < max_linear
{-2}  OK?[Physical_memory, Pdbr_type]
      (read_data[Physical_memory, Pdbr_type](pm_phy, pdbr_data_type)(PDBR)(s'_2))
{-3}  Mem?(addr' 'type_of)
{-4}  0 ≤ addr' 'offset
{-5}  addr' 'offset < max_linear_offset
{-6}  union(pm' 'ro_addr, pm' 'rw_addr)(addr')
{-7}  is_linear_plain_memory?(pm')
{-8}  pm' 'states(s'_1)
{-9}  pm' 'states(s'_2)
-----
{1}  pdir_lvl[Physical_memory, pm_phy] < 2
{2}  Mem?(xlat_idx[Physical_memory, pm_phy]
          (pdir_lvl[Physical_memory, pm_phy],
           data[Physical_memory, Pdbr_type]
           (read_data[Physical_memory, Pdbr_type]
            (pm_phy, pdbr_data_type)(PDBR)(s'_2))'base_addr,
           addr') 'type_of)
      ^
      0 ≤
      xlat_idx[Physical_memory, pm_phy]
          (pdir_lvl[Physical_memory, pm_phy],
           data[Physical_memory, Pdbr_type]
           (read_data[Physical_memory, Pdbr_type]
            (pm_phy, pdbr_data_type)(PDBR)(s'_2))'base_addr,
           addr') 'offset
      ^
      xlat_idx[Physical_memory, pm_phy]
          (pdir_lvl[Physical_memory, pm_phy],
           data[Physical_memory, Pdbr_type]
           (read_data[Physical_memory, Pdbr_type]
            (pm_phy, pdbr_data_type)(PDBR)(s'_2))'base_addr,
           addr') 'offset
      < max_linear_offset

```

Using lemma `xlat_pe_in_pdir_range_TCC1`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `xlat_pe_in_pdir_range_TCC3.1.1.2`.

xlat_pe_in_pdir_range_TCC3.1.2:

<pre> {-1} OK?[Physical_memory, Pdbr_type] (read_data[Physical_memory, Pdbr_type](pm_phy, pdbr_data_type)(PDBR)(s'_2)) {-2} Mem?(addr' 'type_of) {-3} 0 ≤ addr' 'offset {-4} addr' 'offset < max_linear_offset {-5} union(pm' 'ro_addr, pm' 'rw_addr)(addr') {-6} is_linear_plain_memory?(pm') {-7} pm' 'states(s'_1) {-8} pm' 'states(s'_2) </pre>	<hr/> <pre> {1} pdir_lvl[Physical_memory, pm_phy] < 2 {2} Mem?(xlat_idx[Physical_memory, pm_phy] (pdir_lvl[Physical_memory, pm_phy], data[Physical_memory, Pdbr_type] (read_data[Physical_memory, Pdbr_type] (pm_phy, pdbr_data_type)(PDBR)(s'_2)) 'base_addr, addr' 'type_of) ^ 0 ≤ xlat_idx[Physical_memory, pm_phy] (pdir_lvl[Physical_memory, pm_phy], data[Physical_memory, Pdbr_type] (read_data[Physical_memory, Pdbr_type] (pm_phy, pdbr_data_type)(PDBR)(s'_2)) 'base_addr, addr' 'offset) ^ xlat_idx[Physical_memory, pm_phy] (pdir_lvl[Physical_memory, pm_phy], data[Physical_memory, Pdbr_type] (read_data[Physical_memory, Pdbr_type] (pm_phy, pdbr_data_type)(PDBR)(s'_2)) 'base_addr, addr' 'offset) < max_linear_offset </pre>
--	--

Using lemma xlat_pe_in_pdir_range_TCC1,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of xlat_pe_in_pdir_range_TCC3.1.2.

xlat_pe_in_pdir_range_TCC3.2:

{-1}	Mem?(addr' 'type_of)
{-2}	$0 \leq \text{addr}'\text{'offset}$
{-3}	$\text{addr}'\text{'offset} < \text{max_linear_offset}$
{-4}	$\text{union}(\text{pm}'\text{'ro_addr}, \text{pm}'\text{'rw_addr})(\text{addr}'\text{'})$
{-5}	$\text{is_linear_plain_memory}?(\text{pm}'\text{'})$
{-6}	$\text{pm}'\text{'states}(s'_1)$
{-7}	$\text{pm}'\text{'states}(s'_2)$
{1}	$\text{OK?}[\text{Physical_memory}, \text{Pdbr_type}]$ $(\text{read_data}[\text{Physical_memory}, \text{Pdbr_type}](\text{pm_phy}, \text{pdbr_data_type})(\text{PDBR})(s'_2))$
{2}	$\text{Mem?}(\text{xlat_idx}[\text{Physical_memory}, \text{pm_phy}]$ $(\text{pdir_lvl}[\text{Physical_memory}, \text{pm_phy}],$ $\text{data}[\text{Physical_memory}, \text{Pdbr_type}]$ $(\text{read_data}[\text{Physical_memory}, \text{Pdbr_type}]$ $(\text{pm_phy}, \text{pdbr_data_type})(\text{PDBR})(s'_2))\text{'base_addr},$ $\text{addr}'\text{'type_of})$
	\wedge $0 \leq$ $\text{xlat_idx}[\text{Physical_memory}, \text{pm_phy}]$ $(\text{pdir_lvl}[\text{Physical_memory}, \text{pm_phy}],$ $\text{data}[\text{Physical_memory}, \text{Pdbr_type}]$ $(\text{read_data}[\text{Physical_memory}, \text{Pdbr_type}]$ $(\text{pm_phy}, \text{pdbr_data_type})(\text{PDBR})(s'_2))\text{'base_addr},$ $\text{addr}'\text{'offset}$
	\wedge $\text{xlat_idx}[\text{Physical_memory}, \text{pm_phy}]$ $(\text{pdir_lvl}[\text{Physical_memory}, \text{pm_phy}],$ $\text{data}[\text{Physical_memory}, \text{Pdbr_type}]$ $(\text{read_data}[\text{Physical_memory}, \text{Pdbr_type}]$ $(\text{pm_phy}, \text{pdbr_data_type})(\text{PDBR})(s'_2))\text{'base_addr},$ $\text{addr}'\text{'offset}$ $< \text{max_linear_offset}$

Using lemma xlat_pe_in_pdir_range_TCC2,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of xlat_pe_in_pdir_range_TCC3.2.

Q.E.D.

C.117.42 Linear_Memory_Properties.xlat_pe_in_pdir_range

Terse proof for xlat_pe_in_pdir_range.

xlat_pe_in_pdir_range:

<pre> {1} ∃ (pm: Plain_Memory [Linear_memory], addr: Memory_Address_4G, s1, s2: Lin- ear_memory): union(pm'ro_addr, pm'rw_addr)(addr) ∧ is_linear_plain_memory?(pm) ∧ pm'states(s1) ∧ pm'states(s2) ⊃ pe_in_pdir_range?(s1, restrict[Address, Memory_Address_4G, boolean] ((pm'ro_addr ∪ pm'rw_addr))) (xlat_idx(pdir_lvl, data(read_data(pm_phy, pdir_data_type)(PDBR) (s2))'base_addr, addr)) </pre>

Repeatedly Skolemizing and flattening,

Expanding the definition of pe_in_pdir_range?,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 2 subgoals:

xlat_pe_in_pdir_range.1:

<pre> {-1} Mem?(addr' 'type_of) {-2} 0 ≤ addr' 'offset {-3} addr' 'offset < max_linear_offset {-4} union(pm'ro_addr, pm'rw_addr)(addr') {-5} is_linear_plain_memory?(pm') {-6} pm' 'states(s1') {-7} pm' 'states(s2') </pre>	<pre> {1} ∃ (lin_a: (restrict[Address, Memory_Address_4G, boolean]((pm'ro_addr ∪ pm'rw_addr)))): xlat_idx(pdir_lvl, data(read_data(pm_phy, pdir_data_type)(PDBR)(s1'))'base_addr, lin_a) = xlat_idx(pdir_lvl, data(read_data(pm_phy, pdir_data_type)(PDBR)(s2'))'base_addr, addr') </pre>
---	--

Instantiating the top quantifier in 1 with the terms: (addr!1),

we get 2 subgoals:

xlat_pe_in_pdir_range.1.1:

<pre> {-1} Mem?(addr' 'type_of) {-2} 0 ≤ addr' 'offset {-3} addr' 'offset < max_linear_offset {-4} union(pm'ro_addr, pm'rw_addr)(addr') {-5} is_linear_plain_memory?(pm') {-6} pm' 'states(s1') {-7} pm' 'states(s2') </pre>	<pre> {1} xlat_idx(pdir_lvl, data(read_data(pm_phy, pdir_data_type)(PDBR)(s1'))'base_addr, addr') = xlat_idx(pdir_lvl, data(read_data(pm_phy, pdir_data_type)(PDBR)(s2'))'base_addr, addr') </pre>
---	---

Expanding the definition of is_linear_plain_memory?,

Applying disjunctive simplification to flatten sequent,

Instantiating the top quantifier in -9 with the terms: (s1!1 s2!1),

C Proof scripts

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `xlat_pe_in_pdir_range.1.1`.

`xlat_pe_in_pdir_range.1.2`:

{-1}	Mem?(addr' type_of)
{-2}	$0 \leq \text{addr}'\text{offset}$
{-3}	$\text{addr}'\text{offset} < \text{max_linear_offset}$
{-4}	$\text{union}(\text{pm}'\text{ro_addr}, \text{pm}'\text{rw_addr})(\text{addr}')$
{-5}	$\text{is_linear_plain_memory}?(pm')$
{-6}	$\text{pm}'\text{states}(s'_1)$
{-7}	$\text{pm}'\text{states}(s'_2)$
{1}	$\text{restrict}[\text{Address}, \text{Memory_Address_4G}, \text{boolean}]((\text{pm}'\text{ro_addr} \cup \text{pm}'\text{rw_addr})(\text{addr}'))$

Expanding the definition of `restrict`,
which is trivially true.

This completes the proof of `xlat_pe_in_pdir_range.1.2`.

`xlat_pe_in_pdir_range.2`:

{-1}	Mem?(addr' type_of)
{-2}	$0 \leq \text{addr}'\text{offset}$
{-3}	$\text{addr}'\text{offset} < \text{max_linear_offset}$
{-4}	$\text{union}(\text{pm}'\text{ro_addr}, \text{pm}'\text{rw_addr})(\text{addr}')$
{-5}	$\text{is_linear_plain_memory}?(pm')$
{-6}	$\text{pm}'\text{states}(s'_1)$
{-7}	$\text{pm}'\text{states}(s'_2)$
{1}	$\text{OK}?(read_data(\text{pm_phy}, \text{pdbr_data_type})(\text{PDBR})(s'_1))$

Expanding the definition of `is_linear_plain_memory?`,

Applying disjunctive simplification to flatten sequent,

Instantiating quantified variables,

This completes the proof of `xlat_pe_in_pdir_range.2`.

Q.E.D.

C.117.43 Linear_Memory_Properties.xlat_pe_in_pt_range_TCC1

Terse proof for `xlat_pe_in_pt_range_TCC1`.

`xlat_pe_in_pt_range_TCC1`:

{1}	$\forall (\text{pm}: \text{Plain_Memory}[\text{Linear_memory}], \text{addr}: \text{Memory_Address_4G},$ $s_1, s_2: \text{Linear_memory}[\text{Physical_memory}, \text{pm_phy}]):$ $\text{pm}'\text{states}(s_2) \wedge$ $\text{pm}'\text{states}(s_1) \wedge \text{is_linear_plain_memory}?(pm) \wedge \text{union}(\text{pm}'\text{ro_addr}, \text{pm}'\text{rw_addr})(\text{addr})$ \supset $\text{OK}[\text{Physical_memory}, \text{Pdbr_type}]$ $(\text{read_data}[\text{Physical_memory}, \text{Pdbr_type}](\text{pm_phy}, \text{pdbr_data_type})(\text{PDBR})(s_1))$ \vee $\text{Exception}[\text{Physical_memory}, \text{Pdbr_type}]$ $(\text{read_data}[\text{Physical_memory}, \text{Pdbr_type}](\text{pm_phy}, \text{pdbr_data_type})(\text{PDBR})(s_1))$
-----	--

Repeatedly Skolemizing and flattening,

Expanding the definition of `is_linear_plain_memory?`,

Applying disjunctive simplification to flatten sequent,

Instantiating quantified variables,

Instantiating quantified variables,

This completes the proof of `xlat_pe_in_pt_range_TCC1`.
Q.E.D.

C.117.44 Linear_Memory_Properties.xlat_pe_in_pt_range_TCC2

Terse proof for `xlat_pe_in_pt_range_TCC2`.

`xlat_pe_in_pt_range_TCC2`:

<pre>{1} ∃ (pm: Plain_Memory[Linear_memory], addr: Memory_Address_4G, s1, s2: Linear_memory[Physical_memory, pm_phy]): pm'states(s2) ∧ pm'states(s1) ∧ is_linear_plain_memory?(pm) ∧ union(pm'ro_addr, pm'rw_addr)(addr) ⊃ OK?[Physical_memory, Pdbr_type] (read_data[Physical_memory, Pdbr_type](pm_phy, pdbr_data_type)(PDBR)(s1))</pre>

Repeatedly Skolemizing and flattening,
Expanding the definition of `is_linear_plain_memory?`,
Applying disjunctive simplification to flatten sequent,
Instantiating quantified variables,
This completes the proof of `xlat_pe_in_pt_range_TCC2`.
Q.E.D.

C.117.45 Linear_Memory_Properties.xlat_pe_in_pt_range_TCC3

Terse proof for `xlat_pe_in_pt_range_TCC3`.

`xlat_pe_in_pt_range_TCC3`:

<pre>{1} ∃ (pm: Plain_Memory[Linear_memory], addr: Memory_Address_4G, access: Mem- ory_access, priv: Memory_privilege, s1, s2: Linear_memory[Physical_memory, pm_phy]): union(pm'ro_addr, pm'rw_addr)(addr) ∧ is_linear_plain_memory?(pm) ∧ pm'states(s1) ∧ pm'states(s2) ∧ OK?(translate(pdir_lvl, data(read_data(pm_phy, pdbr_data_type)(PDBR)(s1))'base_addr, addr, access, priv) (state(read_data(pm_phy, pdbr_data_type)(PDBR)(s1)))) ∧ ¬ data(translate(pdir_lvl, data(read_data(pm_phy, pdbr_data_type)(PDBR)(s1))'base_addr, addr, access, priv) (state(read_data(pm_phy, pdbr_data_type)(PDBR)(s1))))'1 ⊃ ptab_lvl[Physical_memory, pm_phy] < max_level</pre>
--

Repeatedly Skolemizing and flattening,
Keeping (2) and hiding *,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `xlat_pe_in_pt_range_TCC3`.
Q.E.D.

C.117.46 Linear_Memory_Properties.xlat_pe_in_pt_range_TCC4

Terse proof for xlat_pe_in_pt_range_TCC4.

xlat_pe_in_pt_range_TCC4:

```

{1}  ∀ (pm: Plain_Memory [Linear_memory], addr: Memory_Address_4G, access: Mem-
      ory_access,
      priv: Memory_privilege, s1, s2: Linear_memory [Physical_memory, pm_phy]):
      union(pm'ro_addr, pm'rw_addr)(addr) ∧
      is_linear_plain_memory?(pm) ∧
      pm'states(s1) ∧
      pm'states(s2) ∧
      OK?(translate(pdir_lvl,
                   data(read_data(pm_phy, pdir_data_type)(PDBR)(s1))'base_addr, addr,
                       access, priv)
             (state(read_data(pm_phy, pdir_data_type)(PDBR)(s1))))
      ∧
      ¬ data(translate(pdir_lvl,
                      data(read_data(pm_phy, pdir_data_type)(PDBR)(s1))'base_addr,
                          addr, access, priv)
             (state(read_data(pm_phy, pdir_data_type)(PDBR)(s1))))'1
      ⊃
      Mem?(data [Physical_memory, [bool, Address]]
            (translate [Physical_memory, pm_phy]
                      (pdir_lvl [Physical_memory, pm_phy],
                       data [Physical_memory, Pdir_type]
                         (read_data [Physical_memory, Pdir_type]
                           (pm_phy, pdir_data_type)(PDBR)(s1))'base_addr,
                           addr, access, priv)
                       (state [Physical_memory, Pdir_type]
                         (read_data [Physical_memory, Pdir_type]
                           (pm_phy, pdir_data_type)(PDBR)(s1))))'2'type_of)
      ∧
      0 ≤
      data [Physical_memory, [bool, Address]]
        (translate [Physical_memory, pm_phy]
                  (pdir_lvl [Physical_memory, pm_phy],
                   data [Physical_memory, Pdir_type]
                     (read_data [Physical_memory, Pdir_type]
                       (pm_phy, pdir_data_type)(PDBR)(s1))'base_addr,
                       addr, access, priv)
                   (state [Physical_memory, Pdir_type]
                     (read_data [Physical_memory, Pdir_type]
                       (pm_phy, pdir_data_type)(PDBR)(s1))))'2'offset
      ∧
      data [Physical_memory, [bool, Address]]
        (translate [Physical_memory, pm_phy]
                  (pdir_lvl [Physical_memory, pm_phy],
                   data [Physical_memory, Pdir_type]
                     (read_data [Physical_memory, Pdir_type]
                       (pm_phy, pdir_data_type)(PDBR)(s1))'base_addr,
                       addr, access, priv)
                   (state [Physical_memory, Pdir_type]
                     (read_data [Physical_memory, Pdir_type]
                       (pm_phy, pdir_data_type)(PDBR)(s1))))'2'offset
      < max_linear_offset

```

C Proof scripts

Repeatedly Skolemizing and flattening,

Installing automatic rewrites from: Mem max_linear < bits_per_level pe_size max_level

Using lemma xlat_pe_in_pt_range_TCC2,

Using lemma xlat_idx_memory_address,

Using lemma xlat_idx_memory_address2,

Using lemma translate_result_no_leaf,

Using lemma translate_memory_address,

Using lemma translate_memory_address2,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of xlat_pe_in_pt_range_TCC4.

Q.E.D.

C.117.47 Linear_Memory_Properties.xlat_pe_in_pt_range_TCC5

Terse proof for xlat_pe_in_pt_range_TCC5.

xlat_pe_in_pt_range_TCC5:

```

{1}  ∀ (pm: Plain_Memory [Linear_memory], addr: Memory_Address_4G, access: Mem-
      ory_access,
      priv: Memory_privilege, s1, s2: Linear_memory [Physical_memory, pm_phy]):
      union(pm'ro_addr, pm'rw_addr)(addr) ∧
      is_linear_plain_memory?(pm) ∧
      pm'states(s1) ∧
      pm'states(s2) ∧
      OK?(translate(pdir_lvl,
                   data(read_data(pm_phy, pdir_data_type)(PDBR)(s1))'base_addr, addr,
                       access, priv)
             (state(read_data(pm_phy, pdir_data_type)(PDBR)(s1))))
      ∧
      ¬ data(translate(pdir_lvl,
                     data(read_data(pm_phy, pdir_data_type)(PDBR)(s1))'base_addr,
                         addr, access, priv)
           (state(read_data(pm_phy, pdir_data_type)(PDBR)(s1))))'1
      ⊃
      Mem?(xlat_idx [Physical_memory, pm_phy]
           (ptab_lvl [Physical_memory, pm_phy],
            data [Physical_memory, [bool, Address]]
              (translate [Physical_memory, pm_phy]
                (pdir_lvl [Physical_memory, pm_phy],
                 data [Physical_memory, Pdir_type]
                   (read_data [Physical_memory, Pdir_type]
                     (pm_phy, pdir_data_type)(PDBR)(s1))'base_addr,
                     addr, access, priv)
                  (state [Physical_memory, Pdir_type]
                    (read_data [Physical_memory, Pdir_type]
                      (pm_phy, pdir_data_type)(PDBR)(s1))))'2,
            addr)'type_of)
      ∧
      0 ≤
      xlat_idx [Physical_memory, pm_phy]
        (ptab_lvl [Physical_memory, pm_phy],
         data [Physical_memory, [bool, Address]]
           (translate [Physical_memory, pm_phy]
             (pdir_lvl [Physical_memory, pm_phy],
              data [Physical_memory, Pdir_type]
                (read_data [Physical_memory, Pdir_type]
                  (pm_phy, pdir_data_type)(PDBR)(s1))'base_addr,
                  addr, access, priv)
              (state [Physical_memory, Pdir_type]
                (read_data [Physical_memory, Pdir_type]
                  (pm_phy, pdir_data_type)(PDBR)(s1))))'2,
         addr)'offset
      ∧
      xlat_idx [Physical_memory, pm_phy]
        (ptab_lvl [Physical_memory, pm_phy],
         data [Physical_memory, [bool, Address]]
           (translate [Physical_memory, pm_phy]
             (pdir_lvl [Physical_memory, pm_phy],
              data [Physical_memory, Pdir_type]
                (read_data [Physical_memory, Pdir_type]
                  (pm_phy, pdir_data_type)(PDBR)(s1))'base_addr,
                  addr, access, priv)
              (state [Physical_memory, Pdir_type]
                (read_data [Physical_memory, Pdir_type]
                  (pm_phy, pdir_data_type)(PDBR)(s1))))'2,
         addr)'offset
      < max_linear_offset

```

C Proof scripts

Repeatedly Skolemizing and flattening,

Installing automatic rewrites from: Mem max_linear < bits_per_level pe_size max_level

Using lemma xlat_idx_memory_address,

we get 3 subgoals:

xlat_pe_in_pt_range_TCC5.1:

```

{-1}  aligned?(bits_per_level + pe_size)
      (offset
       (data[Physical_memory, [bool, Address]]
        (translate[Physical_memory, pm_phy]
         (pdir_lvl[Physical_memory, pm_phy],
          data[Physical_memory, Pdbr_type]
          (read_data[Physical_memory, Pdbr_type]
           (pm_phy, pdbr_data_type)(PDBR)(s'_1))'base_addr,
           addr', access', priv')
          (state[Physical_memory, Pdbr_type]
           (read_data[Physical_memory, Pdbr_type]
            (pm_phy, pdbr_data_type)(PDBR)(s'_1))))'2))
      ⊃
      xlat_idx(ptab_lvl[Physical_memory, pm_phy],
              data[Physical_memory, [bool, Address]]
              (translate[Physical_memory, pm_phy]
               (pdir_lvl[Physical_memory, pm_phy],
                data[Physical_memory, Pdbr_type]
                (read_data[Physical_memory, Pdbr_type]
                 (pm_phy, pdbr_data_type)(PDBR)(s'_1))'base_addr,
                 addr', access', priv')
                (state[Physical_memory, Pdbr_type]
                 (read_data[Physical_memory, Pdbr_type]
                  (pm_phy, pdbr_data_type)(PDBR)(s'_1))))'2,
              addr')
      < max_linear
{-2}  Mem?(addr' 'type_of)
{-3}  0 ≤ addr' 'offset
{-4}  addr' 'offset < max_linear_offset
{-5}  union(pm' 'ro_addr, pm' 'rw_addr)(addr')
{-6}  is_linear_plain_memory?(pm')
{-7}  pm' 'states(s'_1)
{-8}  pm' 'states(s'_2)
{-9}  OK?(translate(pdir_lvl, data(read_data(pm_phy, pdbr_data_type)(PDBR)(s'_1))'base_addr,
                    addr', access', priv')
                 (state(read_data(pm_phy, pdbr_data_type)(PDBR)(s'_1))))
-----
{1}  data(translate(pdir_lvl, data(read_data(pm_phy, pdbr_data_type)(PDBR)(s'_1))'base_addr,
                    addr', access', priv')
         (state(read_data(pm_phy, pdbr_data_type)(PDBR)(s'_1))))'1
{2}  Mem?(xlat_idx[Physical_memory, pm_phy]
         (ptab_lvl[Physical_memory, pm_phy],
          data[Physical_memory, [bool, Address]]
          (translate[Physical_memory, pm_phy]
           (pdir_lvl[Physical_memory, pm_phy],
            data[Physical_memory, Pdbr_type]
            (read_data[Physical_memory, Pdbr_type]
             (pm_phy, pdbr_data_type)(PDBR)(s'_1))'base_addr,
             addr', access', priv')
            (state[Physical_memory, Pdbr_type]
             (read_data[Physical_memory, Pdbr_type]
              (pm_phy, pdbr_data_type)(PDBR)(s'_1))))'2,
          addr')'type_of)
      ^
      0 ≤
      xlat_idx[Physical_memory, pm_phy]
      (ptab_lvl[Physical_memory, pm_phy],
       data[Physical_memory, [bool, Address]]
       (translate[Physical_memory, pm_phy]
        (pdir_lvl[Physical_memory, pm_phy],
         data[Physical_memory, Pdbr_type]
         (read_data[Physical_memory, Pdbr_type]
          (pm_phy, pdbr_data_type)(PDBR)(s'_1))'base_addr,
          addr', access', priv')
         (state[Physical_memory, Pdbr_type]
          (read_data[Physical_memory, Pdbr_type]
           (pm_phy, pdbr_data_type)(PDBR)(s'_1))))'2,
       addr')'type_of)

```

C Proof scripts

Using lemma `xlat_idx_memory_address2`,

we get 3 subgoals:

xlat_pe_in_pt_range_TCC5.1.1:

```

{-1}  aligned?(bits_per_level + pe_size)
      (offset
        (data[Physical_memory, [bool, Address]]
          (translate[Physical_memory, pm_phy]
            (pdir_lvl[Physical_memory, pm_phy],
              data[Physical_memory, Pdbr_type]
                (read_data[Physical_memory, Pdbr_type]
                  (pm_phy, pdbr_data_type)(PDBR)(s'_1))'base_addr,
                  addr', access', priv')
                (state[Physical_memory, Pdbr_type]
                  (read_data[Physical_memory, Pdbr_type]
                    (pm_phy, pdbr_data_type)(PDBR)(s'_1))))'2))
        ⊃
        0 ≤
        offset
          (xlat_idx(ptab_lvl[Physical_memory, pm_phy],
            data[Physical_memory, [bool, Address]]
              (translate[Physical_memory, pm_phy]
                (pdir_lvl[Physical_memory, pm_phy],
                  data[Physical_memory, Pdbr_type]
                    (read_data[Physical_memory, Pdbr_type]
                      (pm_phy, pdbr_data_type)(PDBR)(s'_1))'base_addr,
                      addr', access', priv')
                    (state[Physical_memory, Pdbr_type]
                      (read_data[Physical_memory, Pdbr_type]
                        (pm_phy, pdbr_data_type)(PDBR)(s'_1))))'2,
                  addr'))
          addr')
{-2}  aligned?(bits_per_level + pe_size)
      (offset
        (data[Physical_memory, [bool, Address]]
          (translate[Physical_memory, pm_phy]
            (pdir_lvl[Physical_memory, pm_phy],
              data[Physical_memory, Pdbr_type]
                (read_data[Physical_memory, Pdbr_type]
                  (pm_phy, pdbr_data_type)(PDBR)(s'_1))'base_addr,
                  addr', access', priv')
                (state[Physical_memory, Pdbr_type]
                  (read_data[Physical_memory, Pdbr_type]
                    (pm_phy, pdbr_data_type)(PDBR)(s'_1))))'2))
        ⊃
        xlat_idx(ptab_lvl[Physical_memory, pm_phy],
          data[Physical_memory, [bool, Address]]
            (translate[Physical_memory, pm_phy]
              (pdir_lvl[Physical_memory, pm_phy],
                data[Physical_memory, Pdbr_type]
                  (read_data[Physical_memory, Pdbr_type]
                    (pm_phy, pdbr_data_type)(PDBR)(s'_1))'base_addr,
                    addr', access', priv')
                  (state[Physical_memory, Pdbr_type]
                    (read_data[Physical_memory, Pdbr_type]
                      (pm_phy, pdbr_data_type)(PDBR)(s'_1))))'2,
                addr'))
          < max_linear

```

```

{-3}  Mem?(addr' 'type_of)

```

1695

```

{-4}  0 ≤ addr' 'offset

```

```

{-5}  addr' 'offset < max_linear_offset

```

```

{-6}  union(pm' 'ro_addr, pm' 'rw_addr)(addr')

```

```

{-7}  is_linear_plain_memory?(pm')

```

```

{-8}  pm' 'states(s'_1)

```

```

{-9}  pm' 'states(s'_2)

```

```

{-10} OK?(translate(pdir_lvl, data(read_data(pm_phy, pdbr_data_type)(PDBR)(s'_1))'base_addr,

```


C Proof scripts

Using lemma `translate_result_leaf`,

Using lemma `translate_memory_address`,

Using lemma `translate_memory_address2`,

Using lemma `translate_result_no_leaf`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `xlat_pe_in_pt_range_TCC5.1.1`.

xlat_pe_in_pt_range_TCC5.1.2:

```

{-1}  aligned?(bits_per_level + pe_size)
      (offset
       (data[Physical_memory, [bool, Address]]
        (translate[Physical_memory, pm_phy]
         (pdir_lvl[Physical_memory, pm_phy],
          data[Physical_memory, Pdbr_type]
          (read_data[Physical_memory, Pdbr_type]
           (pm_phy, pdbr_data_type)(PDBR)(s'_1))'base_addr,
           addr', access', priv')
          (state[Physical_memory, Pdbr_type]
           (read_data[Physical_memory, Pdbr_type]
            (pm_phy, pdbr_data_type)(PDBR)(s'_1))))'2))
      ⊃
      xlat_idx(ptab_lvl[Physical_memory, pm_phy],
              data[Physical_memory, [bool, Address]]
              (translate[Physical_memory, pm_phy]
               (pdir_lvl[Physical_memory, pm_phy],
                data[Physical_memory, Pdbr_type]
                (read_data[Physical_memory, Pdbr_type]
                 (pm_phy, pdbr_data_type)(PDBR)(s'_1))'base_addr,
                 addr', access', priv')
                (state[Physical_memory, Pdbr_type]
                 (read_data[Physical_memory, Pdbr_type]
                  (pm_phy, pdbr_data_type)(PDBR)(s'_1))))'2,
              addr')
      < max_linear
{-2}  Mem?(addr' 'type_of)
{-3}  0 ≤ addr' 'offset
{-4}  addr' 'offset < max_linear_offset
{-5}  union(pm' 'ro_addr, pm' 'rw_addr)(addr')
{-6}  is_linear_plain_memory?(pm')
{-7}  pm' 'states(s'_1)
{-8}  pm' 'states(s'_2)
{-9}  OK?(translate(pdir_lvl, data(read_data(pm_phy, pdbr_data_type)(PDBR)(s'_1))'base_addr,
                                addr', access', priv')
                  (state(read_data(pm_phy, pdbr_data_type)(PDBR)(s'_1))))
-----
{1}  Mem?(data[Physical_memory, [bool, Address]]
      (translate[Physical_memory, pm_phy]
       (pdir_lvl[Physical_memory, pm_phy],
        data[Physical_memory, Pdbr_type]
        (read_data[Physical_memory, Pdbr_type]
         (pm_phy, pdbr_data_type)(PDBR)(s'_1))'base_addr,
         addr', access', priv')
        (state[Physical_memory, Pdbr_type]
         (read_data[Physical_memory, Pdbr_type]
          (pm_phy, pdbr_data_type)(PDBR)(s'_1))))'2 'type_of)
      ^
      0 ≤
      data[Physical_memory, [bool, Address]]
      (translate[Physical_memory, pm_phy]
       (pdir_lvl[Physical_memory, pm_phy],
        data[Physical_memory, Pdbr_type]
        (read_data[Physical_memory, Pdbr_type]
         (pm_phy, pdbr_data_type)(PDBR)(s'_1))'base_addr, 1697
         addr', access', priv')
        (state[Physical_memory, Pdbr_type]
         (read_data[Physical_memory, Pdbr_type]
          (pm_phy, pdbr_data_type)(PDBR)(s'_1))))'2 'offset
      ^
      data[Physical_memory, [bool, Address]]
      (translate[Physical_memory, pm_phy]

```

C Proof scripts

Using lemma `translate_memory_address`,

Using lemma `translate_memory_address2`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `xlat_pe_in_pt_range_TCC5.1.2`.

xlat_pe_in_pt_range_TCC5.1.3:

```

{-1}  aligned?(bits_per_level + pe_size)
      (offset
       (data[Physical_memory, [bool, Address]]
        (translate[Physical_memory, pm_phy]
         (pdir_lvl[Physical_memory, pm_phy],
          data[Physical_memory, Pdbr_type]
          (read_data[Physical_memory, Pdbr_type]
           (pm_phy, pdbr_data_type)(PDBR)(s'_1))'base_addr,
           addr', access', priv')
          (state[Physical_memory, Pdbr_type]
           (read_data[Physical_memory, Pdbr_type]
            (pm_phy, pdbr_data_type)(PDBR)(s'_1))))'2))
      ⊃
      xlat_idx(ptab_lvl[Physical_memory, pm_phy],
              data[Physical_memory, [bool, Address]]
              (translate[Physical_memory, pm_phy]
               (pdir_lvl[Physical_memory, pm_phy],
                data[Physical_memory, Pdbr_type]
                (read_data[Physical_memory, Pdbr_type]
                 (pm_phy, pdbr_data_type)(PDBR)(s'_1))'base_addr,
                 addr', access', priv')
                (state[Physical_memory, Pdbr_type]
                 (read_data[Physical_memory, Pdbr_type]
                  (pm_phy, pdbr_data_type)(PDBR)(s'_1))))'2,
              addr')
      < max_linear
{-2}  Mem?(addr' 'type_of)
{-3}  0 ≤ addr' 'offset
{-4}  addr' 'offset < max_linear_offset
{-5}  union(pm' 'ro_addr, pm' 'rw_addr)(addr')
{-6}  is_linear_plain_memory?(pm')
{-7}  pm' 'states(s'_1)
{-8}  pm' 'states(s'_2)
{-9}  OK?(translate(pdir_lvl, data(read_data(pm_phy, pdbr_data_type)(PDBR)(s'_1))'base_addr,
                        addr', access', priv')
                (state(read_data(pm_phy, pdbr_data_type)(PDBR)(s'_1))))
-----
{1}  ptab_lvl[Physical_memory, pm_phy] < 2
{2}  data(translate(pdir_lvl, data(read_data(pm_phy, pdbr_data_type)(PDBR)(s'_1))'base_addr,
                        addr', access', priv')
        (state(read_data(pm_phy, pdbr_data_type)(PDBR)(s'_1))))'1
{3}  Mem?(xlat_idx[Physical_memory, pm_phy]
          (ptab_lvl[Physical_memory, pm_phy],
           data[Physical_memory, [bool, Address]]
           (translate[Physical_memory, pm_phy]
            (pdir_lvl[Physical_memory, pm_phy],
             data[Physical_memory, Pdbr_type]
             (read_data[Physical_memory, Pdbr_type]
              (pm_phy, pdbr_data_type)(PDBR)(s'_1))'base_addr,
              addr', access', priv')
             (state[Physical_memory, Pdbr_type]
              (read_data[Physical_memory, Pdbr_type]
               (pm_phy, pdbr_data_type)(PDBR)(s'_1))))'2,
          addr')'type_of)
^
0 ≤
  xlat_idx[Physical_memory, pm_phy]
  (ptab_lvl[Physical_memory, pm_phy],
   data[Physical_memory, [bool, Address]]
   (translate[Physical_memory, pm_phy]
    (pdir_lvl[Physical_memory, pm_phy],
     data[Physical_memory, Pdbr_type]
     (read_data[Physical_memory, Pdbr_type]
      (pm_phy, pdbr_data_type)(PDBR)(s'_1))'base_addr,
      addr', access', priv')
     (state[Physical_memory, Pdbr_type]
      (read_data[Physical_memory, Pdbr_type]
       (pm_phy, pdbr_data_type)(PDBR)(s'_1))))'2,
   addr')'type_of)

```

C Proof scripts

Keeping (1) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `xlat_pe_in_pt_range_TCC5.1.3`.

xlat_pe_in_pt_range_TCC5.2:

```

{-1} Mem?(addr' 'type_of)
{-2} 0 ≤ addr' 'offset
{-3} addr' 'offset < max_linear_offset
{-4} union(pm' 'ro_addr, pm' 'rw_addr)(addr')
{-5} is_linear_plain_memory?(pm')
{-6} pm' 'states(s'_1)
{-7} pm' 'states(s'_2)
{-8} OK?(translate(pdir_lvl, data(read_data(pm_phy, pdir_data_type)(PDBR)(s'_1)) 'base_addr,
      addr', access', priv')
      (state(read_data(pm_phy, pdir_data_type)(PDBR)(s'_1))))
-----
{1} Mem?(data[Physical_memory, [bool, Address]]
      (translate[Physical_memory, pm_phy]
      (pdir_lvl[Physical_memory, pm_phy],
      data[Physical_memory, Pdir_type]
      (read_data[Physical_memory, Pdir_type]
      (pm_phy, pdir_data_type)(PDBR)(s'_1)) 'base_addr,
      addr', access', priv')
      (state[Physical_memory, Pdir_type]
      (read_data[Physical_memory, Pdir_type]
      (pm_phy, pdir_data_type)(PDBR)(s'_1)))) '2 'type_of)
^
0 ≤
data[Physical_memory, [bool, Address]]
  (translate[Physical_memory, pm_phy]
  (pdir_lvl[Physical_memory, pm_phy],
  data[Physical_memory, Pdir_type]
  (read_data[Physical_memory, Pdir_type]
  (pm_phy, pdir_data_type)(PDBR)(s'_1)) 'base_addr,
  addr', access', priv')
  (state[Physical_memory, Pdir_type]
  (read_data[Physical_memory, Pdir_type]
  (pm_phy, pdir_data_type)(PDBR)(s'_1)))) '2 'offset
^
data[Physical_memory, [bool, Address]]
  (translate[Physical_memory, pm_phy]
  (pdir_lvl[Physical_memory, pm_phy],
  data[Physical_memory, Pdir_type]
  (read_data[Physical_memory, Pdir_type]
  (pm_phy, pdir_data_type)(PDBR)(s'_1)) 'base_addr,
  addr', access', priv')
  (state[Physical_memory, Pdir_type]
  (read_data[Physical_memory, Pdir_type]
  (pm_phy, pdir_data_type)(PDBR)(s'_1)))) '2 'offset
< max_linear_offset
{2} data(translate(pdir_lvl, data(read_data(pm_phy, pdir_data_type)(PDBR)(s'_1)) 'base_addr,
      addr', access', priv')
      (state(read_data(pm_phy, pdir_data_type)(PDBR)(s'_1)))) '1
{3} Mem?(xlat_idx[Physical_memory, pm_phy]
      (ptab_lvl[Physical_memory, pm_phy],
      data[Physical_memory, [bool, Address]]
      (translate[Physical_memory, pm_phy]
      (pdir_lvl[Physical_memory, pm_phy],
      data[Physical_memory, Pdir_type]
      (read_data[Physical_memory, Pdir_type]
      (pm_phy, pdir_data_type)(PDBR)(s'_1)) 'base_addr,
      addr', access', priv')
      (state[Physical_memory, Pdir_type]
      (read_data[Physical_memory, Pdir_type]
      (pm_phy, pdir_data_type)(PDBR)(s'_1)))) '2,
      addr') 'type_of)
^

```

C Proof scripts

Using lemma `translate_memory_address`,

Using lemma `translate_memory_address2`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `xlat_pe_in_pt_range_TCC5.2`.

xlat_pe_in_pt_range_TCC5.3:

```

{-1} Mem?(addr' 'type_of)
{-2} 0 ≤ addr' 'offset
{-3} addr' 'offset < max_linear_offset
{-4} union(pm' 'ro_addr, pm' 'rw_addr)(addr')
{-5} is_linear_plain_memory?(pm')
{-6} pm' 'states(s'_1)
{-7} pm' 'states(s'_2)
{-8} OK?(translate(pdir_lvl, data(read_data(pm_phy, pdir_data_type)(PDBR)(s'_1)) 'base_addr,
                    addr', access', priv')
              (state(read_data(pm_phy, pdir_data_type)(PDBR)(s'_1))))
-----
{1} ptab_lvl[Physical_memory, pm_phy] < 2
{2} data(translate(pdir_lvl, data(read_data(pm_phy, pdir_data_type)(PDBR)(s'_1)) 'base_addr,
                    addr', access', priv')
      (state(read_data(pm_phy, pdir_data_type)(PDBR)(s'_1)))) '1
{3} Mem?(xlat_idx[Physical_memory, pm_phy]
      (ptab_lvl[Physical_memory, pm_phy],
       data[Physical_memory, [bool, Address]]
         (translate[Physical_memory, pm_phy]
           (pdir_lvl[Physical_memory, pm_phy],
            data[Physical_memory, Pdir_type]
              (read_data[Physical_memory, Pdir_type]
                (pm_phy, pdir_data_type)(PDBR)(s'_1)) 'base_addr,
                addr', access', priv')
            (state[Physical_memory, Pdir_type]
              (read_data[Physical_memory, Pdir_type]
                (pm_phy, pdir_data_type)(PDBR)(s'_1)))) '2,
        addr') 'type_of)
    ^
    0 ≤
    xlat_idx[Physical_memory, pm_phy]
      (ptab_lvl[Physical_memory, pm_phy],
       data[Physical_memory, [bool, Address]]
         (translate[Physical_memory, pm_phy]
           (pdir_lvl[Physical_memory, pm_phy],
            data[Physical_memory, Pdir_type]
              (read_data[Physical_memory, Pdir_type]
                (pm_phy, pdir_data_type)(PDBR)(s'_1)) 'base_addr,
                addr', access', priv')
            (state[Physical_memory, Pdir_type]
              (read_data[Physical_memory, Pdir_type]
                (pm_phy, pdir_data_type)(PDBR)(s'_1)))) '2,
        addr') 'offset
    ^
    xlat_idx[Physical_memory, pm_phy]
      (ptab_lvl[Physical_memory, pm_phy],
       data[Physical_memory, [bool, Address]]
         (translate[Physical_memory, pm_phy]
           (pdir_lvl[Physical_memory, pm_phy],
            data[Physical_memory, Pdir_type]
              (read_data[Physical_memory, Pdir_type]
                (pm_phy, pdir_data_type)(PDBR)(s'_1)) 'base_addr,
                addr', access', priv')
            (state[Physical_memory, Pdir_type]
              (read_data[Physical_memory, Pdir_type]
                (pm_phy, pdir_data_type)(PDBR)(s'_1)))) '2,
        addr') 'offset
    < max_linear_offset

```


Keeping (1) and hiding *,
 Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of `xlat_pe_in_pt_range_TCC5.3`.
 Q.E.D.

C.117.48 Linear_Memory_Properties.xlat_pe_in_pt_range

Terse proof for `xlat_pe_in_pt_range`.

`xlat_pe_in_pt_range`:

```

{1}  ∀ (pm: Plain_Memory [Linear_memory], addr: Memory_Address_4G, access: Mem-
      ory_access,
      priv: Memory_privilege, s1, s2: Linear_memory):
      union(pm'ro_addr, pm'rw_addr)(addr) ∧
      is_linear_plain_memory?(pm) ∧
      pm'states(s1) ∧
      pm'states(s2) ∧
      OK?(translate(pdir_lvl,
                   data(read_data(pm_phy, pdbr_data_type)(PDBR)(s1))'base_addr, addr,
                   access, priv)
             (state(read_data(pm_phy, pdbr_data_type)(PDBR)(s1))))
      ∧
      ¬ data(translate(pdir_lvl,
                     data(read_data(pm_phy, pdbr_data_type)(PDBR)(s1))'base_addr,
                     addr, access, priv)
              (state(read_data(pm_phy, pdbr_data_type)(PDBR)(s1))))'1
      ⊃
      pe_in_pt_range?(s2,
                      restrict [Address, Memory_Address_4G, boolean]
                        ((pm'ro_addr ∪ pm'rw_addr)),
                      ptab_lvl)
      (xlat_idx(ptab_lvl,
               data(translate(pdir_lvl,
                             data(read_data(pm_phy, pdbr_data_type)
                                         (PDBR)
                                         (s1))'base_addr,
                             addr, access, priv)
                       (state(read_data(pm_phy, pdbr_data_type)
                                         (PDBR)
                                         (s1))))'2,
               addr))
    
```

Installing automatic rewrites from: (bits_per_level pe_size singleton has_next_state)
 Repeatedly Skolemizing and flattening,
 Expanding the definition of `pe_in_pt_range?`,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 we get 2 subgoals:

xlat_pe_in_pt_range.1:

```

{-1} Mem?(addr' type_of)
{-2} 0 ≤ addr' offset
{-3} addr' offset < max_linear_offset
{-4} union(pm' ro_addr, pm' rw_addr)(addr')
{-5} is_linear_plain_memory?(pm')
{-6} pm' states(s'_1)
{-7} pm' states(s'_2)
{-8} OK?(translate(pdir_lvl, data(read_data(pm_phy, pdir_data_type)(PDBR)(s'_1)) base_addr,
                    addr', access', priv')
            (state(read_data(pm_phy, pdir_data_type)(PDBR)(s'_1))))
{-9} ptab_lvl = pdir_lvl
{1} data(translate(pdir_lvl, data(read_data(pm_phy, pdir_data_type)(PDBR)(s'_1)) base_addr,
                    addr', access', priv')
          (state(read_data(pm_phy, pdir_data_type)(PDBR)(s'_1)))) '1
{2} pe_in_pdir_range?(s'_2,
    restrict[Address, Memory_Address_4G, boolean]
      ((pm' ro_addr ∪ pm' rw_addr))
    (xlat_idx(ptab_lvl,
              data(translate(pdir_lvl,
                            data(read_data(pm_phy, pdir_data_type)
                                      (PDBR)
                                      (s'_1)) base_addr,
                            addr', access', priv')
                          (state(read_data(pm_phy, pdir_data_type)
                                      (PDBR)
                                      (s'_1)))) '2,
              addr'))

```

Keeping (-9) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of xlat_pe_in_pt_range.1.

C Proof scripts

xlat_pe_in_pt_range.2:

<pre> {-1} Mem?(addr' 'type_of) {-2} 0 ≤ addr' 'offset {-3} addr' 'offset < max_linear_offset {-4} union(pm' 'ro_addr, pm' 'rw_addr)(addr') {-5} is_linear_plain_memory?(pm') {-6} pm' 'states(s'_1) {-7} pm' 'states(s'_2) {-8} OK?(translate(pdir_lvl, data(read_data(pm_phy, pdir_data_type)(PDBR)(s'_1)) 'base_addr, addr', access', priv') (state(read_data(pm_phy, pdir_data_type)(PDBR)(s'_1)))) </pre>	<pre> {1} data(translate(pdir_lvl, data(read_data(pm_phy, pdir_data_type)(PDBR)(s'_1)) 'base_addr, addr', access', priv') (state(read_data(pm_phy, pdir_data_type)(PDBR)(s'_1)))) '1 {2} ptab_lvl = pdir_lvl {3} pe_in_ptab_range?(s'_2, restrict[Address, Memory_Address_4G, boolean] ((pm' 'ro_addr ∪ pm' 'rw_addr))) (xlat_idx(ptab_lvl, data(translate(pdir_lvl, data(read_data(pm_phy, pdir_data_type) (PDBR) (s'_1)) 'base_addr, addr', access', priv') (state(read_data(pm_phy, pdir_data_type) (PDBR) (s'_1)))) '2, addr')) </pre>
---	--

Copying formula number: -5

Expanding the definition of is_linear_plain_memory?,

Applying disjunctive simplification to flatten sequent,

Hiding formulas: (-1 -2 -3 -5 -7 -8),

Case splitting on $OK?(read_data(pm_phy, pdir_data_type)(PDBR)(s1!1)), pm!1'states(state(read_data[Physical_memory, Pdir_type](pm_phy, pdir_data_type)(PDBR)(s1!1)))$,

we get 3 subgoals:

xlat_pe_in_pt_range.2.1:

{-1}	pm' states
	(state(read_data[Physical_memory, Pdbr_type](pm_phy, pdbr_data_type)(PDBR)(s'_1)))
{-2}	OK?(read_data(pm_phy, pdbr_data_type)(PDBR)(s'_1))
{-3}	$\forall (s: (pm' \text{ states})): \text{OK?}(\text{read_data}(\text{pm_phy}, \text{pdbr_data_type})(\text{PDBR})(s))$
{-4}	$\forall (s_1, s_2: (pm' \text{ states})): \text{pe_in_pdir_range?}(s_1,$ $\text{restrict}[\text{Address}, \text{Memory_Address_4G}, \text{boolean}]$ $((\text{pm}'\text{ro_addr} \cup \text{pm}'\text{rw_addr})))$ $=$ $\text{pe_in_pdir_range?}(s_2,$ $\text{restrict}[\text{Address}, \text{Memory_Address_4G}, \text{boolean}]$ $((\text{pm}'\text{ro_addr} \cup \text{pm}'\text{rw_addr})))$ \wedge $\text{pe_in_ptab_range?}(s_1,$ $\text{restrict}[\text{Address}, \text{Memory_Address_4G}, \text{boolean}]$ $((\text{pm}'\text{ro_addr} \cup \text{pm}'\text{rw_addr})))$ $=$ $\text{pe_in_ptab_range?}(s_2,$ $\text{restrict}[\text{Address}, \text{Memory_Address_4G}, \text{boolean}]$ $((\text{pm}'\text{ro_addr} \cup \text{pm}'\text{rw_addr})))$
{-5}	plain_memory?(pm_phy)
{-6}	pm' states = pm_phy' states
{-7}	(pm' other_actions \subseteq pm_phy' other_actions)
{-8}	(linear_resolve_register_transformers \subseteq pm_phy' other_actions)
{-9}	$\forall (a: ((\text{pm}'\text{ro_addr} \cup \text{pm}'\text{rw_addr})))$: $\text{Mem?}(\text{type_of}(a)) \wedge 0 \leq \text{offset}(a) \wedge \text{offset}(a) < \text{max_linear_offset}$
{-10}	$\forall (s: (pm' \text{ states})): \text{linear_blessed?}(s, \text{restrict}[\text{Address}, \text{Memory_Address_4G}, \text{boolean}](\text{pm}'\text{ro_addr}),$ $\text{restrict}[\text{Address}, \text{Memory_Address_4G}, \text{boolean}](\text{pm}'\text{rw_addr}))$
{-11}	Mem?(addr' type_of)
{-12}	$0 \leq \text{addr}'\text{offset}$
{-13}	$\text{addr}'\text{offset} < \text{max_linear_offset}$
{-14}	$\text{union}(\text{pm}'\text{ro_addr}, \text{pm}'\text{rw_addr})(\text{addr}')$
{-15}	is_linear_plain_memory?(pm')
{-16}	pm' states(s'_1)
{-17}	pm' states(s'_2)
{-18}	$\text{OK?}(\text{translate}(\text{pdir_lvl}, \text{data}(\text{read_data}(\text{pm_phy}, \text{pdbr_data_type})(\text{PDBR})(s'_1))\text{'base_addr},$ $\text{addr}', \text{access}', \text{priv}')$ $(\text{state}(\text{read_data}(\text{pm_phy}, \text{pdbr_data_type})(\text{PDBR})(s'_1))))$
{1}	$\text{data}(\text{translate}(\text{pdir_lvl}, \text{data}(\text{read_data}(\text{pm_phy}, \text{pdbr_data_type})(\text{PDBR})(s'_1))\text{'base_addr},$ $\text{addr}', \text{access}', \text{priv}')$ $(\text{state}(\text{read_data}(\text{pm_phy}, \text{pdbr_data_type})(\text{PDBR})(s'_1)))) \text{'1}$
{2}	ptab_lvl = pdir_lvl
{3}	$\text{pe_in_ptab_range?}(s'_2,$ $\text{restrict}[\text{Address}, \text{Memory_Address_4G}, \text{boolean}]$ $((\text{pm}'\text{ro_addr} \cup \text{pm}'\text{rw_addr})))$ $(\text{xlat_idx}(\text{ptab_lvl},$ $\text{data}(\text{translate}(\text{pdir_lvl},$ $\text{data}(\text{read_data}(\text{pm_phy}, \text{pdbr_data_type})$ (PDBR) $(s'_1))\text{'base_addr},$ $\text{addr}', \text{access}', \text{priv}')$ $(\text{state}(\text{read_data}(\text{pm_phy}, \text{pdbr_data_type})$ (PDBR) $(s'_1)))) \text{'2},$ $\text{addr}')$

C Proof scripts

Instantiating the top quantifier in -4 with the terms: `(s2!1 state(read_data[Physical_memory, Pdbr_type] (pm_phy, pdbr_data_type)(PDBR)(s1!1)))`,

we get 2 subgoals:

xlat_pe_in_pt_range.2.1.1:

```

{-1} pm' states
      (state(read_data[Physical_memory, Pdbr_type](pm_phy, pdbr_data_type)(PDBR)(s'_1)))
{-2} OK?(read_data(pm_phy, pdbr_data_type)(PDBR)(s'_1))
{-3}  $\forall (s: (pm' states)): OK?(read_data(pm_phy, pdbr_data_type)(PDBR)(s))$ 
{-4} pe_in_pdir_range?(s'_2,
      restrict[Address, Memory_Address_4G, boolean]
      ((pm'ro_addr  $\cup$  pm'rw_addr)))
=
pe_in_pdir_range?(state(read_data[Physical_memory, Pdbr_type]
      (pm_phy, pdbr_data_type)(PDBR)(s'_1),
      restrict[Address, Memory_Address_4G, boolean]
      ((pm'ro_addr  $\cup$  pm'rw_addr))))
^
pe_in_ptab_range?(s'_2,
      restrict[Address, Memory_Address_4G, boolean]
      ((pm'ro_addr  $\cup$  pm'rw_addr)))
=
pe_in_ptab_range?(state(read_data[Physical_memory, Pdbr_type]
      (pm_phy, pdbr_data_type)(PDBR)(s'_1),
      restrict[Address, Memory_Address_4G, boolean]
      ((pm'ro_addr  $\cup$  pm'rw_addr))))
{-5} plain_memory?(pm_phy)
{-6} pm' states = pm_phy states
{-7} (pm'other_actions  $\subseteq$  pm_phy'other_actions)
{-8} (linear_resolve_register_transformers  $\subseteq$  pm_phy'other_actions)
{-9}  $\forall (a: ((pm'ro_addr  $\cup$  pm'rw_addr)))$ :
      Mem?(type_of(a))  $\wedge$  0  $\leq$  offset(a)  $\wedge$  offset(a) < max_linear_offset
{-10}  $\forall (s: (pm' states))$ :
      linear_blessed?(s, restrict[Address, Memory_Address_4G, boolean](pm'ro_addr),
      restrict[Address, Memory_Address_4G, boolean](pm'rw_addr))
{-11} Mem?(addr' type_of)
{-12} 0  $\leq$  addr' offset
{-13} addr' offset < max_linear_offset
{-14} union(pm'ro_addr, pm'rw_addr)(addr')
{-15} is_linear_plain_memory?(pm')
{-16} pm' states(s'_1)
{-17} pm' states(s'_2)
{-18} OK?(translate(pdir_lvl, data(read_data(pm_phy, pdbr_data_type)(PDBR)(s'_1))'base_addr,
      addr', access', priv')
      (state(read_data(pm_phy, pdbr_data_type)(PDBR)(s'_1))))
-----
{1} data(translate(pdir_lvl, data(read_data(pm_phy, pdbr_data_type)(PDBR)(s'_1))'base_addr,
      addr', access', priv')
      (state(read_data(pm_phy, pdbr_data_type)(PDBR)(s'_1))))'1
{2} ptab_lvl = pdir_lvl
{3} pe_in_ptab_range?(s'_2,
      restrict[Address, Memory_Address_4G, boolean]
      ((pm'ro_addr  $\cup$  pm'rw_addr)))
      (xlat_idx(ptab_lvl,
      data(translate(pdir_lvl,
      data(read_data(pm_phy, pdbr_data_type)
      (PDBR)
      (s'_1))'base_addr,
      addr', access', priv')
      (state(read_data(pm_phy, pdbr_data_type)
      (PDBR)
      (s'_1))))'1,
      addr'))

```

C Proof scripts

Applying disjunctive simplification to flatten sequent,

Replacing using formula -5,

Expanding the definition of `pe_in_ptab_range?`,

```
Case splitting on pe_in_pdir_range?[Physical_memory, pm_phy] (state(read_data[Physical_memory, Pdir_type] (pm_phy, pdir_data_type)(PDBR)(s1!1)), restrict[Address, Memory_Address_4G, boolean] (union[Address](pm!1'ro_addr, pm!1'rw_addr))) (xlat_idx[Physical_memory, pm_phy] (pdir_lvl, data[Physical_memory, Pdir_type] (read_data[Physical_memory, Pdir_type] (pm_phy, pdir_data_type)(PDBR)(s1!1))'base_addr, addr!1)),
```

we get 3 subgoals:

xlat_pe_in_pt_range.2.1.1.1:

```

{-1} pe_in_pdir_range?[Physical_memory, pm_phy]
      (state(read_data[Physical_memory, Pdbr_type](pm_phy, pdbr_data_type)(PDBR)(s'_1),
            restrict[Address, Memory_Address_4G, boolean]((pm'ro_addr ∪ pm'rw_addr)))
      (xlat_idx[Physical_memory, pm_phy]
        (pdir_lvl,
         data[Physical_memory, Pdbr_type]
           (read_data[Physical_memory, Pdbr_type]
             (pm_phy, pdbr_data_type)(PDBR)(s'_1)'base_addr,
            addr')))
{-2} pm'states
      (state(read_data[Physical_memory, Pdbr_type](pm_phy, pdbr_data_type)(PDBR)(s'_1)))
{-3} OK?(read_data(pm_phy, pdbr_data_type)(PDBR)(s'_1))
{-4} ∀ (s: (pm'states)): OK?(read_data(pm_phy, pdbr_data_type)(PDBR)(s))
{-5} pe_in_pdir_range?(s'_2,
      restrict[Address, Memory_Address_4G, boolean]
        ((pm'ro_addr ∪ pm'rw_addr)))
=
pe_in_pdir_range?(state(read_data[Physical_memory, Pdbr_type]
      (pm_phy, pdbr_data_type)(PDBR)(s'_1),
      restrict[Address, Memory_Address_4G, boolean]
        ((pm'ro_addr ∪ pm'rw_addr))))
{-6} (λ (phy_a: Memory_Address_4G):
      ∃ (pde:
        (pe_in_pdir_range?(s'_2,
          restrict[Address, Memory_Address_4G, boolean]
            ((pm'ro_addr ∪ pm'rw_addr))))):
      ∃ (lin_a:
        (restrict[Address, Memory_Address_4G, boolean]
          ((pm'ro_addr ∪ pm'rw_addr)))):
      OK?(read_data(pm_phy, paging_data_type(pdir_lvl))(pde)(s'_2)) ∧
      pde_pt?(pdir(data(read_data(pm_phy, paging_data_type(pdir_lvl))(pde)
        (s'_2))))
      ∧
      xlat_idx(ptab_lvl,
        base(data(read_data(pm_phy, paging_data_type(pdir_lvl))(pde)
          (s'_2))),
        lin_a)
      = phy_a)
=
(λ (phy_a: Memory_Address_4G):
  ∃ (pde:
    (pe_in_pdir_range?(state(read_data[Physical_memory, Pdbr_type]
      (pm_phy, pdbr_data_type)(PDBR)(s'_1),
      restrict[Address, Memory_Address_4G, boolean]
        ((pm'ro_addr ∪ pm'rw_addr))))):
  ∃ (lin_a:
    (restrict[Address, Memory_Address_4G, boolean]
      ((pm'ro_addr ∪ pm'rw_addr)))):
  OK?(read_data(pm_phy, paging_data_type(pdir_lvl))(pde)
    (state(read_data[Physical_memory, Pdbr_type]
      (pm_phy, pdbr_data_type)(PDBR)(s'_1))))
  ∧
  pde_pt?(pdir(data(read_data(pm_phy, paging_data_type(pdir_lvl))(pde)
    (state(read_data[Physical_memory, Pdbr_type]
      (pm_phy, pdbr_data_type)(PDBR)
        (s'_1))))))
  ∧
  xlat_idx(ptab_lvl,
    base(data(read_data(pm_phy, paging_data_type(pdir_lvl))(pde)
      (state(read_data[Physical_memory, Pdbr_type]
        (pm_phy, pdbr_data_type)(PDBR)
          (s'_1))))),
    lin_a)
  = phy_a)

```


C Proof scripts

Instantiating the top quantifier in 3 with the terms: `(xlat_idx(pdir_lvl, data(read_data(pm_phy, pdir_data_type)(PDBR)(s1!1))'base_addr, addr!1))`,

we get 2 subgoals:

xlat_pe_in_pt_range.2.1.1.1.1:

```

{-1} pe_in_pdir_range?[Physical_memory, pm_phy]
      (state(read_data[Physical_memory, Pdbr_type](pm_phy, pdbr_data_type)(PDBR)(s'_1),
            restrict[Address, Memory_Address_4G, boolean]((pm'ro_addr ∪ pm'rw_addr)))
      (xlat_idx[Physical_memory, pm_phy]
        (pdir_lvl,
         data[Physical_memory, Pdbr_type]
           (read_data[Physical_memory, Pdbr_type]
             (pm_phy, pdbr_data_type)(PDBR)(s'_1))'base_addr,
          addr'))
{-2} pm'states
      (state(read_data[Physical_memory, Pdbr_type](pm_phy, pdbr_data_type)(PDBR)(s'_1)))
{-3} OK?(read_data(pm_phy, pdbr_data_type)(PDBR)(s'_1))
{-4} ∀ (s: (pm'states)): OK?(read_data(pm_phy, pdbr_data_type)(PDBR)(s))
{-5} pe_in_pdir_range?(s'_2,
      restrict[Address, Memory_Address_4G, boolean]
        ((pm'ro_addr ∪ pm'rw_addr)))
=
pe_in_pdir_range?(state(read_data[Physical_memory, Pdbr_type]
      (pm_phy, pdbr_data_type)(PDBR)(s'_1),
      restrict[Address, Memory_Address_4G, boolean]
        ((pm'ro_addr ∪ pm'rw_addr))))
{-6} (λ (phy_a: Memory_Address_4G):
      ∃ (pde:
        (pe_in_pdir_range?(s'_2,
          restrict[Address, Memory_Address_4G, boolean]
            ((pm'ro_addr ∪ pm'rw_addr))))):
      ∃ (lin_a:
        (restrict[Address, Memory_Address_4G, boolean]
          ((pm'ro_addr ∪ pm'rw_addr)))):
      OK?(read_data(pm_phy, paging_data_type(pdir_lvl))(pde)(s'_2)) ∧
      pde_pt?(pdir(data(read_data(pm_phy, paging_data_type(pdir_lvl))(pde)
        (s'_2))))
      ∧
      xlat_idx(ptab_lvl,
        base(data(read_data(pm_phy, paging_data_type(pdir_lvl))(pde)
          (s'_2))),
        lin_a)
      = phy_a)
=
(λ (phy_a: Memory_Address_4G):
  ∃ (pde:
    (pe_in_pdir_range?(state(read_data[Physical_memory, Pdbr_type]
      (pm_phy, pdbr_data_type)(PDBR)(s'_1),
      restrict[Address, Memory_Address_4G, boolean]
        ((pm'ro_addr ∪ pm'rw_addr))))):
  ∃ (lin_a:
    (restrict[Address, Memory_Address_4G, boolean]
      ((pm'ro_addr ∪ pm'rw_addr)))):
  OK?(read_data(pm_phy, paging_data_type(pdir_lvl))(pde)
    (state(read_data[Physical_memory, Pdbr_type]
      (pm_phy, pdbr_data_type)(PDBR)(s'_1))))
  ∧
  pde_pt?(pdir(data(read_data(pm_phy, paging_data_type(pdir_lvl))(pde)
    (state(read_data[Physical_memory, Pdbr_type]
      (pm_phy, pdbr_data_type)(PDBR)
        (s'_1))))))
  ∧
  xlat_idx(ptab_lvl,
    base(data(read_data(pm_phy, paging_data_type(pdir_lvl))(pde)
      (state(read_data[Physical_memory, Pdbr_type]
        (pm_phy, pdbr_data_type)(PDBR)
          (s'_1))))),
    lin_a)
  = phy_a)

```

C Proof scripts

Instantiating the top quantifier in 3 with the terms: (addr!1),

we get 2 subgoals:

xlat_pe_in_pt_range.2.1.1.1.1.1:

```

{-1} pe_in_pdir_range?[Physical_memory, pm_phy]
      (state(read_data[Physical_memory, Pdbr_type](pm_phy, pdbr_data_type)(PDBR)(s'_1),
            restrict[Address, Memory_Address_4G, boolean]((pm'ro_addr ∪ pm'rw_addr)))
      (xlat_idx[Physical_memory, pm_phy]
        (pdir_lvl,
         data[Physical_memory, Pdbr_type]
           (read_data[Physical_memory, Pdbr_type]
             (pm_phy, pdbr_data_type)(PDBR)(s'_1))'base_addr,
          addr'))
{-2} pm'states
      (state(read_data[Physical_memory, Pdbr_type](pm_phy, pdbr_data_type)(PDBR)(s'_1)))
{-3} OK?(read_data(pm_phy, pdbr_data_type)(PDBR)(s'_1))
{-4} ∀ (s: (pm'states)): OK?(read_data(pm_phy, pdbr_data_type)(PDBR)(s))
{-5} pe_in_pdir_range?(s'_2,
      restrict[Address, Memory_Address_4G, boolean]
        ((pm'ro_addr ∪ pm'rw_addr)))
=
pe_in_pdir_range?(state(read_data[Physical_memory, Pdbr_type]
      (pm_phy, pdbr_data_type)(PDBR)(s'_1),
      restrict[Address, Memory_Address_4G, boolean]
        ((pm'ro_addr ∪ pm'rw_addr))))
{-6} (λ (phy_a: Memory_Address_4G):
      ∃ (pde:
        (pe_in_pdir_range?(s'_2,
          restrict[Address, Memory_Address_4G, boolean]
            ((pm'ro_addr ∪ pm'rw_addr))))):
      ∃ (lin_a:
        (restrict[Address, Memory_Address_4G, boolean]
          ((pm'ro_addr ∪ pm'rw_addr)))):
      OK?(read_data(pm_phy, paging_data_type(pdir_lvl))(pde)(s'_2)) ∧
      pde_pt?(pdir(data(read_data(pm_phy, paging_data_type(pdir_lvl))(pde)
        (s'_2))))
      ∧
      xlat_idx(ptab_lvl,
        base(data(read_data(pm_phy, paging_data_type(pdir_lvl))(pde)
          (s'_2))),
        lin_a)
      = phy_a)
=
(λ (phy_a: Memory_Address_4G):
  ∃ (pde:
    (pe_in_pdir_range?(state(read_data[Physical_memory, Pdbr_type]
      (pm_phy, pdbr_data_type)(PDBR)(s'_1),
      restrict[Address, Memory_Address_4G, boolean]
        ((pm'ro_addr ∪ pm'rw_addr))))):
  ∃ (lin_a:
    (restrict[Address, Memory_Address_4G, boolean]
      ((pm'ro_addr ∪ pm'rw_addr)))):
  OK?(read_data(pm_phy, paging_data_type(pdir_lvl))(pde)
    (state(read_data[Physical_memory, Pdbr_type]
      (pm_phy, pdbr_data_type)(PDBR)(s'_1))))
  ∧
  pde_pt?(pdir(data(read_data(pm_phy, paging_data_type(pdir_lvl))(pde)
    (state(read_data[Physical_memory, Pdbr_type]
      (pm_phy, pdbr_data_type)(PDBR)
        (s'_1))))))
  ∧
  xlat_idx(ptab_lvl,
    base(data(read_data(pm_phy, paging_data_type(pdir_lvl))(pde)
      (state(read_data[Physical_memory, Pdbr_type]
        (pm_phy, pdbr_data_type)(PDBR)
          (s'_1))))),
    lin_a)
  = phy_a)

```

C Proof scripts

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 3 subgoals:

xlat_pe_in_pt_range.2.1.1.1.1.1.1:

```

{-1} pe_in_pdir_range?[Physical_memory, pm_phy]
      (state(read_data[Physical_memory, Pdbr_type](pm_phy, pdbr_data_type)(PDBR)(s'_1),
            restrict[Address, Memory_Address_4G, boolean]((pm'ro_addr ∪ pm'rw_addr)))
      (xlat_idx[Physical_memory, pm_phy]
        (pdir_lvl,
         data[Physical_memory, Pdbr_type]
           (read_data[Physical_memory, Pdbr_type]
             (pm_phy, pdbr_data_type)(PDBR)(s'_1))'base_addr,
          addr'))
{-2} pm'states
      (state(read_data[Physical_memory, Pdbr_type](pm_phy, pdbr_data_type)(PDBR)(s'_1)))
{-3} OK?(read_data(pm_phy, pdbr_data_type)(PDBR)(s'_1))
{-4} ∀ (s: (pm'states)): OK?(read_data(pm_phy, pdbr_data_type)(PDBR)(s))
{-5} pe_in_pdir_range?(s'_2,
      restrict[Address, Memory_Address_4G, boolean]
        ((pm'ro_addr ∪ pm'rw_addr)))
=
pe_in_pdir_range?(state(read_data[Physical_memory, Pdbr_type]
      (pm_phy, pdbr_data_type)(PDBR)(s'_1),
      restrict[Address, Memory_Address_4G, boolean]
        ((pm'ro_addr ∪ pm'rw_addr))))
{-6} (λ (phy_a: Memory_Address_4G):
      ∃ (pde:
        (pe_in_pdir_range?(s'_2,
          restrict[Address, Memory_Address_4G, boolean]
            ((pm'ro_addr ∪ pm'rw_addr))))):
      ∃ (lin_a:
        (restrict[Address, Memory_Address_4G, boolean]
          ((pm'ro_addr ∪ pm'rw_addr)))):
      OK?(read_data(pm_phy, paging_data_type(pdir_lvl))(pde)(s'_2)) ∧
      pde_pt?(pdir(data(read_data(pm_phy, paging_data_type(pdir_lvl))(pde)
        (s'_2))))
      ∧
      xlat_idx(ptab_lvl,
        base(data(read_data(pm_phy, paging_data_type(pdir_lvl))(pde)
          (s'_2))),
        lin_a)
      = phy_a)
=
(λ (phy_a: Memory_Address_4G):
  ∃ (pde:
    (pe_in_pdir_range?(state(read_data[Physical_memory, Pdbr_type]
      (pm_phy, pdbr_data_type)(PDBR)(s'_1),
      restrict[Address, Memory_Address_4G, boolean]
        ((pm'ro_addr ∪ pm'rw_addr))))):
  ∃ (lin_a:
    (restrict[Address, Memory_Address_4G, boolean]
      ((pm'ro_addr ∪ pm'rw_addr)))):
  OK?(read_data(pm_phy, paging_data_type(pdir_lvl))(pde)
    (state(read_data[Physical_memory, Pdbr_type]
      (pm_phy, pdbr_data_type)(PDBR)(s'_1))))
  ∧
  pde_pt?(pdir(data(read_data(pm_phy, paging_data_type(pdir_lvl))(pde)
    (state(read_data[Physical_memory, Pdbr_type]
      (pm_phy, pdbr_data_type)(PDBR)
        (s'_1))))))
  ∧
  xlat_idx(ptab_lvl,
    base(data(read_data(pm_phy, paging_data_type(pdir_lvl))(pde)
      (state(read_data[Physical_memory, Pdbr_type]
        (pm_phy, pdbr_data_type)(PDBR)
          (s'_1))))),
    lin_a)
  = phy_a)

```

C Proof scripts

Using lemma `translate_result_as_read`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `xlat_pe_in_pt_range.2.1.1.1.1.1.1`.

xlat_pe_in_pt_range.2.1.1.1.1.1.2:

```

{-1} pe_in_pdir_range?[Physical_memory, pm_phy]
      (state(read_data[Physical_memory, Pdbr_type](pm_phy, pdbr_data_type)(PDBR)(s'_1),
            restrict[Address, Memory_Address_4G, boolean]((pm'ro_addr ∪ pm'rw_addr)))
      (xlat_idx[Physical_memory, pm_phy]
        (pdir_lvl,
         data[Physical_memory, Pdbr_type]
           (read_data[Physical_memory, Pdbr_type]
             (pm_phy, pdbr_data_type)(PDBR)(s'_1))'base_addr,
          addr'))
{-2} pm'states
      (state(read_data[Physical_memory, Pdbr_type](pm_phy, pdbr_data_type)(PDBR)(s'_1)))
{-3} OK?(read_data(pm_phy, pdbr_data_type)(PDBR)(s'_1))
{-4} ∀ (s: (pm'states)): OK?(read_data(pm_phy, pdbr_data_type)(PDBR)(s))
{-5} pe_in_pdir_range?(s'_2,
      restrict[Address, Memory_Address_4G, boolean]
        ((pm'ro_addr ∪ pm'rw_addr)))
=
pe_in_pdir_range?(state(read_data[Physical_memory, Pdbr_type]
      (pm_phy, pdbr_data_type)(PDBR)(s'_1),
      restrict[Address, Memory_Address_4G, boolean]
        ((pm'ro_addr ∪ pm'rw_addr))))
{-6} (λ (phy_a: Memory_Address_4G):
      ∃ (pde:
        (pe_in_pdir_range?(s'_2,
          restrict[Address, Memory_Address_4G, boolean]
            ((pm'ro_addr ∪ pm'rw_addr))))):
      ∃ (lin_a:
        (restrict[Address, Memory_Address_4G, boolean]
          ((pm'ro_addr ∪ pm'rw_addr)))):
      OK?(read_data(pm_phy, paging_data_type(pdir_lvl))(pde)(s'_2)) ∧
      pde_pt?(pdir(data(read_data(pm_phy, paging_data_type(pdir_lvl))(pde)
        (s'_2))))
      ∧
      xlat_idx(ptab_lvl,
        base(data(read_data(pm_phy, paging_data_type(pdir_lvl))(pde)
          (s'_2))),
        lin_a)
      = phy_a)
=
(λ (phy_a: Memory_Address_4G):
  ∃ (pde:
    (pe_in_pdir_range?(state(read_data[Physical_memory, Pdbr_type]
      (pm_phy, pdbr_data_type)(PDBR)(s'_1),
      restrict[Address, Memory_Address_4G, boolean]
        ((pm'ro_addr ∪ pm'rw_addr))))):
  ∃ (lin_a:
    (restrict[Address, Memory_Address_4G, boolean]
      ((pm'ro_addr ∪ pm'rw_addr)))):
  OK?(read_data(pm_phy, paging_data_type(pdir_lvl))(pde)
    (state(read_data[Physical_memory, Pdbr_type]
      (pm_phy, pdbr_data_type)(PDBR)(s'_1))))
  ∧
  pde_pt?(pdir(data(read_data(pm_phy, paging_data_type(pdir_lvl))(pde)
    (state(read_data[Physical_memory, Pdbr_type]
      (pm_phy, pdbr_data_type)(PDBR)
        (s'_1))))))
  ∧
  xlat_idx(ptab_lvl,
    base(data(read_data(pm_phy, paging_data_type(pdir_lvl))(pde)
      (state(read_data[Physical_memory, Pdbr_type]
        (pm_phy, pdbr_data_type)(PDBR)
          (s'_1))))),
    lin_a)
  = phy_a)

```


C Proof scripts

Expanding the definition of `translate`,

Installing automatic rewrites from: (`##! fatal_result! raise_fault! exception_result! ok_result!`)

Expanding the definition of `paging_type?`,

Expanding the definition of `pdir_entry?`,

Expanding the definition of `present?`,

Expanding the definition of `is_leaf?`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `xlat_pe_in_pt_range.2.1.1.1.1.1.2`.

xlat_pe_in_pt_range.2.1.1.1.1.1.3:

```

{-1} pe_in_pdir_range?[Physical_memory, pm_phy]
      (state(read_data[Physical_memory, Pdbr_type](pm_phy, pdbr_data_type)(PDBR)(s'_1)),
       restrict[Address, Memory_Address_4G, boolean]((pm'ro_addr ∪ pm'rw_addr)))
      (xlat_idx[Physical_memory, pm_phy]
       (pdir_lvl,
        data[Physical_memory, Pdbr_type]
          (read_data[Physical_memory, Pdbr_type]
            (pm_phy, pdbr_data_type)(PDBR)(s'_1))'base_addr,
        addr'))
{-2} pm'states
      (state(read_data[Physical_memory, Pdbr_type](pm_phy, pdbr_data_type)(PDBR)(s'_1)))
{-3} OK?(read_data(pm_phy, pdbr_data_type)(PDBR)(s'_1))
{-4} ∀ (s: (pm'states)): OK?(read_data(pm_phy, pdbr_data_type)(PDBR)(s))
{-5} pe_in_pdir_range?(s'_2,
      restrict[Address, Memory_Address_4G, boolean]
        ((pm'ro_addr ∪ pm'rw_addr)))
=
pe_in_pdir_range?(state(read_data[Physical_memory, Pdbr_type]
      (pm_phy, pdbr_data_type)(PDBR)(s'_1)),
      restrict[Address, Memory_Address_4G, boolean]
        ((pm'ro_addr ∪ pm'rw_addr)))
{-6} (λ (phy_a: Memory_Address_4G):
      ∃ (pde:
        (pe_in_pdir_range?(s'_2,
          restrict[Address, Memory_Address_4G, boolean]
            ((pm'ro_addr ∪ pm'rw_addr))))):
      ∃ (lin_a:
        (restrict[Address, Memory_Address_4G, boolean]
          ((pm'ro_addr ∪ pm'rw_addr)))):
      OK?(read_data(pm_phy, paging_data_type(pdir_lvl))(pde)(s'_2)) ∧
      pde_pt?(pdir(data(read_data(pm_phy, paging_data_type(pdir_lvl))(pde)
        (s'_2))))
      ∧
      xlat_idx(ptab_lvl,
        base(data(read_data(pm_phy, paging_data_type(pdir_lvl))(pde)
          (s'_2))),
        lin_a)
      = phy_a)
=
(λ (phy_a: Memory_Address_4G):
      ∃ (pde:
        (pe_in_pdir_range?(state(read_data[Physical_memory, Pdbr_type]
          (pm_phy, pdbr_data_type)(PDBR)(s'_1)),
          restrict[Address, Memory_Address_4G, boolean]
            ((pm'ro_addr ∪ pm'rw_addr))))):
      ∃ (lin_a:
        (restrict[Address, Memory_Address_4G, boolean]
          ((pm'ro_addr ∪ pm'rw_addr)))):
      OK?(read_data(pm_phy, paging_data_type(pdir_lvl))(pde)
        (state(read_data[Physical_memory, Pdbr_type]
          (pm_phy, pdbr_data_type)(PDBR)(s'_1))))
      ∧
      pde_pt?(pdir(data(read_data(pm_phy, paging_data_type(pdir_lvl))(pde)
        (state(read_data[Physical_memory, Pdbr_type]
          (pm_phy, pdbr_data_type)(PDBR)
            (s'_1))))))
      ∧
      xlat_idx(ptab_lvl,
        base(data(read_data(pm_phy, paging_data_type(pdir_lvl))(pde)
          (state(read_data[Physical_memory, Pdbr_type]
            (pm_phy, pdbr_data_type)(PDBR)
              (s'_1))))),
        lin_a)
      = phy_a)

```

C Proof scripts

Keeping (-20 3) and hiding *,

Expanding the definition of translate,

Installing automatic rewrites from: (##! fatal_result! raise_fault! exception_result! ok_result!)

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `xlat_pe_in_pt_range.2.1.1.1.1.1.3`.

xlat_pe_in_pt_range.2.1.1.1.1.2:

```

{-1} pe_in_pdir_range?[Physical_memory, pm_phy]
      (state(read_data[Physical_memory, Pdbr_type](pm_phy, pdbr_data_type)(PDBR)(s'_1),
            restrict[Address, Memory_Address_4G, boolean]((pm'ro_addr ∪ pm'rw_addr)))
      (xlat_idx[Physical_memory, pm_phy]
        (pdir_lvl,
         data[Physical_memory, Pdbr_type]
           (read_data[Physical_memory, Pdbr_type]
             (pm_phy, pdbr_data_type)(PDBR)(s'_1))'base_addr,
          addr'))
{-2} pm'states
      (state(read_data[Physical_memory, Pdbr_type](pm_phy, pdbr_data_type)(PDBR)(s'_1)))
{-3} OK?(read_data(pm_phy, pdbr_data_type)(PDBR)(s'_1))
{-4} ∀ (s: (pm'states)): OK?(read_data(pm_phy, pdbr_data_type)(PDBR)(s))
{-5} pe_in_pdir_range?(s'_2,
      restrict[Address, Memory_Address_4G, boolean]
        ((pm'ro_addr ∪ pm'rw_addr)))
=
pe_in_pdir_range?(state(read_data[Physical_memory, Pdbr_type]
      (pm_phy, pdbr_data_type)(PDBR)(s'_1),
      restrict[Address, Memory_Address_4G, boolean]
        ((pm'ro_addr ∪ pm'rw_addr))))
{-6} (λ (phy_a: Memory_Address_4G):
      ∃ (pde:
        (pe_in_pdir_range?(s'_2,
          restrict[Address, Memory_Address_4G, boolean]
            ((pm'ro_addr ∪ pm'rw_addr))))):
      ∃ (lin_a:
        (restrict[Address, Memory_Address_4G, boolean]
          ((pm'ro_addr ∪ pm'rw_addr)))):
      OK?(read_data(pm_phy, paging_data_type(pdir_lvl))(pde)(s'_2)) ∧
      pde_pt?(pdir(data(read_data(pm_phy, paging_data_type(pdir_lvl))(pde)
        (s'_2))))
      ∧
      xlat_idx(ptab_lvl,
        base(data(read_data(pm_phy, paging_data_type(pdir_lvl))(pde)
          (s'_2))),
        lin_a)
      = phy_a)
=
(λ (phy_a: Memory_Address_4G):
  ∃ (pde:
    (pe_in_pdir_range?(state(read_data[Physical_memory, Pdbr_type]
      (pm_phy, pdbr_data_type)(PDBR)(s'_1),
      restrict[Address, Memory_Address_4G, boolean]
        ((pm'ro_addr ∪ pm'rw_addr))))):
  ∃ (lin_a:
    (restrict[Address, Memory_Address_4G, boolean]
      ((pm'ro_addr ∪ pm'rw_addr)))):
  OK?(read_data(pm_phy, paging_data_type(pdir_lvl))(pde)
    (state(read_data[Physical_memory, Pdbr_type]
      (pm_phy, pdbr_data_type)(PDBR)(s'_1))))
  ∧
  pde_pt?(pdir(data(read_data(pm_phy, paging_data_type(pdir_lvl))(pde)
    (state(read_data[Physical_memory, Pdbr_type]
      (pm_phy, pdbr_data_type)(PDBR)
        (s'_1))))))
  ∧
  xlat_idx(ptab_lvl,
    base(data(read_data(pm_phy, paging_data_type(pdir_lvl))(pde)
      (state(read_data[Physical_memory, Pdbr_type]
        (pm_phy, pdbr_data_type)(PDBR)
          (s'_1))))),
    lin_a)
  = phy_a)

```

C Proof scripts

Expanding the definition of restrict,

which is trivially true.

This completes the proof of `xlat_pe_in_pt_range.2.1.1.1.1.2`.

xlat_pe_in_pt_range.2.1.1.1.2:

```

{-1} pe_in_pdir_range?[Physical_memory, pm_phy]
      (state(read_data[Physical_memory, Pdbr_type](pm_phy, pdbr_data_type)(PDBR)(s'_1),
            restrict[Address, Memory_Address_4G, boolean]((pm'ro_addr ∪ pm'rw_addr)))
      (xlat_idx[Physical_memory, pm_phy]
        (pdir_lvl,
         data[Physical_memory, Pdbr_type]
           (read_data[Physical_memory, Pdbr_type]
             (pm_phy, pdbr_data_type)(PDBR)(s'_1))'base_addr,
          addr'))
{-2} pm'states
      (state(read_data[Physical_memory, Pdbr_type](pm_phy, pdbr_data_type)(PDBR)(s'_1)))
{-3} OK?(read_data(pm_phy, pdbr_data_type)(PDBR)(s'_1))
{-4} ∀ (s: (pm'states)): OK?(read_data(pm_phy, pdbr_data_type)(PDBR)(s))
{-5} pe_in_pdir_range?(s'_2,
      restrict[Address, Memory_Address_4G, boolean]
        ((pm'ro_addr ∪ pm'rw_addr)))
=
pe_in_pdir_range?(state(read_data[Physical_memory, Pdbr_type]
      (pm_phy, pdbr_data_type)(PDBR)(s'_1),
      restrict[Address, Memory_Address_4G, boolean]
        ((pm'ro_addr ∪ pm'rw_addr))))
{-6} (λ (phy_a: Memory_Address_4G):
      ∃ (pde:
        (pe_in_pdir_range?(s'_2,
          restrict[Address, Memory_Address_4G, boolean]
            ((pm'ro_addr ∪ pm'rw_addr))))):
      ∃ (lin_a:
        (restrict[Address, Memory_Address_4G, boolean]
          ((pm'ro_addr ∪ pm'rw_addr)))):
      OK?(read_data(pm_phy, paging_data_type(pdir_lvl))(pde)(s'_2)) ∧
      pde_pt?(pdir(data(read_data(pm_phy, paging_data_type(pdir_lvl))(pde)
        (s'_2))))
      ∧
      xlat_idx(ptab_lvl,
        base(data(read_data(pm_phy, paging_data_type(pdir_lvl))(pde)
          (s'_2))),
        lin_a)
      = phy_a)
=
(λ (phy_a: Memory_Address_4G):
  ∃ (pde:
    (pe_in_pdir_range?(state(read_data[Physical_memory, Pdbr_type]
      (pm_phy, pdbr_data_type)(PDBR)(s'_1),
      restrict[Address, Memory_Address_4G, boolean]
        ((pm'ro_addr ∪ pm'rw_addr))))):
  ∃ (lin_a:
    (restrict[Address, Memory_Address_4G, boolean]
      ((pm'ro_addr ∪ pm'rw_addr)))):
  OK?(read_data(pm_phy, paging_data_type(pdir_lvl))(pde)
    (state(read_data[Physical_memory, Pdbr_type]
      (pm_phy, pdbr_data_type)(PDBR)(s'_1))))
  ∧
  pde_pt?(pdir(data(read_data(pm_phy, paging_data_type(pdir_lvl))(pde)
    (state(read_data[Physical_memory, Pdbr_type]
      (pm_phy, pdbr_data_type)(PDBR)
        (s'_1))))))
  ∧
  xlat_idx(ptab_lvl,
    base(data(read_data(pm_phy, paging_data_type(pdir_lvl))(pde)
      (state(read_data[Physical_memory, Pdbr_type]
        (pm_phy, pdbr_data_type)(PDBR)
          (s'_1))))),
    lin_a)
  = phy_a)

```

C Proof scripts

which is trivially true.

This completes the proof of `xlat_pe_in_pt_range.2.1.1.1.2`.

xlat_pe_in_pt_range.2.1.1.2:

```

{-1} pm' states
      (state(read_data[Physical_memory, Pdbr_type](pm_phy, pdbr_data_type)(PDBR)(s'_1)))
{-2} OK?(read_data(pm_phy, pdbr_data_type)(PDBR)(s'_1))
{-3}  $\forall (s: (pm' states)): OK?(read_data(pm_phy, pdbr_data_type)(PDBR)(s))$ 
{-4} pe_in_pdir_range?(s'_2,
      restrict[Address, Memory_Address_4G, boolean]
      ((pm'ro_addr  $\cup$  pm'rw_addr)))
=
pe_in_pdir_range?(state(read_data[Physical_memory, Pdbr_type]
      (pm_phy, pdbr_data_type)(PDBR)(s'_1),
      restrict[Address, Memory_Address_4G, boolean]
      ((pm'ro_addr  $\cup$  pm'rw_addr))))
{-5} ( $\lambda$  (phy_a: Memory_Address_4G):
       $\exists$  (pde:
      (pe_in_pdir_range?(s'_2,
      restrict[Address, Memory_Address_4G, boolean]
      ((pm'ro_addr  $\cup$  pm'rw_addr))))))
       $\exists$  (lin_a:
      (restrict[Address, Memory_Address_4G, boolean]
      ((pm'ro_addr  $\cup$  pm'rw_addr)))):
      OK?(read_data(pm_phy, paging_data_type(pdir_lvl))(pde)(s'_2))  $\wedge$ 
      pde_pt?(pdir(data(read_data(pm_phy, paging_data_type(pdir_lvl))(pde)
      (s'_2))))
       $\wedge$ 
      xlat_idx(ptab_lvl,
      base(data(read_data(pm_phy, paging_data_type(pdir_lvl))(pde)
      (s'_2))),
      lin_a)
      = phy_a)
=
( $\lambda$  (phy_a: Memory_Address_4G):
       $\exists$  (pde:
      (pe_in_pdir_range?(state(read_data[Physical_memory, Pdbr_type]
      (pm_phy, pdbr_data_type)(PDBR)(s'_1),
      restrict[Address, Memory_Address_4G, boolean]
      ((pm'ro_addr  $\cup$  pm'rw_addr))))))
       $\exists$  (lin_a:
      (restrict[Address, Memory_Address_4G, boolean]
      ((pm'ro_addr  $\cup$  pm'rw_addr)))):
      OK?(read_data(pm_phy, paging_data_type(pdir_lvl))(pde)
      (state(read_data[Physical_memory, Pdbr_type]
      (pm_phy, pdbr_data_type)(PDBR)(s'_1))))
       $\wedge$ 
      pde_pt?(pdir(data(read_data(pm_phy, paging_data_type(pdir_lvl))(pde)
      (state(read_data[Physical_memory, Pdbr_type]
      (pm_phy, pdbr_data_type)(PDBR)
      (s'_1))))))
       $\wedge$ 
      xlat_idx(ptab_lvl,
      base(data(read_data(pm_phy, paging_data_type(pdir_lvl))(pde)
      (state(read_data[Physical_memory, Pdbr_type]
      (pm_phy, pdbr_data_type)(PDBR)
      (s'_1))))),
      lin_a)
      = phy_a)

```

{-6} plain_memory?(pm_phy)

1727

{-7} pm' states = pm_phy' states

{-8} (pm'other_actions \subseteq pm_phy'other_actions)

{-9} (linear_resolve_register_transformers \subseteq pm_phy'other_actions)

{-10} $\forall (a: ((pm'ro_addr \cup pm'rw_addr))):$

Mem?(type_of(a)) \wedge 0 \leq offset(a) \wedge offset(a) < max_linear_offset

{-11} $\forall (s: (pm' states)):$
linear_blessed?(s, restrict[Address, Memory_Address_4G, boolean](pm'ro_addr),

C Proof scripts

Hiding formulas: 4,

Using lemma `xlat_pe_in_pdir_range`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `xlat_pe_in_pt_range.2.1.1.2`.

xlat_pe_in_pt_range.2.1.1.3:

```

{-1} pm' states
      (state(read_data[Physical_memory, Pdbr_type](pm_phy, pdbr_data_type)(PDBR)(s'_1)))
{-2} OK?(read_data(pm_phy, pdbr_data_type)(PDBR)(s'_1))
{-3}  $\forall (s: (pm' states)): OK?(read_data(pm_phy, pdbr_data_type)(PDBR)(s))$ 
{-4} pe_in_pdir_range?(s'_2,
      restrict[Address, Memory_Address_4G, boolean]
      ((pm'ro_addr  $\cup$  pm'rw_addr)))
=
pe_in_pdir_range?(state(read_data[Physical_memory, Pdbr_type]
      (pm_phy, pdbr_data_type)(PDBR)(s'_1),
      restrict[Address, Memory_Address_4G, boolean]
      ((pm'ro_addr  $\cup$  pm'rw_addr))))
{-5} ( $\lambda$  (phy_a: Memory_Address_4G):
       $\exists$  (pde:
        (pe_in_pdir_range?(s'_2,
          restrict[Address, Memory_Address_4G, boolean]
          ((pm'ro_addr  $\cup$  pm'rw_addr))))))
       $\exists$  (lin_a:
        (restrict[Address, Memory_Address_4G, boolean]
          ((pm'ro_addr  $\cup$  pm'rw_addr)))):
        OK?(read_data(pm_phy, paging_data_type(pdir_lvl))(pde)(s'_2))  $\wedge$ 
        pde_pt?(pdir(data(read_data(pm_phy, paging_data_type(pdir_lvl))(pde)
          (s'_2))))
           $\wedge$ 
          xlat_idx(ptab_lvl,
            base(data(read_data(pm_phy, paging_data_type(pdir_lvl))(pde)
              (s'_2))),
            lin_a)
          = phy_a)
      =
      ( $\lambda$  (phy_a: Memory_Address_4G):
         $\exists$  (pde:
          (pe_in_pdir_range?(state(read_data[Physical_memory, Pdbr_type]
            (pm_phy, pdbr_data_type)(PDBR)(s'_1),
            restrict[Address, Memory_Address_4G, boolean]
            ((pm'ro_addr  $\cup$  pm'rw_addr))))))
           $\exists$  (lin_a:
            (restrict[Address, Memory_Address_4G, boolean]
              ((pm'ro_addr  $\cup$  pm'rw_addr)))):
            OK?(read_data(pm_phy, paging_data_type(pdir_lvl))(pde)
              (state(read_data[Physical_memory, Pdbr_type]
                (pm_phy, pdbr_data_type)(PDBR)(s'_1))))))
           $\wedge$ 
          pde_pt?(pdir(data(read_data(pm_phy, paging_data_type(pdir_lvl))(pde)
            (state(read_data[Physical_memory, Pdbr_type]
              (pm_phy, pdbr_data_type)(PDBR)
              (s'_1))))))
             $\wedge$ 
            xlat_idx(ptab_lvl,
              base(data(read_data(pm_phy, paging_data_type(pdir_lvl))(pde)
                (state(read_data[Physical_memory, Pdbr_type]
                  (pm_phy, pdbr_data_type)(PDBR)
                  (s'_1))))),
              lin_a)
            = phy_a)

```

{-6} plain_memory?(pm_phy)

1729

{-7} pm' states = pm_phy' states

{-8} (pm'other_actions \subseteq pm_phy'other_actions){-9} (linear_resolve_register_transformers \subseteq pm_phy'other_actions){-10} $\forall (a: ((pm'ro_addr \cup pm'rw_addr))):$ Mem?(type_of(a)) \wedge 0 \leq offset(a) \wedge offset(a) < max_linear_offset{-11} $\forall (s: (pm' states)):$

linear_blessed?(s, restrict[Address, Memory_Address_4G, boolean](pm'ro_addr),

C Proof scripts

Using lemma `xlat_idx_memory_address`,

Using lemma `xlat_idx_memory_address2`,

Installing automatic rewrites from: `max_linear < Mem <= min_linear`

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `xlat_pe_in_pt_range.2.1.1.3`.

`xlat_pe_in_pt_range.2.1.2`:

{-1}	<code>pm' states</code>
	<code>(state[read_data[Physical_memory, Pdbr_type](pm_phy, pdbr_data_type)(PDBR)(s'_1))</code>
{-2}	<code>OK?(read_data(pm_phy, pdbr_data_type)(PDBR)(s'_1))</code>
{-3}	<code>∀ (s: (pm' states)): OK?(read_data(pm_phy, pdbr_data_type)(PDBR)(s))</code>
{-4}	<code>plain_memory?(pm_phy)</code>
{-5}	<code>pm' states = pm_phy states</code>
{-6}	<code>(pm' other_actions ⊆ pm_phy other_actions)</code>
{-7}	<code>(linear_resolve_register_transformers ⊆ pm_phy other_actions)</code>
{-8}	<code>∀ (a: ((pm' ro_addr ∪ pm' rw_addr))):</code>
	<code>Mem?(type_of(a)) ∧ 0 ≤ offset(a) ∧ offset(a) < max_linear_offset</code>
{-9}	<code>∀ (s: (pm' states)):</code>
	<code>linear_blessed?(s, restrict[Address, Memory_Address_4G, boolean](pm' ro_addr),</code>
	<code>restrict[Address, Memory_Address_4G, boolean](pm' rw_addr))</code>
{-10}	<code>Mem?(addr' type_of)</code>
{-11}	<code>0 ≤ addr' offset</code>
{-12}	<code>addr' offset < max_linear_offset</code>
{-13}	<code>union(pm' ro_addr, pm' rw_addr)(addr')</code>
{-14}	<code>is_linear_plain_memory?(pm')</code>
{-15}	<code>pm' states(s'_1)</code>
{-16}	<code>pm' states(s'_2)</code>
{-17}	<code>OK?(translate(pdir_lvl, data(read_data(pm_phy, pdbr_data_type)(PDBR)(s'_1))'base_addr,</code>
	<code>addr', access', priv')</code>
	<code>(state(read_data(pm_phy, pdbr_data_type)(PDBR)(s'_1)))</code>
{1}	<code>pm' states</code>
	<code>(state[Physical_memory, Pdbr_type]</code>
	<code>(read_data[Physical_memory, Pdbr_type](pm_phy, pdbr_data_type)(PDBR)(s'_1))</code>
{2}	<code>data(translate(pdir_lvl, data(read_data(pm_phy, pdbr_data_type)(PDBR)(s'_1))'base_addr,</code>
	<code>addr', access', priv')</code>
	<code>(state(read_data(pm_phy, pdbr_data_type)(PDBR)(s'_1))))'1</code>
{3}	<code>ptab_lvl = pdir_lvl</code>
{4}	<code>pe_in_ptab_range?(s'_2,</code>
	<code>restrict[Address, Memory_Address_4G, boolean]</code>
	<code>((pm' ro_addr ∪ pm' rw_addr))</code>
	<code>(xlat_idx(ptab_lvl,</code>
	<code>data(translate(pdir_lvl,</code>
	<code>data(read_data(pm_phy, pdbr_data_type)</code>
	<code>(PDBR)</code>
	<code>(s'_1))'base_addr,</code>
	<code>addr', access', priv')</code>
	<code>(state(read_data(pm_phy, pdbr_data_type)</code>
	<code>(PDBR)</code>
	<code>(s'_1))))'2,</code>
	<code>addr'))</code>

which is trivially true.

This completes the proof of `xlat_pe_in_pt_range.2.1.2`.

xlat_pe_in_pt_range.2.2:

<pre> {-1} OK?(read_data(pm_phy, pdbr_data_type)(PDBR)(s'_1)) {-2} $\forall (s: (pm' \text{states})): \text{OK?}(\text{read_data}(\text{pm_phy}, \text{pdbr_data_type})(\text{PDBR})(s))$ {-3} $\forall (s_1, s_2: (pm' \text{states})):$ pe_in_pdir_range?(s_1, restrict[Address, Memory_Address_4G, boolean] ((pm'ro_addr \cup pm'rw_addr))) = pe_in_pdir_range?(s_2, restrict[Address, Memory_Address_4G, boolean] ((pm'ro_addr \cup pm'rw_addr))) \wedge pe_in_ptab_range?(s_1, restrict[Address, Memory_Address_4G, boolean] ((pm'ro_addr \cup pm'rw_addr))) = pe_in_ptab_range?(s_2, restrict[Address, Memory_Address_4G, boolean] ((pm'ro_addr \cup pm'rw_addr))) {-4} plain_memory?(pm_phy) {-5} pm'states = pm_phy'states {-6} (pm'other_actions \subseteq pm_phy'other_actions) {-7} (linear_resolve_register_transformers \subseteq pm_phy'other_actions) {-8} $\forall (a: ((pm'ro_addr \cup pm'rw_addr))):$ Mem?(type_of(a)) \wedge 0 \leq offset(a) \wedge offset(a) < max_linear_offset {-9} $\forall (s: (pm' \text{states})):$ linear_blessed?(s, restrict[Address, Memory_Address_4G, boolean](pm'ro_addr), restrict[Address, Memory_Address_4G, boolean](pm'rw_addr)) {-10} Mem?(addr' type_of) {-11} 0 \leq addr' offset {-12} addr' offset < max_linear_offset {-13} union(pm'ro_addr, pm'rw_addr)(addr') {-14} is_linear_plain_memory?(pm') {-15} pm'states(s'_1) {-16} pm'states(s'_2) {-17} OK?(translate(pdir_lvl, data(read_data(pm_phy, pdbr_data_type)(PDBR)(s'_1))'base_addr, addr', access', priv') (state(read_data(pm_phy, pdbr_data_type)(PDBR)(s'_1)))) </pre>	<pre> {1} pm'states (state(read_data[Physical_memory, Pdbr_type](pm_phy, pdbr_data_type)(PDBR)(s'_1))) {2} data(translate(pdir_lvl, data(read_data(pm_phy, pdbr_data_type)(PDBR)(s'_1))'base_addr, addr', access', priv') (state(read_data(pm_phy, pdbr_data_type)(PDBR)(s'_1))))'1 {3} ptab_lvl = pdir_lvl {4} pe_in_ptab_range?(s'_2, restrict[Address, Memory_Address_4G, boolean] ((pm'ro_addr \cup pm'rw_addr))) (xlat_idx(ptab_lvl, data(translate(pdir_lvl, data(read_data(pm_phy, pdbr_data_type) (PDBR) (s'_1))'base_addr, addr', access', priv') (state(read_data(pm_phy, pdbr_data_type) (PDBR) (s'_1))))'2, addr')) </pre>
---	---

C Proof scripts

Using lemma `pm_unchanged_singleton_linear_resolve_reg_transformers`,

Expanding the definition of `linear_resolve_register_transformers`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Using lemma `unchanged_memory_invariant_invariant`,

Using lemma `expr_transformer_invariant_next_ok`,

Using lemma `pm_states`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `xlat_pe_in_pt_range.2.2`.

xlat_pe_in_pt_range.2.3:

{-1}	$\forall (s: (pm' \text{states})): \text{OK?}(\text{read_data}(pm_phy, \text{pdbr_data_type})(\text{PDBR})(s))$
{-2}	$\forall (s_1, s_2: (pm' \text{states})):$ $\text{pe_in_pdir_range?}(s_1,$ $\quad \text{restrict}[\text{Address}, \text{Memory_Address_4G}, \text{boolean}]$ $\quad \quad ((pm' \text{ro_addr} \cup pm' \text{rw_addr}))$ $=$ $\text{pe_in_pdir_range?}(s_2,$ $\quad \text{restrict}[\text{Address}, \text{Memory_Address_4G}, \text{boolean}]$ $\quad \quad ((pm' \text{ro_addr} \cup pm' \text{rw_addr}))$ \wedge $\text{pe_in_ptab_range?}(s_1,$ $\quad \text{restrict}[\text{Address}, \text{Memory_Address_4G}, \text{boolean}]$ $\quad \quad ((pm' \text{ro_addr} \cup pm' \text{rw_addr}))$ $=$ $\text{pe_in_ptab_range?}(s_2,$ $\quad \text{restrict}[\text{Address}, \text{Memory_Address_4G}, \text{boolean}]$ $\quad \quad ((pm' \text{ro_addr} \cup pm' \text{rw_addr}))$
{-3}	$\text{plain_memory?}(pm_phy)$
{-4}	$pm' \text{states} = pm_phy \text{states}$
{-5}	$(pm' \text{other_actions} \subseteq pm_phy \text{other_actions})$
{-6}	$(\text{linear_resolve_register_transformers} \subseteq pm_phy \text{other_actions})$
{-7}	$\forall (a: ((pm' \text{ro_addr} \cup pm' \text{rw_addr}))):$ $\text{Mem?}(\text{type_of}(a)) \wedge 0 \leq \text{offset}(a) \wedge \text{offset}(a) < \text{max_linear_offset}$
{-8}	$\forall (s: (pm' \text{states})):$ $\text{linear_blessed?}(s, \text{restrict}[\text{Address}, \text{Memory_Address_4G}, \text{boolean}](pm' \text{ro_addr}),$ $\quad \text{restrict}[\text{Address}, \text{Memory_Address_4G}, \text{boolean}](pm' \text{rw_addr}))$
{-9}	$\text{Mem?}(\text{addr}' \text{type_of})$
{-10}	$0 \leq \text{addr}' \text{offset}$
{-11}	$\text{addr}' \text{offset} < \text{max_linear_offset}$
{-12}	$\text{union}(pm' \text{ro_addr}, pm' \text{rw_addr})(\text{addr}')$
{-13}	$\text{is_linear_plain_memory?}(pm')$
{-14}	$pm' \text{states}(s'_1)$
{-15}	$pm' \text{states}(s'_2)$
{-16}	$\text{OK?}(\text{translate}(\text{pdir_lvl}, \text{data}(\text{read_data}(pm_phy, \text{pdbr_data_type})(\text{PDBR})(s'_1)) \text{base_addr},$ $\quad \text{addr}', \text{access}', \text{priv}')$ $\quad \quad (\text{state}(\text{read_data}(pm_phy, \text{pdbr_data_type})(\text{PDBR})(s'_1))))$
{1}	$\text{OK?}(\text{read_data}(pm_phy, \text{pdbr_data_type})(\text{PDBR})(s'_1))$
{2}	$\text{data}(\text{translate}(\text{pdir_lvl}, \text{data}(\text{read_data}(pm_phy, \text{pdbr_data_type})(\text{PDBR})(s'_1)) \text{base_addr},$ $\quad \text{addr}', \text{access}', \text{priv}')$ $\quad \quad (\text{state}(\text{read_data}(pm_phy, \text{pdbr_data_type})(\text{PDBR})(s'_1)))) \text{'1}$
{3}	$\text{ptab_lvl} = \text{pdir_lvl}$
{4}	$\text{pe_in_ptab_range?}(s'_2,$ $\quad \text{restrict}[\text{Address}, \text{Memory_Address_4G}, \text{boolean}]$ $\quad \quad ((pm' \text{ro_addr} \cup pm' \text{rw_addr}))$ $\quad (\text{xlat_idx}(\text{ptab_lvl},$ $\quad \quad \text{data}(\text{translate}(\text{pdir_lvl},$ $\quad \quad \quad \text{data}(\text{read_data}(pm_phy, \text{pdbr_data_type})$ $\quad \quad \quad \quad (\text{PDBR})$ $\quad \quad \quad \quad (s'_1)) \text{base_addr},$ $\quad \quad \quad \quad \text{addr}', \text{access}', \text{priv}')$ $\quad \quad \quad \quad (\text{state}(\text{read_data}(pm_phy, \text{pdbr_data_type})$ $\quad \quad \quad \quad \quad (\text{PDBR})$ $\quad \quad \quad \quad \quad (s'_1)))) \text{'2},$ $\quad \quad \quad \quad \text{addr}'))$

Instantiating quantified variables,
 This completes the proof of `xlat_pe_in_pt_range.2.3`.
 Q.E.D.

C.117.49

Linear_Memory_Properties.linear_resolve_unchanged_pm_phy

Terse proof for `linear_resolve_unchanged_pm_phy`.

`linear_resolve_unchanged_pm_phy`:

<p>{1} \forall (pm: Plain_Memory[Linear_memory], s: Linear_memory[Physical_memory, pm_phy], addr: Memory_Address_4G, access: Memory_access): union(pm'ro_addr, pm'rw_addr)(addr) \wedge is_linear_plain_memory?(pm) \wedge pm'states(s) \supset unchanged_memory_invariant?[Physical_memory] (pm_phy'mem, pm'states, singleton(expr_2_super[Physical_memory, Address](linear_resolve(addr, ac- cess))), ((pm_phy'ro_addr \cup pm_phy'rw_addr) \ extend [Address, Memory_Address_4G, bool, F</p>

Installing automatic rewrites from: `plain_memory_unchanged_memory_ok_result`

Repeatedly Skolemizing and flattening,

Letting addresses name difference(union(pm_phy'ro_addr, pm_phy'rw_addr), extend[Address, Mem-
 ory_Address_4G, bool, FALSE] (address_in_pt_range?(s!1, restrict [Address, Memory_Address_4G,
 boolean] (union (pm!1'ro_addr, pm!1'rw_addr))))),

Replacing using formula -1,

Case splitting on subset?(addresses, union(pm_phy'ro_addr, pm_phy'rw_addr)),

we get 2 subgoals:

`linear_resolve_unchanged_pm_phy.1`:

<p>{-1} (addresses \subseteq (pm_phy'ro_addr \cup pm_phy'rw_addr)) {-2} ((pm_phy'ro_addr \cup pm_phy'rw_addr) \ extend [Address, Memory_Address_4G, bool, FALSE](ad- = addresses {-3} Mem?(addr' 'type_of) {-4} $0 \leq$ addr' 'offset {-5} addr' 'offset < max_linear_offset {-6} union(pm' 'ro_addr, pm' 'rw_addr)(addr') {-7} is_linear_plain_memory?(pm') {-8} pm' 'states(s')</p>
<p>{1} unchanged_memory_invariant?[Physical_memory] (pm_phy'mem, pm'states, singleton(expr_2_super[Physical_memory, Address](linear_resolve(addr', access'))), addresses)</p>

Expanding the definition of `linear_resolve`,

Using lemma `pm_plain_phy`,

Using lemma `pm_states`,

Simplifying, rewriting, and recording with decision procedures,

Replacing using formula -1,

Installing automatic rewrites from: (has_next_state! max_level! bits_per_level! bus_width! pe_size!
 singleton! <! Mem! max_linear!)

Copying formula number: -9

Expanding the definition of `is_linear_plain_memory?`,

Applying disjunctive simplification to flatten sequent,

Installing automatic rewrites from: pdir_lvl!! ptab_lvl!!

Case splitting on FORALL (cs: Segment_Reg_type, pdir: Pdir_type, leaf: bool, base: {b: Memory_Address_4G | IF leaf THEN aligned?(-1 * (pdir_lvl[Physical_Memory, pm_phy] * bits_per_level [Physical_Memory, pm_phy]) - bits_per_level [Physical_Memory, pm_phy] + bus_width) (offset(b)) ELSE aligned?(bits_per_level [Physical_Memory, pm_phy] + pe_size) (offset(b)) ENDIF}): NOT leaf IMPLIES (FORALL (x1: Physical_Memory): every[Physical_Memory, [bool, Address]] (LAMBDA (x: Physical_Memory): TRUE, LAMBDA (t: [bool, Address]): Mem?(t'2'type_of) AND 0 <= t'2'offset AND t'2'offset < max_linear_offset) (translate[Physical_Memory, pm_phy] (1, base, addr!1, access!1, segment_to_priv(cs))(x1))), FORALL (cs: Segment_Reg_type, pdir: Pdir_type): FORALL (x1: Physical_Memory): every[Physical_Memory, [bool, Address]] (LAMBDA (x: Physical_Memory): TRUE, LAMBDA (t: [bool, Address]): Mem?(t'2'type_of) AND (0 <= t'2'offset AND t'2'offset < max_linear_offset) AND IF t'1 THEN aligned?(22)(offset(t'2)) ELSE aligned?(12)(offset(t'2)) ENDIF) (translate[Physical_Memory, pm_phy] (0, pdir'base_addr, addr!1, access!1, segment_to_priv(cs)) (x1)),

we get 5 subgoals:

linear_resolve_unchanged_pm_phy.1.1:

```

{-1}  ∀ (cs: Segment_Reg_type, paddr: Paddr_type):
      ∀ (x1: Physical_memory):
        every[Physical_memory, [bool, Address]]
          (λ (x: Physical_memory): TRUE,
           λ (t: [bool, Address]):
             Mem?(t'2'type_of) ∧
             (0 ≤ t'2'offset ∧ t'2'offset < max_linear_offset) ∧
             IF t'1
               THEN aligned?(22)(offset(t'2))
               ELSE aligned?(12)(offset(t'2))
             ENDIF)
          (translate[Physical_memory, pm_phy]
           (0, paddr'base_addr, addr', access', segment_to_priv(cs))(x1))
{-2}  ∀ (cs: Segment_Reg_type, paddr: Paddr_type, leaf: bool,
      base:
        {b: Memory_Address_4G |
          IF leaf
            THEN aligned?(-1 × (pdir_lvl [Physical_memory, pm_phy] × bits_per_level
              (offset(b))
            ELSE aligned?(bits_per_level[Physical_memory, pm_phy] + pe_size)
              (offset(b))
          ENDIF}):
      ¬ leaf ⊃
      (∀ (x1: Physical_memory):
        every[Physical_memory, [bool, Address]]
          (λ (x: Physical_memory): TRUE,
           λ (t: [bool, Address]):
             Mem?(t'2'type_of) ∧ 0 ≤ t'2'offset ∧ t'2'offset < max_linear_offset)
          (translate[Physical_memory, pm_phy]
           (1, base, addr', access', segment_to_priv(cs))(x1)))
{-3}  pm'mem = linear_pm
{-4}  ∀ (s: (pm'states)): OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-5}  ∀ (s1, s2: (pm'states)):
      data(read_data(pm_phy, segment_reg_data_type)(CS)(s1)) =
      data(read_data(pm_phy, segment_reg_data_type)(CS)(s2))
{-6}  ∀ (s: (pm'states)): OK?(read_data(pm_phy, paddr_data_type)(PADDR)(s))
{-7}  ∀ (s1, s2: (pm'states)):
      data(read_data(pm_phy, paddr_data_type)(PADDR)(s1)) =
      data(read_data(pm_phy, paddr_data_type)(PADDR)(s2))
{-8}  ∀ (s1, s2: (pm'states)):
      pe_in_pdir_range?(s1,
        restrict[Address, Memory_Address_4G, boolean]
          ((pm'ro_addr ∪ pm'rw_addr)))
      =
      pe_in_pdir_range?(s2,
        restrict[Address, Memory_Address_4G, boolean]
          ((pm'ro_addr ∪ pm'rw_addr)))
      ∧
      pe_in_ptab_range?(s1,
        restrict[Address, Memory_Address_4G, boolean]
          ((pm'ro_addr ∪ pm'rw_addr)))
      =
      pe_in_ptab_range?(s2,
        restrict[Address, Memory_Address_4G, boolean]
          ((pm'ro_addr ∪ pm'rw_addr)))
1736 {-9}  ∀ (s: (pm'states), lvl: Level,
      a:
        (pe_in_pt_range?(s,
          restrict[Address, Memory_Address_4G, boolean]
            ((pm'ro_addr ∪ pm'rw_addr)),
          lvl))):
      OK?(read_data(pm_phy, paging_data_type(lvl))(a)(s))
{-10}  ∀ (s1, s2: (pm'states), lvl: Level,

```

C.117 Proofs for Linear_Memory_Properties (challenge-linear.pvs)

Rewriting using `plain_memory_unchanged_composition` [`Physical_memory`, `Segment_Reg_type`, `Address`], matching in `*` where `P` gets `LAMBDA (t: Segment_Reg_type): FORALL (s: (pm!1'states)):`
`OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s)) AND t = data(read_data(pm_phy, segment_reg_data_type)(CS)(s)),`

we get 3 subgoals:

linear_resolve_unchanged_pm_phy.1.1.1:

```

{-1}  ∀ (cs: Segment_Reg_type, pdir: Pdir_type):
      ∀ (x1: Physical_memory):
        every[Physical_memory, [bool, Address]]
          (λ (x: Physical_memory): TRUE,
           λ (t: [bool, Address]):
             Mem?(t'2'type_of) ∧
             (0 ≤ t'2'offset ∧ t'2'offset < max_linear_offset) ∧
             IF t'1
               THEN aligned?(22)(offset(t'2))
               ELSE aligned?(12)(offset(t'2))
             ENDIF)
          (translate[Physical_memory, pm_phy]
           (0, pdir'base_addr, addr', access', segment_to_priv(cs))(x1))
{-2}  ∀ (cs: Segment_Reg_type, pdir: Pdir_type, leaf: bool,
      base:
        {b: Memory_Address_4G |
          IF leaf
            THEN aligned?(-1 × (pdir_lvl [Physical_memory, pm_phy] × bits_per_level
              (offset(b))
            ELSE aligned?(bits_per_level[Physical_memory, pm_phy] + pe_size)
              (offset(b))
          ENDIF}):
      ¬ leaf ⊃
      (∀ (x1: Physical_memory):
        every[Physical_memory, [bool, Address]]
          (λ (x: Physical_memory): TRUE,
           λ (t: [bool, Address]):
             Mem?(t'2'type_of) ∧ 0 ≤ t'2'offset ∧ t'2'offset < max_linear_offset)
          (translate[Physical_memory, pm_phy]
           (1, base, addr', access', segment_to_priv(cs))(x1)))
{-3}  pm'mem = linear_pm
{-4}  ∀ (s: (pm'states)): OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-5}  ∀ (s1, s2: (pm'states)):
      data(read_data(pm_phy, segment_reg_data_type)(CS)(s1)) =
      data(read_data(pm_phy, segment_reg_data_type)(CS)(s2))
{-6}  ∀ (s: (pm'states)): OK?(read_data(pm_phy, pdir_data_type)(PDIR)(s))
{-7}  ∀ (s1, s2: (pm'states)):
      data(read_data(pm_phy, pdir_data_type)(PDIR)(s1)) =
      data(read_data(pm_phy, pdir_data_type)(PDIR)(s2))
{-8}  ∀ (s1, s2: (pm'states)):
      pe_in_pdir_range?(s1,
        restrict[Address, Memory_Address_4G, boolean]
          ((pm'ro_addr ∪ pm'rw_addr)))
      =
      pe_in_pdir_range?(s2,
        restrict[Address, Memory_Address_4G, boolean]
          ((pm'ro_addr ∪ pm'rw_addr)))
      ∧
      pe_in_ptab_range?(s1,
        restrict[Address, Memory_Address_4G, boolean]
          ((pm'ro_addr ∪ pm'rw_addr)))
      =
      pe_in_ptab_range?(s2,
        restrict[Address, Memory_Address_4G, boolean]
          ((pm'ro_addr ∪ pm'rw_addr)))
1738 {-9}  ∀ (s: (pm'states), lvl: Level,
      a:
        (pe_in_pt_range?(s,
          restrict[Address, Memory_Address_4G, boolean]
            ((pm'ro_addr ∪ pm'rw_addr)),
          lvl)):
      OK?(read_data(pm_phy, paging_data_type(lvl))(a)(s))
{-10}  ∀ (s1, s2: (pm'states), lvl: Level,

```

Hiding formulas: 2,

Repeatedly Skolemizing and flattening,

Keeping (-7 -8 1) and hiding *,

Instantiating quantified variables,

Instantiating quantified variables,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `linear_resolve_unchanged_pm_phy.1.1.1`.

linear_resolve_unchanged_pm_phy.1.1.2:

```

{-1}  ∀ (cs: Segment_Reg_type, pdir: Pdir_type):
      ∀ (x1: Physical_memory):
        every[Physical_memory, [bool, Address]]
          (λ (x: Physical_memory): TRUE,
           λ (t: [bool, Address]):
             Mem?(t'2'type_of) ∧
             (0 ≤ t'2'offset ∧ t'2'offset < max_linear_offset) ∧
             IF t'1
               THEN aligned?(22)(offset(t'2))
               ELSE aligned?(12)(offset(t'2))
             ENDIF)
          (translate[Physical_memory, pm_phy]
           (0, pdir'base_addr, addr', access', segment_to_priv(cs))(x1))
{-2}  ∀ (cs: Segment_Reg_type, pdir: Pdir_type, leaf: bool,
      base:
        {b: Memory_Address_4G |
          IF leaf
            THEN aligned?(-1 × (pdir_lvl [Physical_memory, pm_phy] × bits_per_level
              (offset(b))
            ELSE aligned?(bits_per_level[Physical_memory, pm_phy] + pe_size)
              (offset(b))
          ENDIF}):
      ¬ leaf ⊃
      (∀ (x1: Physical_memory):
        every[Physical_memory, [bool, Address]]
          (λ (x: Physical_memory): TRUE,
           λ (t: [bool, Address]):
             Mem?(t'2'type_of) ∧ 0 ≤ t'2'offset ∧ t'2'offset < max_linear_offset)
          (translate[Physical_memory, pm_phy]
           (1, base, addr', access', segment_to_priv(cs))(x1)))
{-3}  pm'mem = linear_pm
{-4}  ∀ (s: (pm'states)): OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-5}  ∀ (s1, s2: (pm'states)):
      data(read_data(pm_phy, segment_reg_data_type)(CS)(s1)) =
      data(read_data(pm_phy, segment_reg_data_type)(CS)(s2))
{-6}  ∀ (s: (pm'states)): OK?(read_data(pm_phy, pdir_data_type)(PDIR)(s))
{-7}  ∀ (s1, s2: (pm'states)):
      data(read_data(pm_phy, pdir_data_type)(PDIR)(s1)) =
      data(read_data(pm_phy, pdir_data_type)(PDIR)(s2))
{-8}  ∀ (s1, s2: (pm'states)):
      pe_in_pdir_range?(s1,
        restrict[Address, Memory_Address_4G, boolean]
          ((pm'ro_addr ∪ pm'rw_addr)))
      =
      pe_in_pdir_range?(s2,
        restrict[Address, Memory_Address_4G, boolean]
          ((pm'ro_addr ∪ pm'rw_addr)))
      ∧
      pe_in_ptab_range?(s1,
        restrict[Address, Memory_Address_4G, boolean]
          ((pm'ro_addr ∪ pm'rw_addr)))
      =
      pe_in_ptab_range?(s2,
        restrict[Address, Memory_Address_4G, boolean]
          ((pm'ro_addr ∪ pm'rw_addr)))
1740 {-9}  ∀ (s: (pm'states), lvl: Level,
      a:
        (pe_in_pt_range?(s,
          restrict[Address, Memory_Address_4G, boolean]
            ((pm'ro_addr ∪ pm'rw_addr)),
          lvl))):
      OK?(read_data(pm_phy, paging_data_type(lvl))(a)(s))
{-10}  ∀ (s1, s2: (pm'states), lvl: Level,

```

Hiding formulas: 2,

Using lemma `pm_unchanged_singleton_linear_resolve_reg_transformers`,

Expanding the definition of `linear_resolve_register_transformers`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Using lemma `unchanged_memory_invariant_mono`,

Rewriting using `subset_reflexive`, matching in `*`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `linear_resolve_unchanged_pm_phy.1.1.2`.

linear_resolve_unchanged_pm_phy.1.1.3:

```

{-1}  ∀ (cs: Segment_Reg_type, pdir: Pdir_type):
      ∀ (x1: Physical_memory):
        every[Physical_memory, [bool, Address]]
          (λ (x: Physical_memory): TRUE,
           λ (t: [bool, Address]):
             Mem?(t'2'type_of) ∧
             (0 ≤ t'2'offset ∧ t'2'offset < max_linear_offset) ∧
             IF t'1
               THEN aligned?(22)(offset(t'2))
               ELSE aligned?(12)(offset(t'2))
             ENDIF)
          (translate[Physical_memory, pm_phy]
           (0, pdir'base_addr, addr', access', segment_to_priv(cs))(x1))
{-2}  ∀ (cs: Segment_Reg_type, pdir: Pdir_type, leaf: bool,
      base:
        {b: Memory_Address_4G |
          IF leaf
            THEN aligned?(-1 × (pdir_lvl [Physical_memory, pm_phy] × bits_per_level
              (offset(b))
            ELSE aligned?(bits_per_level[Physical_memory, pm_phy] + pe_size)
              (offset(b))
          ENDIF}):
      ¬ leaf ⊃
      (∀ (x1: Physical_memory):
        every[Physical_memory, [bool, Address]]
          (λ (x: Physical_memory): TRUE,
           λ (t: [bool, Address]):
             Mem?(t'2'type_of) ∧ 0 ≤ t'2'offset ∧ t'2'offset < max_linear_offset)
          (translate[Physical_memory, pm_phy]
           (1, base, addr', access', segment_to_priv(cs))(x1)))
{-3}  pm'mem = linear_pm
{-4}  ∀ (s: (pm'states)): OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-5}  ∀ (s1, s2: (pm'states)):
      data(read_data(pm_phy, segment_reg_data_type)(CS)(s1)) =
      data(read_data(pm_phy, segment_reg_data_type)(CS)(s2))
{-6}  ∀ (s: (pm'states)): OK?(read_data(pm_phy, pdir_data_type)(PDIR)(s))
{-7}  ∀ (s1, s2: (pm'states)):
      data(read_data(pm_phy, pdir_data_type)(PDIR)(s1)) =
      data(read_data(pm_phy, pdir_data_type)(PDIR)(s2))
{-8}  ∀ (s1, s2: (pm'states)):
      pe_in_pdir_range?(s1,
        restrict[Address, Memory_Address_4G, boolean]
          ((pm'ro_addr ∪ pm'rw_addr)))
      =
      pe_in_pdir_range?(s2,
        restrict[Address, Memory_Address_4G, boolean]
          ((pm'ro_addr ∪ pm'rw_addr)))
      ∧
      pe_in_ptab_range?(s1,
        restrict[Address, Memory_Address_4G, boolean]
          ((pm'ro_addr ∪ pm'rw_addr)))
      =
      pe_in_ptab_range?(s2,
        restrict[Address, Memory_Address_4G, boolean]
          ((pm'ro_addr ∪ pm'rw_addr)))
1742 {-9}  ∀ (s: (pm'states), lvl: Level,
      a:
        (pe_in_pt_range?(s,
          restrict[Address, Memory_Address_4G, boolean]
            ((pm'ro_addr ∪ pm'rw_addr)),
          lvl))):
      OK?(read_data(pm_phy, paging_data_type(lvl))(a)(s))
{-10}  ∀ (s1, s2: (pm'states), lvl: Level,

```

Hiding formulas: 2,

Repeatedly Skolemizing and flattening,

Rewriting using `plain_memory_unchanged_composition[Physical_memory, Pdbr_type, Address]`,
matching in * where P gets `LAMBDA (base: Pdbr_type): FORALL (s: (pm_phy'states)):`
`OK?(read_data(pm_phy, pdbr_data_type)(PDBR)(s)) AND data(read_data(pm_phy, pdbr_data_type)(PDBR)(s))`
`= base,`

we get 5 subgoals:

linear_resolve_unchanged_pm_phy.1.1.3.1:

```

{-1}  ∀ (s: (pm' states)):
      OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s)) ∧
      d' = data(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-2}  ∀ (cs: Segment_Reg_type, pdir: Pdir_type):
      ∀ (x1: Physical_memory):
        every[Physical_memory, [bool, Address]]
          (λ (x: Physical_memory): TRUE,
           λ (t: [bool, Address]):
             Mem?(t'2'type_of) ∧
             (0 ≤ t'2'offset ∧ t'2'offset < max_linear_offset) ∧
             IF t'1
               THEN aligned?(22)(offset(t'2))
               ELSE aligned?(12)(offset(t'2))
             ENDIF)
          (translate[Physical_memory, pm_phy]
           (0, pdir'base_addr, addr', access', segment_to_priv(cs))(x1))
{-3}  ∀ (cs: Segment_Reg_type, pdir: Pdir_type, leaf: bool,
      base:
        {b: Memory_Address_4G |
          IF leaf
            THEN aligned?(-1 × (pdir_lvl [Physical_memory, pm_phy] × bits_per_level
              (offset(b))
            ELSE aligned?(bits_per_level[Physical_memory, pm_phy] + pe_size)
              (offset(b))
            ENDIF}):
      ¬ leaf ⊃
      (∀ (x1: Physical_memory):
        every[Physical_memory, [bool, Address]]
          (λ (x: Physical_memory): TRUE,
           λ (t: [bool, Address]):
             Mem?(t'2'type_of) ∧ 0 ≤ t'2'offset ∧ t'2'offset < max_linear_offset)
          (translate[Physical_memory, pm_phy]
           (1, base, addr', access', segment_to_priv(cs))(x1)))
{-4}  pm'mem = linear_pm
{-5}  ∀ (s: (pm' states)): OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-6}  ∀ (s1, s2: (pm' states)):
      data(read_data(pm_phy, segment_reg_data_type)(CS)(s1)) =
      data(read_data(pm_phy, segment_reg_data_type)(CS)(s2))
{-7}  ∀ (s: (pm' states)): OK?(read_data(pm_phy, pdir_data_type)(PDIR)(s))
{-8}  ∀ (s1, s2: (pm' states)):
      data(read_data(pm_phy, pdir_data_type)(PDIR)(s1)) =
      data(read_data(pm_phy, pdir_data_type)(PDIR)(s2))
{-9}  ∀ (s1, s2: (pm' states)):
      pe_in_pdir_range?(s1,
        restrict[Address, Memory_Address_4G, boolean]
          ((pm'ro_addr ∪ pm'rw_addr)))
      =
      pe_in_pdir_range?(s2,
        restrict[Address, Memory_Address_4G, boolean]
          ((pm'ro_addr ∪ pm'rw_addr)))
      ∧
      pe_in_ptab_range?(s1,
        restrict[Address, Memory_Address_4G, boolean]
          ((pm'ro_addr ∪ pm'rw_addr)))
      =
      pe_in_ptab_range?(s2,
        restrict[Address, Memory_Address_4G, boolean]
          ((pm'ro_addr ∪ pm'rw_addr)))
1744
{-10} ∀ (s: (pm' states), lvl: Level,
      a:
        (pe_in_pt_range?(s,
          restrict[Address, Memory_Address_4G, boolean]
            ((pm'ro_addr ∪ pm'rw_addr)),

```

Keeping (-7 -8 1) and hiding *,

Repeatedly Skolemizing and flattening,

Instantiating quantified variables,

Instantiating quantified variables,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `linear_resolve_unchanged_pm_phy.1.1.3.1`.

linear_resolve_unchanged_pm_phy.1.1.3.2:

```

{-1}  ∀ (s: (pm' states)):
      OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s)) ∧
      d' = data(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-2}  ∀ (cs: Segment_Reg_type, pdir: Pdir_type):
      ∀ (x1: Physical_memory):
        every[Physical_memory, [bool, Address]]
          (λ (x: Physical_memory): TRUE,
           λ (t: [bool, Address]):
             Mem?(t'2'type_of) ∧
             (0 ≤ t'2'offset ∧ t'2'offset < max_linear_offset) ∧
             IF t'1
               THEN aligned?(22)(offset(t'2))
               ELSE aligned?(12)(offset(t'2))
             ENDIF)
          (translate[Physical_memory, pm_phy]
           (0, pdir'base_addr, addr', access', segment_to_priv(cs))(x1))
{-3}  ∀ (cs: Segment_Reg_type, pdir: Pdir_type, leaf: bool,
      base:
        {b: Memory_Address_4G |
          IF leaf
            THEN aligned?(-1 × (pdir_lvl [Physical_memory, pm_phy] × bits_per_level
              (offset(b))
            ELSE aligned?(bits_per_level[Physical_memory, pm_phy] + pe_size)
              (offset(b))
            ENDIF}):
      ¬ leaf ⊃
      (∀ (x1: Physical_memory):
        every[Physical_memory, [bool, Address]]
          (λ (x: Physical_memory): TRUE,
           λ (t: [bool, Address]):
             Mem?(t'2'type_of) ∧ 0 ≤ t'2'offset ∧ t'2'offset < max_linear_offset)
          (translate[Physical_memory, pm_phy]
           (1, base, addr', access', segment_to_priv(cs))(x1)))
{-4}  pm'mem = linear_pm
{-5}  ∀ (s: (pm' states)): OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-6}  ∀ (s1, s2: (pm' states)):
      data(read_data(pm_phy, segment_reg_data_type)(CS)(s1)) =
      data(read_data(pm_phy, segment_reg_data_type)(CS)(s2))
{-7}  ∀ (s: (pm' states)): OK?(read_data(pm_phy, pdir_data_type)(PDIR)(s))
{-8}  ∀ (s1, s2: (pm' states)):
      data(read_data(pm_phy, pdir_data_type)(PDIR)(s1)) =
      data(read_data(pm_phy, pdir_data_type)(PDIR)(s2))
{-9}  ∀ (s1, s2: (pm' states)):
      pe_in_pdir_range?(s1,
        restrict[Address, Memory_Address_4G, boolean]
          ((pm'ro_addr ∪ pm'rw_addr)))
      =
      pe_in_pdir_range?(s2,
        restrict[Address, Memory_Address_4G, boolean]
          ((pm'ro_addr ∪ pm'rw_addr)))
      ∧
      pe_in_ptab_range?(s1,
        restrict[Address, Memory_Address_4G, boolean]
          ((pm'ro_addr ∪ pm'rw_addr)))
      =
      pe_in_ptab_range?(s2,
        restrict[Address, Memory_Address_4G, boolean]
          ((pm'ro_addr ∪ pm'rw_addr)))
1746
{-10} ∀ (s: (pm' states), lvl: Level,
      a:
        (pe_in_pt_range?(s,
          restrict[Address, Memory_Address_4G, boolean]
            ((pm'ro_addr ∪ pm'rw_addr)),

```

Hiding formulas: 2,

Using lemma `pm_unchanged_singleton_linear_resolve_reg_transformers`,

Expanding the definition of `linear_resolve_register_transformers`,

Using lemma `unchanged_memory_invariant_mono`,

Rewriting using `subset_reflexive`, matching in `*`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `linear_resolve_unchanged_pm_phy.1.1.3.2`.

linear_resolve_unchanged_pm_phy.1.1.3.3:

```

{-1}  ∀ (s: (pm' states)):
      OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s)) ∧
      d' = data(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-2}  ∀ (cs: Segment_Reg_type, pdr: Pdr_type):
      ∀ (x1: Physical_memory):
        every[Physical_memory, [bool, Address]]
          (λ (x: Physical_memory): TRUE,
           λ (t: [bool, Address]):
             Mem?(t'2'type_of) ∧
             (0 ≤ t'2'offset ∧ t'2'offset < max_linear_offset) ∧
             IF t'1
               THEN aligned?(22)(offset(t'2))
               ELSE aligned?(12)(offset(t'2))
             ENDIF)
          (translate[Physical_memory, pm_phy]
           (0, pdr'base_addr, addr', access', segment_to_priv(cs))(x1))
{-3}  ∀ (cs: Segment_Reg_type, pdr: Pdr_type, leaf: bool,
      base:
        {b: Memory_Address_4G |
          IF leaf
            THEN aligned?(-1 × (pdir_lvl [Physical_memory, pm_phy] × bits_per_level
              (offset(b))
            ELSE aligned?(bits_per_level[Physical_memory, pm_phy] + pe_size)
              (offset(b))
            ENDIF}):
      ¬ leaf ⊃
      (∀ (x1: Physical_memory):
        every[Physical_memory, [bool, Address]]
          (λ (x: Physical_memory): TRUE,
           λ (t: [bool, Address]):
             Mem?(t'2'type_of) ∧ 0 ≤ t'2'offset ∧ t'2'offset < max_linear_offset)
          (translate[Physical_memory, pm_phy]
           (1, base, addr', access', segment_to_priv(cs))(x1)))
{-4}  pm'mem = linear_pm
{-5}  ∀ (s: (pm' states)): OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-6}  ∀ (s1, s2: (pm' states)):
      data(read_data(pm_phy, segment_reg_data_type)(CS)(s1)) =
      data(read_data(pm_phy, segment_reg_data_type)(CS)(s2))
{-7}  ∀ (s: (pm' states)): OK?(read_data(pm_phy, pdr_data_type)(PDR)(s))
{-8}  ∀ (s1, s2: (pm' states)):
      data(read_data(pm_phy, pdr_data_type)(PDR)(s1)) =
      data(read_data(pm_phy, pdr_data_type)(PDR)(s2))
{-9}  ∀ (s1, s2: (pm' states)):
      pe_in_pdir_range?(s1,
        restrict[Address, Memory_Address_4G, boolean]
          ((pm'ro_addr ∪ pm'rw_addr)))
      =
      pe_in_pdir_range?(s2,
        restrict[Address, Memory_Address_4G, boolean]
          ((pm'ro_addr ∪ pm'rw_addr)))
      ∧
      pe_in_ptab_range?(s1,
        restrict[Address, Memory_Address_4G, boolean]
          ((pm'ro_addr ∪ pm'rw_addr)))
      =
      pe_in_ptab_range?(s2,
        restrict[Address, Memory_Address_4G, boolean]
          ((pm'ro_addr ∪ pm'rw_addr)))
1748
{-10} ∀ (s: (pm' states), lvl: Level,
      a:
        (pe_in_pt_range?(s,
          restrict[Address, Memory_Address_4G, boolean]
            ((pm'ro_addr ∪ pm'rw_addr))),

```

Hiding formulas: 2,

Repeatedly Skolemizing and flattening,

Rewriting using plain_memory_unchanged_composition [Physical_Memory, [leaf: bool, {b: Memory_Address_4G | IF leaf THEN aligned?(-1 * (pdir_lvl * bits_per_level) - bits_per_level + bus_width) (offset(b)) ELSE aligned?(bits_per_level + pe_size)(offset(b)) ENDIF}], Address], matching in * where P gets LAMBDA (d: [leaf: bool, {b: Memory_Address_4G | IF leaf THEN aligned?(-1 * (pdir_lvl * bits_per_level) - bits_per_level + bus_width) (offset(b)) ELSE aligned?(bits_per_level + pe_size)(offset(b)) ENDIF}]): FORALL (s: (pm_phy'states)): OK?(translate(0, d!2'base_addr, addr!1, access!1, segment_to_priv(d!1)) (s)) AND d = data(translate(0, d!2'base_addr, addr!1, access!1, segment_to_priv(d!1)) (s)),

we get 4 subgoals:

linear_resolve_unchanged_pm_phy.1.1.3.3.1:

```

{-1}  ∀ (s: (pm_phy' states)):
      OK?(read_data(pm_phy, pabr_data_type)(PDBR)(s)) ∧
      data(read_data(pm_phy, pabr_data_type)(PDBR)(s)) = d''
{-2}  ∀ (s: (pm' states)):
      OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s)) ∧
      d' = data(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-3}  ∀ (cs: Segment_Reg_type, pabr: Pabr_type):
      ∀ (x1: Physical_memory):
        every[Physical_memory, [bool, Address]]
          (λ (x: Physical_memory): TRUE,
           λ (t: [bool, Address]):
             Mem?(t'2'type_of) ∧
             (0 ≤ t'2'offset ∧ t'2'offset < max_linear_offset) ∧
             IF t'1
               THEN aligned?(22)(offset(t'2))
               ELSE aligned?(12)(offset(t'2))
             ENDIF)
          (translate[Physical_memory, pm_phy]
           (0, pabr'base_addr, addr', access', segment_to_priv(cs))(x1))
{-4}  ∀ (cs: Segment_Reg_type, pabr: Pabr_type, leaf: bool,
      base:
        {b: Memory_Address_4G |
          IF leaf
            THEN aligned?(-1 × (pdir_lvl [Physical_memory, pm_phy] × bits_per_level
              (offset(b))
            ELSE aligned?(bits_per_level[Physical_memory, pm_phy] + pe_size)
              (offset(b))
            ENDIF}):
      ¬ leaf ⊃
      (∀ (x1: Physical_memory):
        every[Physical_memory, [bool, Address]]
          (λ (x: Physical_memory): TRUE,
           λ (t: [bool, Address]):
             Mem?(t'2'type_of) ∧ 0 ≤ t'2'offset ∧ t'2'offset < max_linear_offset)
          (translate[Physical_memory, pm_phy]
           (1, base, addr', access', segment_to_priv(cs))(x1)))
{-5}  pm' mem = linear_pm
{-6}  ∀ (s: (pm' states)): OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-7}  ∀ (s1, s2: (pm' states)):
      data(read_data(pm_phy, segment_reg_data_type)(CS)(s1)) =
      data(read_data(pm_phy, segment_reg_data_type)(CS)(s2))
{-8}  ∀ (s: (pm' states)): OK?(read_data(pm_phy, pabr_data_type)(PDBR)(s))
{-9}  ∀ (s1, s2: (pm' states)):
      data(read_data(pm_phy, pabr_data_type)(PDBR)(s1)) =
      data(read_data(pm_phy, pabr_data_type)(PDBR)(s2))
{-10} ∀ (s1, s2: (pm' states)):
      pe_in_pdir_range?(s1,
        restrict[Address, Memory_Address_4G, boolean]
          ((pm'ro_addr ∪ pm'rw_addr)))
      =
      pe_in_pdir_range?(s2,
        restrict[Address, Memory_Address_4G, boolean]
          ((pm'ro_addr ∪ pm'rw_addr)))
      ∧
      pe_in_ptab_range?(s1,
        restrict[Address, Memory_Address_4G, boolean]
          ((pm'ro_addr ∪ pm'rw_addr)))
      =
      pe_in_ptab_range?(s2,
        restrict[Address, Memory_Address_4G, boolean]
          ((pm'ro_addr ∪ pm'rw_addr)))
{-11} ∀ (s: (pm' states), lvl: Level,
      a:

```

C.117 Proofs for Linear_Memory_Properties (challenge-linear.pvs)

Hiding formulas: 2,

Repeatedly Skolemizing and flattening,

Case splitting on `pe_in_pt_range?(s!2, restrict[Address, Memory_Address_4G, boolean]
(union(pm!1'ro_addr, pm!1'rw_addr)), 0) (xlat_idx(0, d!2'base_addr, addr!1))`,

we get 3 subgoals:

linear_resolve_unchanged_pm_phy.1.1.3.3.1.1:

```

{-1}  pe_in_pt_range?(s'',
      restrict[Address, Memory_Address_4G, boolean]
        ((pm''_ro_addr ∪ pm''_rw_addr)),
      0)
      (xlat_idx(0, d'''_base_addr, addr'))
{-2}  pm_phy'states(s''')
{-3}  pm_phy'states(s'')
{-4}  OK?(translate(0, d'''_base_addr, addr', access', segment_to_priv(d'))(s''))
{-5}  ∀ (s: (pm_phy'states)):
      OK?(read_data(pm_phy, pdb_r_data_type)(PDBR)(s)) ∧
      data(read_data(pm_phy, pdb_r_data_type)(PDBR)(s)) = d''
{-6}  ∀ (s: (pm''_states)):
      OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s)) ∧
      d' = data(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-7}  ∀ (cs: Segment_Reg_type, pdb_r: Pdb_r_type):
      ∀ (x1: Physical_memory):
        every[Physical_memory, [bool, Address]]
          (λ (x: Physical_memory): TRUE,
           λ (t: [bool, Address]):
             Mem?(t'2'type_of) ∧
             (0 ≤ t'2'offset ∧ t'2'offset < max_linear_offset) ∧
             IF t'1
               THEN aligned?(22)(offset(t'2))
               ELSE aligned?(12)(offset(t'2))
             ENDIF)
          (translate[Physical_memory, pm_phy]
            (0, pdb_r'_base_addr, addr', access', segment_to_priv(cs))(x1))
{-8}  ∀ (cs: Segment_Reg_type, pdb_r: Pdb_r_type, leaf: bool,
      base:
        {b: Memory_Address_4G |
          IF leaf
            THEN aligned?(-1 × (pdir_lvl [Physical_memory, pm_phy] × bits_per_level
              (offset(b)))
            ELSE aligned?(bits_per_level[Physical_memory, pm_phy] + pe_size)
              (offset(b))
            ENDIF}):
      ¬ leaf ⊃
      (∀ (x1: Physical_memory):
        every[Physical_memory, [bool, Address]]
          (λ (x: Physical_memory): TRUE,
           λ (t: [bool, Address]):
             Mem?(t'2'type_of) ∧ 0 ≤ t'2'offset ∧ t'2'offset < max_linear_offset)
          (translate[Physical_memory, pm_phy]
            (1, base, addr', access', segment_to_priv(cs))(x1)))
{-9}  pm''_mem = linear_pm
{-10} ∀ (s: (pm''_states)): OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-11} ∀ (s1, s2: (pm''_states)):
      data(read_data(pm_phy, segment_reg_data_type)(CS)(s1)) =
      data(read_data(pm_phy, segment_reg_data_type)(CS)(s2))
{-12} ∀ (s: (pm''_states)): OK?(read_data(pm_phy, pdb_r_data_type)(PDBR)(s))
{-13} ∀ (s1, s2: (pm''_states)):
      data(read_data(pm_phy, pdb_r_data_type)(PDBR)(s1)) =
      data(read_data(pm_phy, pdb_r_data_type)(PDBR)(s2))
{-14} ∀ (s1, s2: (pm''_states)):
      pe_in_pdir_range?(s1,
1752      restrict[Address, Memory_Address_4G, boolean]
        ((pm''_ro_addr ∪ pm''_rw_addr)))
      =
      pe_in_pdir_range?(s2,
        restrict[Address, Memory_Address_4G, boolean]
          ((pm''_ro_addr ∪ pm''_rw_addr)))
      ∧
      pe_in_ptab_range?(s1,

```

Using lemma `translate_same_result`,

Using lemma `translate_result_ok`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `linear_resolve_unchanged_pm_phy.1.1.3.3.1.1`.

linear_resolve_unchanged_pm_phy.1.1.3.3.1.2:

```

{-1} pm_phy' states(s''')
{-2} pm_phy' states(s'')
{-3} OK?(translate(0, d''base_addr, addr', access', segment_to_priv(d'))(s''))
{-4}  $\forall (s: (\text{pm\_phy}'\text{states})):$ 
      OK?(read_data(pm_phy, pdir_data_type)(PDBR)(s))  $\wedge$ 
      data(read_data(pm_phy, pdir_data_type)(PDBR)(s)) = d''
{-5}  $\forall (s: (\text{pm}'\text{states})):$ 
      OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s))  $\wedge$ 
      d' = data(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-6}  $\forall (cs: \text{Segment\_Reg\_type}, pdir: \text{Pdir\_type}):$ 
       $\forall (x_1: \text{Physical\_memory}):$ 
        every[Physical_memory, [bool, Address]]
          ( $\lambda (x: \text{Physical\_memory}): \text{TRUE},$ 
            $\lambda (t: [\text{bool}, \text{Address}]):$ 
             Mem?(t'2'type_of)  $\wedge$ 
             (0  $\leq$  t'2'offset  $\wedge$  t'2'offset < max_linear_offset)  $\wedge$ 
             IF t'1
               THEN aligned?(22)(offset(t'2))
               ELSE aligned?(12)(offset(t'2))
             ENDIF)
          (translate[Physical_memory, pm_phy]
           (0, pdir'base_addr, addr', access', segment_to_priv(cs))(x_1))
{-7}  $\forall (cs: \text{Segment\_Reg\_type}, pdir: \text{Pdir\_type}, \text{leaf}: \text{bool},$ 
      base:
        {b: Memory_Address_4G |
          IF leaf
            THEN aligned?(-1  $\times$  (pdir_lvl [Physical_memory, pm_phy]  $\times$  bits_per_level
              (offset(b)))
            ELSE aligned?(bits_per_level[Physical_memory, pm_phy] + pe_size)
              (offset(b))
            ENDIF}):
       $\neg$  leaf  $\supset$ 
      ( $\forall (x_1: \text{Physical\_memory}):$ 
        every[Physical_memory, [bool, Address]]
          ( $\lambda (x: \text{Physical\_memory}): \text{TRUE},$ 
            $\lambda (t: [\text{bool}, \text{Address}]):$ 
             Mem?(t'2'type_of)  $\wedge$  0  $\leq$  t'2'offset  $\wedge$  t'2'offset < max_linear_offset)
          (translate[Physical_memory, pm_phy]
           (1, base, addr', access', segment_to_priv(cs))(x_1)))
{-8} pm' mem = linear_pm
{-9}  $\forall (s: (\text{pm}'\text{states})): \text{OK?}(\text{read\_data}(\text{pm\_phy}, \text{segment\_reg\_data\_type})(\text{CS})(s))$ 
{-10}  $\forall (s_1, s_2: (\text{pm}'\text{states})):$ 
      data(read_data(pm_phy, segment_reg_data_type)(CS)(s_1)) =
      data(read_data(pm_phy, segment_reg_data_type)(CS)(s_2))
{-11}  $\forall (s: (\text{pm}'\text{states})): \text{OK?}(\text{read\_data}(\text{pm\_phy}, \text{pdir\_data\_type})(\text{PDBR})(s))$ 
{-12}  $\forall (s_1, s_2: (\text{pm}'\text{states})):$ 
      data(read_data(pm_phy, pdir_data_type)(PDBR)(s_1)) =
      data(read_data(pm_phy, pdir_data_type)(PDBR)(s_2))
{-13}  $\forall (s_1, s_2: (\text{pm}'\text{states})):$ 
      pe_in_pdir_range?(s_1,
        restrict[Address, Memory_Address_4G, boolean]
          ((pm'ro_addr  $\cup$  pm'rw_addr)))
      =
      pe_in_pdir_range?(s_2,
        restrict[Address, Memory_Address_4G, boolean]
          ((pm'ro_addr  $\cup$  pm'rw_addr)))
       $\wedge$ 
      pe_in_ptab_range?(s_1,
        restrict[Address, Memory_Address_4G, boolean]
          ((pm'ro_addr  $\cup$  pm'rw_addr)))
      =
      pe_in_ptab_range?(s_2,
        restrict[Address, Memory_Address_4G, boolean]
          ((pm'ro_addr  $\cup$  pm'rw_addr)))

```

Hiding formulas: 2,

Expanding the definition of `pe_in_pt_range?`,

Expanding the definition of `pe_in_pdir_range?`,

Instantiating quantified variables,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Instantiating the top quantifier in 1 with the terms: `addr'`,

we get 2 subgoals:

linear_resolve_unchanged_pm_phy.1.1.3.3.1.2.1:

```

{-1} 0 = 0
{-2} OK?(read_data(pm_phy, pabr_data_type)(PDBR)(s''))
{-3} pm_phy'states(s''')
{-4} pm_phy'states(s'')
{-5} OK?(translate(0, d''base_addr, addr', access', segment_to_priv(d''))(s''))
{-6} data(read_data(pm_phy, pabr_data_type)(PDBR)(s'')) = d''
{-7}  $\forall (s: (pm\_phy'states)):$ 
      OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s))  $\wedge$ 
      d' = data(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-8}  $\forall (cs: Segment\_Reg\_type, pabr: Pabr\_type):$ 
       $\forall (x_1: Physical\_memory):$ 
        every[Physical_memory, [bool, Address]]
          ( $\lambda (x: Physical\_memory):$  TRUE,
            $\lambda (t: [bool, Address]):$ 
             Mem?(t'2'type_of)  $\wedge$ 
             (0  $\leq$  t'2'offset  $\wedge$  t'2'offset < max_linear_offset)  $\wedge$ 
             IF t'1
               THEN aligned?(22)(offset(t'2))
             ELSE aligned?(12)(offset(t'2))
             ENDIF)
           (translate[Physical_memory, pm_phy]
            (0, pabr'base_addr, addr', access', segment_to_priv(cs))(x_1))
{-9}  $\forall (cs: Segment\_Reg\_type, pabr: Pabr\_type, leaf: bool,$ 
      base:
        {b: Memory_Address_4G |
          IF leaf
            THEN aligned?(-1  $\times$  (pdir_lvl [Physical_memory, pm_phy]  $\times$  bits_per_level
              (offset(b)))
            ELSE aligned?(bits_per_level[Physical_memory, pm_phy] + pe_size
              (offset(b)))
            ENDIF}):
       $\neg$  leaf  $\supset$ 
      ( $\forall (x_1: Physical\_memory):$ 
        every[Physical_memory, [bool, Address]]
          ( $\lambda (x: Physical\_memory):$  TRUE,
            $\lambda (t: [bool, Address]):$ 
             Mem?(t'2'type_of)  $\wedge$  0  $\leq$  t'2'offset  $\wedge$  t'2'offset < max_linear_offset)
          (translate[Physical_memory, pm_phy]
           (1, base, addr', access', segment_to_priv(cs))(x_1)))
{-10} pm' mem = linear_pm
{-11}  $\forall (s: (pm\_phy'states)):$  OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-12}  $\forall (s_1, s_2: (pm\_phy'states)):$ 
      data(read_data(pm_phy, segment_reg_data_type)(CS)(s_1)) =
      data(read_data(pm_phy, segment_reg_data_type)(CS)(s_2))
{-13}  $\forall (s: (pm\_phy'states)):$  OK?(read_data(pm_phy, pabr_data_type)(PDBR)(s))
{-14}  $\forall (s_1, s_2: (pm\_phy'states)):$ 
      data(read_data(pm_phy, pabr_data_type)(PDBR)(s_1)) =
      data(read_data(pm_phy, pabr_data_type)(PDBR)(s_2))
{-15}  $\forall (s_1, s_2: (pm\_phy'states)):$ 
      (( $\lambda (phy\_a: Memory\_Address\_4G):$ 
        OK?(read_data(pm_phy, pabr_data_type)(PDBR)(s_1))  $\wedge$ 
        ( $\exists$  (lin_a:
          (restrict[Address, Memory_Address_4G, boolean]
            ((pm'ro_addr  $\cup$  pm'rw_addr))))):
          xlat_idx(0, data(read_data(pm_phy, pabr_data_type)(PDBR)(s_1))'base_addr,
            lin_a)
          = phy_a))
      =
      ( $\lambda (phy\_a: Memory\_Address\_4G):$ 
        OK?(read_data(pm_phy, pabr_data_type)(PDBR)(s_2))  $\wedge$ 
        ( $\exists$  (lin_a:
          (restrict[Address, Memory_Address_4G, boolean]
            ((pm'ro_addr  $\cup$  pm'rw_addr))))):

```

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `linear_resolve_unchanged_pm_phy.1.1.3.3.1.2.1`.

linear_resolve_unchanged_pm_phy.1.1.3.3.1.2.2:

```

{-1} 0 = 0
{-2} OK?(read_data(pm_phy, pabr_data_type)(PDBR)(s''))
{-3} pm_phy'states(s'')
{-4} pm_phy'states(s'')
{-5} OK?(translate(0, d''base_addr, addr', access', segment_to_priv(d''))(s''))
{-6} data(read_data(pm_phy, pabr_data_type)(PDBR)(s'')) = d''
{-7}  $\forall (s: (pm\_phy'states)):$ 
      OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s))  $\wedge$ 
      d' = data(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-8}  $\forall (cs: Segment\_Reg\_type, pabr: Pabr\_type):$ 
       $\forall (x_1: Physical\_memory):$ 
        every[Physical_memory, [bool, Address]]
          ( $\lambda (x: Physical\_memory):$  TRUE,
            $\lambda (t: [bool, Address]):$ 
             Mem?(t'2'type_of)  $\wedge$ 
             (0  $\leq$  t'2'offset  $\wedge$  t'2'offset < max_linear_offset)  $\wedge$ 
             IF t'1
               THEN aligned?(22)(offset(t'2))
               ELSE aligned?(12)(offset(t'2))
             ENDIF)
          (translate[Physical_memory, pm_phy]
           (0, pabr'base_addr, addr', access', segment_to_priv(cs))(x_1))
{-9}  $\forall (cs: Segment\_Reg\_type, pabr: Pabr\_type, leaf: bool,$ 
      base:
        {b: Memory_Address_4G |
          IF leaf
            THEN aligned?(-1  $\times$  (pdir_lvl [Physical_memory, pm_phy]  $\times$  bits_per_level
              (offset(b)))
            ELSE aligned?(bits_per_level[Physical_memory, pm_phy] + pe_size
              (offset(b)))
            ENDIF}):
       $\neg$  leaf  $\supset$ 
      ( $\forall (x_1: Physical\_memory):$ 
        every[Physical_memory, [bool, Address]]
          ( $\lambda (x: Physical\_memory):$  TRUE,
            $\lambda (t: [bool, Address]):$ 
             Mem?(t'2'type_of)  $\wedge$  0  $\leq$  t'2'offset  $\wedge$  t'2'offset < max_linear_offset)
          (translate[Physical_memory, pm_phy]
           (1, base, addr', access', segment_to_priv(cs))(x_1)))
{-10} pm' mem = linear_pm
{-11}  $\forall (s: (pm\_phy'states)):$  OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-12}  $\forall (s_1, s_2: (pm\_phy'states)):$ 
      data(read_data(pm_phy, segment_reg_data_type)(CS)(s_1)) =
      data(read_data(pm_phy, segment_reg_data_type)(CS)(s_2))
{-13}  $\forall (s: (pm\_phy'states)):$  OK?(read_data(pm_phy, pabr_data_type)(PDBR)(s))
{-14}  $\forall (s_1, s_2: (pm\_phy'states)):$ 
      data(read_data(pm_phy, pabr_data_type)(PDBR)(s_1)) =
      data(read_data(pm_phy, pabr_data_type)(PDBR)(s_2))
{-15}  $\forall (s_1, s_2: (pm\_phy'states)):$ 
      (( $\lambda$  (phy_a: Memory_Address_4G):
        OK?(read_data(pm_phy, pabr_data_type)(PDBR)(s_1))  $\wedge$ 
        ( $\exists$  (lin_a:
          (restrict[Address, Memory_Address_4G, boolean]
            ((pm'ro_addr  $\cup$  pm'rw_addr))))):
          xlat_idx(0, data(read_data(pm_phy, pabr_data_type)(PDBR)(s_1))'base_addr,
            lin_a)
          = phy_a))
      =
      ( $\lambda$  (phy_a: Memory_Address_4G):
        OK?(read_data(pm_phy, pabr_data_type)(PDBR)(s_2))  $\wedge$ 
        ( $\exists$  (lin_a:
          (restrict[Address, Memory_Address_4G, boolean]
            ((pm'ro_addr  $\cup$  pm'rw_addr))))):

```

Keeping (-29 1) and hiding *,

Expanding the definition of restrict,

which is trivially true.

This completes the proof of `linear_resolve_unchanged_pm_phy.1.1.3.3.1.2.2`.

linear_resolve_unchanged_pm_phy.1.1.3.3.1.3:

```

{-1} pm_phy' states(s''')
{-2} pm_phy' states(s'')
{-3} OK?(translate(0, d''base_addr, addr', access', segment_to_priv(d'))(s''))
{-4} ∀ (s: (pm_phy' states)):
      OK?(read_data(pm_phy, pdbr_data_type)(PDBR)(s)) ∧
      data(read_data(pm_phy, pdbr_data_type)(PDBR)(s)) = d''
{-5} ∀ (s: (pm' states)):
      OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s)) ∧
      d' = data(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-6} ∀ (cs: Segment_Reg_type, pdbr: Pdbr_type):
      ∀ (x1: Physical_memory):
        every[Physical_memory, [bool, Address]]
          (λ (x: Physical_memory): TRUE,
           λ (t: [bool, Address]):
             Mem?(t'2'type_of) ∧
             (0 ≤ t'2'offset ∧ t'2'offset < max_linear_offset) ∧
             IF t'1
               THEN aligned?(22)(offset(t'2))
               ELSE aligned?(12)(offset(t'2))
             ENDIF)
          (translate[Physical_memory, pm_phy]
           (0, pdbr'base_addr, addr', access', segment_to_priv(cs))(x1))
{-7} ∀ (cs: Segment_Reg_type, pdbr: Pdbr_type, leaf: bool,
      base:
        {b: Memory_Address_4G |
          IF leaf
            THEN aligned?(-1 × (pdir_lvl [Physical_memory, pm_phy] × bits_per_level
              (offset(b))
            ELSE aligned?(bits_per_level[Physical_memory, pm_phy] + pe_size)
              (offset(b))
            ENDIF}):
      ¬ leaf ⊃
      (∀ (x1: Physical_memory):
        every[Physical_memory, [bool, Address]]
          (λ (x: Physical_memory): TRUE,
           λ (t: [bool, Address]):
             Mem?(t'2'type_of) ∧ 0 ≤ t'2'offset ∧ t'2'offset < max_linear_offset)
          (translate[Physical_memory, pm_phy]
           (1, base, addr', access', segment_to_priv(cs))(x1)))
{-8} pm' mem = linear_pm
{-9} ∀ (s: (pm' states)): OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-10} ∀ (s1, s2: (pm' states)):
      data(read_data(pm_phy, segment_reg_data_type)(CS)(s1)) =
      data(read_data(pm_phy, segment_reg_data_type)(CS)(s2))
{-11} ∀ (s: (pm' states)): OK?(read_data(pm_phy, pdbr_data_type)(PDBR)(s))
{-12} ∀ (s1, s2: (pm' states)):
      data(read_data(pm_phy, pdbr_data_type)(PDBR)(s1)) =
      data(read_data(pm_phy, pdbr_data_type)(PDBR)(s2))
{-13} ∀ (s1, s2: (pm' states)):
      pe_in_pdir_range?(s1,
        restrict[Address, Memory_Address_4G, boolean]
          ((pm'ro_addr ∪ pm'rw_addr)))
      =
      pe_in_pdir_range?(s2,
        restrict[Address, Memory_Address_4G, boolean]
          ((pm'ro_addr ∪ pm'rw_addr)))
      ∧
      pe_in_ptab_range?(s1,
        restrict[Address, Memory_Address_4G, boolean]
          ((pm'ro_addr ∪ pm'rw_addr)))
      =
      pe_in_ptab_range?(s2,
        restrict[Address, Memory_Address_4G, boolean]
          ((pm'ro_addr ∪ pm'rw_addr)))

```

Hiding formulas: 2,

Using lemma `xlat_idx_memory_address`,

Using lemma `xlat_idx_memory_address2`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `linear_resolve_unchanged_pm_phy.1.1.3.3.1.3`.

linear_resolve_unchanged_pm_phy.1.1.3.3.2:

```

{-1}  ∀ (s: (pm_phy' states)):
      OK?(read_data(pm_phy, pabr_data_type)(PDBR)(s)) ∧
      data(read_data(pm_phy, pabr_data_type)(PDBR)(s)) = d''
{-2}  ∀ (s: (pm' states)):
      OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s)) ∧
      d' = data(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-3}  ∀ (cs: Segment_Reg_type, pabr: Pabr_type):
      ∀ (x1: Physical_memory):
        every[Physical_memory, [bool, Address]]
          (λ (x: Physical_memory): TRUE,
           λ (t: [bool, Address]):
             Mem?(t'2'type_of) ∧
             (0 ≤ t'2'offset ∧ t'2'offset < max_linear_offset) ∧
             IF t'1
               THEN aligned?(22)(offset(t'2))
               ELSE aligned?(12)(offset(t'2))
             ENDIF)
          (translate[Physical_memory, pm_phy]
           (0, pabr'base_addr, addr', access', segment_to_priv(cs))(x1))
{-4}  ∀ (cs: Segment_Reg_type, pabr: Pabr_type, leaf: bool,
      base:
        {b: Memory_Address_4G |
          IF leaf
            THEN aligned?(-1 × (pdir_lvl [Physical_memory, pm_phy] × bits_per_level
              (offset(b))
            ELSE aligned?(bits_per_level[Physical_memory, pm_phy] + pe_size)
              (offset(b))
            ENDIF}):
      ¬ leaf ⊃
      (∀ (x1: Physical_memory):
        every[Physical_memory, [bool, Address]]
          (λ (x: Physical_memory): TRUE,
           λ (t: [bool, Address]):
             Mem?(t'2'type_of) ∧ 0 ≤ t'2'offset ∧ t'2'offset < max_linear_offset)
          (translate[Physical_memory, pm_phy]
           (1, base, addr', access', segment_to_priv(cs))(x1)))
{-5}  pm' mem = linear_pm
{-6}  ∀ (s: (pm' states)): OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-7}  ∀ (s1, s2: (pm' states)):
      data(read_data(pm_phy, segment_reg_data_type)(CS)(s1)) =
      data(read_data(pm_phy, segment_reg_data_type)(CS)(s2))
{-8}  ∀ (s: (pm' states)): OK?(read_data(pm_phy, pabr_data_type)(PDBR)(s))
{-9}  ∀ (s1, s2: (pm' states)):
      data(read_data(pm_phy, pabr_data_type)(PDBR)(s1)) =
      data(read_data(pm_phy, pabr_data_type)(PDBR)(s2))
{-10} ∀ (s1, s2: (pm' states)):
      pe_in_pdir_range?(s1,
        restrict[Address, Memory_Address_4G, boolean]
          ((pm'ro_addr ∪ pm'rw_addr)))
      =
      pe_in_pdir_range?(s2,
        restrict[Address, Memory_Address_4G, boolean]
          ((pm'ro_addr ∪ pm'rw_addr)))
      ∧
      pe_in_ptab_range?(s1,
        restrict[Address, Memory_Address_4G, boolean]
          ((pm'ro_addr ∪ pm'rw_addr)))
      =
      pe_in_ptab_range?(s2,
        restrict[Address, Memory_Address_4G, boolean]
          ((pm'ro_addr ∪ pm'rw_addr)))
{-11} ∀ (s: (pm' states), lvl: Level,
      a:

```

C.117 Proofs for Linear_Memory_Properties (challenge-linear.pvs)

Hiding formulas: 2,

Using lemma `translate_unchanged_pm_phy_except_pe`,

Simplifying, rewriting, and recording with decision procedures,

Using lemma `unchanged_memory_invariant_mono`,

Simplifying, rewriting, and recording with decision procedures,

Rewriting using `subset_reflexive`, matching in `*`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 5 subgoals:

linear_resolve_unchanged_pm_phy.1.1.3.3.2.1:

```

{-1}  unchanged_memory_invariant?(pm_phy' mem, pm_phy' states,
                                         singleton(expr_2_super(translate(0,
                                                                                   d''' base_addr,
                                                                                   addr',
                                                                                   access',
                                                                                   seg-
                                                                                   ment_to_priv(d')))),
                                         ((pm_phy' ro_addr ∪ pm_phy' rw_addr) \ address_block(xlat_idx(
{-2}  unchanged_memory_invariant?(pm_phy' mem, pm_phy' states,
                                         singleton(expr_2_super(translate(0,
                                                                                   d''' base_addr,
                                                                                   addr',
                                                                                   access',
                                                                                   seg-
                                                                                   ment_to_priv(d')))),
                                         addresses)
{-3}  ∀ (s: (pm_phy' states)):
      OK?(read_data(pm_phy, pdbr_data_type)(PDBR)(s)) ∧
      data(read_data(pm_phy, pdbr_data_type)(PDBR)(s)) = d''
{-4}  ∀ (s: (pm' states)):
      OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s)) ∧
      d' = data(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-5}  ∀ (cs: Segment_Reg_type, pdbr: Pdbr_type):
      ∀ (x1: Physical_memory):
        every[Physical_memory, [bool, Address]]
          (λ (x: Physical_memory): TRUE,
           λ (t: [bool, Address]):
             Mem?(t'2'type_of) ∧
             (0 ≤ t'2'offset ∧ t'2'offset < max_linear_offset) ∧
             IF t'1
               THEN aligned?(22)(offset(t'2))
               ELSE aligned?(12)(offset(t'2))
             ENDIF)
          (translate[Physical_memory, pm_phy]
            (0, pdbr' base_addr, addr', access', segment_to_priv(cs))(x1))
{-6}  ∀ (cs: Segment_Reg_type, pdbr: Pdbr_type, leaf: bool,
      base:
        {b: Memory_Address_4G |
          IF leaf
            THEN aligned?(-1 × (pdir_lvl [Physical_memory, pm_phy] × bits_per_level
              (offset(b)))
            ELSE aligned?(bits_per_level[Physical_memory, pm_phy] + pe_size)
              (offset(b))
            ENDIF}):
      ¬ leaf ⊃
      (∀ (x1: Physical_memory):
        every[Physical_memory, [bool, Address]]
          (λ (x: Physical_memory): TRUE,
           λ (t: [bool, Address]):
             Mem?(t'2'type_of) ∧ 0 ≤ t'2'offset ∧ t'2'offset < max_linear_offset)
          (translate[Physical_memory, pm_phy]
            (1, base, addr', access', segment_to_priv(cs))(x1)))
{-7}  pm' mem = linear_pm
{-8}  ∀ (s: (pm' states)): OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-9}  ∀ (s1, s2: (pm' states)):
      data(read_data(pm_phy, segment_reg_data_type)(CS)(s1)) =
      data(read_data(pm_phy, segment_reg_data_type)(CS)(s2))
{-10} ∀ (s: (pm' states)): OK?(read_data(pm_phy, pdbr_data_type)(PDBR)(s))
{-11} ∀ (s1, s2: (pm' states)):
      data(read_data(pm_phy, pdbr_data_type)(PDBR)(s1)) =
      data(read_data(pm_phy, pdbr_data_type)(PDBR)(s2))
{-12} ∀ (s1, s2: (pm' states)):
      pe_in_pdir_range?(s1,
restrict[Address, Memory_Address_4G, boolean]

```

C.117 Proofs for Linear_Memory_Properties (challenge-linear.pvs)

```
Case splitting on Transformer_Super_Embedding_Expr [Physical_memory, [bool, Ad-
dress_Helpers.Address]].expr_2_super (Linear_Memory[Physical_memory, pm_phy].translate (0,
d!2'base_addr, addr!1, access!1, Segment_Types.segment_to_priv(d!1))) = Transformer_Super_Embedding_Expr
[Physical_memory, [leaf: bool, {b: Memory_Address_4G | IF leaf THEN aligned?(-1 *
(pdir_lvl[Physical_memory, pm_phy] * bits_per_level [Physical_memory, pm_phy]) - bits_per_level[Physical_memory,
pm_phy] + bus_width) (offset(b)) ELSE aligned?(bits_per_level[Physical_memory, pm_phy] +
pe_size) (offset(b)) ENDIF}]].expr_2_super (Linear_Memory[Physical_memory, pm_phy].translate
(0, d!2'base_addr, addr!1, access!1, Segment_Types.segment_to_priv(d!1))),
```

we get 2 subgoals:

linear_resolve_unchanged_pm_phy.1.1.3.3.2.1.1:

```

{-1} Transformer_Super_Embedding_Expr[Physical_memory, [bool, Ad-
dress_Helpers.Address]].expr_2_super
      (Linear_Memory[Physical_memory, pm_phy].translate
        (0, d''base_addr, addr', access', Segment_Types.segment_to_priv(d')))
      =
      Transformer_Super_Embedding_Expr
        [Physical_memory,
          [leaf: bool,
            {b: Memory_Address_4G |
              IF leaf
                THEN aligned?(-1 × (pdir_lvl [Physical_memory, pm_phy] × bits_per_level
                  (offset(b))
                ELSE aligned?(bits_per_level[Physical_memory, pm_phy] + pe_size)
                  (offset(b))
                ENDIF}]]].expr_2_super
        (Linear_Memory[Physical_memory, pm_phy].translate
          (0, d''base_addr, addr', access', Segment_Types.segment_to_priv(d')))
{-2} unchanged_memory_invariant?(pm_phy'mem, pm_phy'states,
      singleton(expr_2_super(translate(0,
      d''base_addr,
      addr',
      access',
      seg-
      ment_to_priv(d')))),
      ((pm_phy'ro_addr ∪ pm_phy'rw_addr) \ address_block(xlat_idx(
{-3} unchanged_memory_invariant?(pm_phy'mem, pm_phy'states,
      singleton(expr_2_super(translate(0,
      d''base_addr,
      addr',
      access',
      seg-
      ment_to_priv(d')))),
      addresses)
{-4} ∀ (s: (pm_phy'states)):
      OK?(read_data(pm_phy, pdir_data_type)(PDBR)(s)) ∧
      data(read_data(pm_phy, pdir_data_type)(PDBR)(s)) = d''
{-5} ∀ (s: (pm'states)):
      OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s)) ∧
      d' = data(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-6} ∀ (cs: Segment_Reg_type, pdir: Pdir_type):
      ∀ (x1: Physical_memory):
        every[Physical_memory, [bool, Address]]
          (λ (x: Physical_memory): TRUE,
            λ (t: [bool, Address]):
              Mem?(t'2'type_of) ∧
              (0 ≤ t'2'offset ∧ t'2'offset < max_linear_offset) ∧
              IF t'1
                THEN aligned?(22)(offset(t'2))
                ELSE aligned?(12)(offset(t'2))
                ENDIF)
          (translate[Physical_memory, pm_phy]
            (0, pdir'base_addr, addr', access', segment_to_priv(cs))(x1))
{-7} ∀ (cs: Segment_Reg_type, pdir: Pdir_type, leaf: bool,
      base:
      {b: Memory_Address_4G |
        IF leaf
          THEN aligned?(-1 × (pdir_lvl [Physical_memory, pm_phy] × bits_per_level
            (offset(b))
          ELSE aligned?(bits_per_level[Physical_memory, pm_phy] + pe_size)
            (offset(b))
          ENDIF}):
      ¬ leaf ⊃
      (∀ (x1: Physical_memory):

```

C.117 Proofs for Linear_Memory_Properties (challenge-linear.pvs)

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `linear_resolve_unchanged_pm_phy.1.1.3.3.2.1.1`.

linear_resolve_unchanged_pm_phy.1.1.3.3.2.1.2:

```

{-1}  unchanged_memory_invariant?(pm_phy'mem, pm_phy'states,
                                         singleton(expr_2_super(translate(0,
                                                                                   d''base_addr,
                                                                                   addr',
                                                                                   access',
                                                                                   seg-
                                                                                   ment_to_priv(d')))),
                                         ((pm_phy'ro_addr ∪ pm_phy'rw_addr) \ address_block(xlat_idx(
{-2}  unchanged_memory_invariant?(pm_phy'mem, pm_phy'states,
                                         singleton(expr_2_super(translate(0,
                                                                                   d''base_addr,
                                                                                   addr',
                                                                                   access',
                                                                                   seg-
                                                                                   ment_to_priv(d')))),
                                         addresses)
{-3}  ∀ (s: (pm_phy'states)):
      OK?(read_data(pm_phy, pdbr_data_type)(PDBR)(s)) ∧
      data(read_data(pm_phy, pdbr_data_type)(PDBR)(s)) = d''
{-4}  ∀ (s: (pm' states)):
      OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s)) ∧
      d' = data(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-5}  ∀ (cs: Segment_Reg_type, pdbr: Pdbr_type):
      ∀ (x1: Physical_memory):
        every[Physical_memory, [bool, Address]]
          (λ (x: Physical_memory): TRUE,
           λ (t: [bool, Address]):
             Mem?(t'2'type_of) ∧
             (0 ≤ t'2'offset ∧ t'2'offset < max_linear_offset) ∧
             IF t'1
               THEN aligned?(22)(offset(t'2))
               ELSE aligned?(12)(offset(t'2))
             ENDIF)
          (translate[Physical_memory, pm_phy]
            (0, pdbr'base_addr, addr', access', segment_to_priv(cs))(x1))
{-6}  ∀ (cs: Segment_Reg_type, pdbr: Pdbr_type, leaf: bool,
      base:
        {b: Memory_Address_4G |
          IF leaf
            THEN aligned?(-1 × (pdir_lvl [Physical_memory, pm_phy] × bits_per_level
              (offset(b)))
            ELSE aligned?(bits_per_level[Physical_memory, pm_phy] + pe_size)
              (offset(b))
            ENDIF}):
      ¬ leaf ⊃
      (∀ (x1: Physical_memory):
        every[Physical_memory, [bool, Address]]
          (λ (x: Physical_memory): TRUE,
           λ (t: [bool, Address]):
             Mem?(t'2'type_of) ∧ 0 ≤ t'2'offset ∧ t'2'offset < max_linear_offset)
          (translate[Physical_memory, pm_phy]
            (1, base, addr', access', segment_to_priv(cs))(x1)))
{-7}  pm'mem = linear_pm
{-8}  ∀ (s: (pm' states)): OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-9}  ∀ (s1, s2: (pm' states)):
      data(read_data(pm_phy, segment_reg_data_type)(CS)(s1)) =
      data(read_data(pm_phy, segment_reg_data_type)(CS)(s2))
1768 {-10} ∀ (s: (pm' states)): OK?(read_data(pm_phy, pdbr_data_type)(PDBR)(s))
{-11} ∀ (s1, s2: (pm' states)):
      data(read_data(pm_phy, pdbr_data_type)(PDBR)(s1)) =
      data(read_data(pm_phy, pdbr_data_type)(PDBR)(s2))
{-12} ∀ (s1, s2: (pm' states)):
      pe_in_pdir_range?(s1,
restrict[Address, Memory_Address_4G, boolean]

```

Keeping (1) and hiding *,

Applying extensionality,

Expanding the definition(s) of (expr_2_super expr_2_super_res has_next_state),

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `linear_resolve_unchanged_pm_phy.1.1.3.3.2.1.2`.

linear_resolve_unchanged_pm_phy.1.1.3.3.2.2:

```

{-1}  ∀ (s: (pm_phy' states)):
      OK?(read_data(pm_phy, pabr_data_type)(PDBR)(s)) ∧
      data(read_data(pm_phy, pabr_data_type)(PDBR)(s)) = d''
{-2}  ∀ (s: (pm' states)):
      OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s)) ∧
      d' = data(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-3}  ∀ (cs: Segment_Reg_type, pabr: Pabr_type):
      ∀ (x1: Physical_memory):
        every[Physical_memory, [bool, Address]]
          (λ (x: Physical_memory): TRUE,
           λ (t: [bool, Address]):
             Mem?(t'2'type_of) ∧
             (0 ≤ t'2'offset ∧ t'2'offset < max_linear_offset) ∧
             IF t'1
               THEN aligned?(22)(offset(t'2))
               ELSE aligned?(12)(offset(t'2))
             ENDIF)
          (translate[Physical_memory, pm_phy]
           (0, pabr'base_addr, addr', access', segment_to_priv(cs))(x1))
{-4}  ∀ (cs: Segment_Reg_type, pabr: Pabr_type, leaf: bool,
      base:
        {b: Memory_Address_4G |
          IF leaf
            THEN aligned?(-1 × (pdir_lvl [Physical_memory, pm_phy] × bits_per_level
              (offset(b))
            ELSE aligned?(bits_per_level[Physical_memory, pm_phy] + pe_size)
              (offset(b))
            ENDIF}):
      ¬ leaf ⊃
      (∀ (x1: Physical_memory):
        every[Physical_memory, [bool, Address]]
          (λ (x: Physical_memory): TRUE,
           λ (t: [bool, Address]):
             Mem?(t'2'type_of) ∧ 0 ≤ t'2'offset ∧ t'2'offset < max_linear_offset)
          (translate[Physical_memory, pm_phy]
           (1, base, addr', access', segment_to_priv(cs))(x1)))
{-5}  pm' mem = linear_pm
{-6}  ∀ (s: (pm' states)): OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-7}  ∀ (s1, s2: (pm' states)):
      data(read_data(pm_phy, segment_reg_data_type)(CS)(s1)) =
      data(read_data(pm_phy, segment_reg_data_type)(CS)(s2))
{-8}  ∀ (s: (pm' states)): OK?(read_data(pm_phy, pabr_data_type)(PDBR)(s))
{-9}  ∀ (s1, s2: (pm' states)):
      data(read_data(pm_phy, pabr_data_type)(PDBR)(s1)) =
      data(read_data(pm_phy, pabr_data_type)(PDBR)(s2))
{-10} ∀ (s1, s2: (pm' states)):
      pe_in_pdir_range?(s1,
        restrict[Address, Memory_Address_4G, boolean]
          ((pm'ro_addr ∪ pm'rw_addr)))
      =
      pe_in_pdir_range?(s2,
        restrict[Address, Memory_Address_4G, boolean]
          ((pm'ro_addr ∪ pm'rw_addr)))
      ∧
      pe_in_ptab_range?(s1,
        restrict[Address, Memory_Address_4G, boolean]
          ((pm'ro_addr ∪ pm'rw_addr)))
      =
      pe_in_ptab_range?(s2,
        restrict[Address, Memory_Address_4G, boolean]
          ((pm'ro_addr ∪ pm'rw_addr)))
{-11} ∀ (s: (pm' states), lvl: Level,
      a:

```

Hiding formulas: (1 3),

Repeatedly Skolemizing and flattening,

Instantiating the top quantifier in -14 with the terms: (s1!1 s2!1 0 xlat_idx(0, d!2'base_addr, addr!1)),

we get 2 subgoals:

linear_resolve_unchanged_pm_phy.1.1.3.3.2.2.1:

```

{-1} pm' 'states(s1)
{-2} pm' 'states(s2)
{-3} ∀ (s: (pm_phy 'states)):
      OK?(read_data(pm_phy, pdir_data_type)(PDBR)(s)) ∧
      data(read_data(pm_phy, pdir_data_type)(PDBR)(s)) = d''
{-4} ∀ (s: (pm' 'states)):
      OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s)) ∧
      d' = data(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-5} ∀ (cs: Segment_Reg_type, pdir: Pdir_type):
      ∀ (x1: Physical_memory):
        every[Physical_memory, [bool, Address]]
          (λ (x: Physical_memory): TRUE,
           λ (t: [bool, Address]):
             Mem?(t'2'type_of) ∧
             (0 ≤ t'2'offset ∧ t'2'offset < max_linear_offset) ∧
             IF t'1
               THEN aligned?(22)(offset(t'2))
               ELSE aligned?(12)(offset(t'2))
             ENDIF)
          (translate[Physical_memory, pm_phy]
           (0, pdir'base_addr, addr', access', segment_to_priv(cs))(x1))
{-6} ∀ (cs: Segment_Reg_type, pdir: Pdir_type, leaf: bool,
      base:
        {b: Memory_Address_4G |
          IF leaf
            THEN aligned?(-1 × (pdir_lvl [Physical_memory, pm_phy] × bits_per_level
              (offset(b)))
            ELSE aligned?(bits_per_level[Physical_memory, pm_phy] + pe_size)
              (offset(b))
            ENDIF}):
      ¬ leaf ⊃
      (∀ (x1: Physical_memory):
        every[Physical_memory, [bool, Address]]
          (λ (x: Physical_memory): TRUE,
           λ (t: [bool, Address]):
             Mem?(t'2'type_of) ∧ 0 ≤ t'2'offset ∧ t'2'offset < max_linear_offset)
          (translate[Physical_memory, pm_phy]
           (1, base, addr', access', segment_to_priv(cs))(x1)))
{-7} pm' 'mem = linear_pm
{-8} ∀ (s: (pm' 'states)): OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-9} ∀ (s1, s2: (pm' 'states)):
      data(read_data(pm_phy, segment_reg_data_type)(CS)(s1)) =
      data(read_data(pm_phy, segment_reg_data_type)(CS)(s2))
{-10} ∀ (s: (pm' 'states)): OK?(read_data(pm_phy, pdir_data_type)(PDBR)(s))
{-11} ∀ (s1, s2: (pm' 'states)):
      data(read_data(pm_phy, pdir_data_type)(PDBR)(s1)) =
      data(read_data(pm_phy, pdir_data_type)(PDBR)(s2))
{-12} ∀ (s1, s2: (pm' 'states)):
      pe_in_pdir_range?(s1,
        restrict[Address, Memory_Address_4G, boolean]
          ((pm'ro_addr ∪ pm'rw_addr)))
      =
      pe_in_pdir_range?(s2,
        restrict[Address, Memory_Address_4G, boolean]
          ((pm'ro_addr ∪ pm'rw_addr)))
      ∧
      pe_in_ptab_range?(s1,
        restrict[Address, Memory_Address_4G, boolean]
          ((pm'ro_addr ∪ pm'rw_addr)))
      =
      pe_in_ptab_range?(s2,
        restrict[Address, Memory_Address_4G, boolean]
          ((pm'ro_addr ∪ pm'rw_addr)))
{-13} ∀ (s: (pm' 'states)):
      pe_in_pdir_range?(s,
        restrict[Address, Memory_Address_4G, boolean]
          ((pm'ro_addr ∪ pm'rw_addr)))
      =
      pe_in_ptab_range?(s,
        restrict[Address, Memory_Address_4G, boolean]
          ((pm'ro_addr ∪ pm'rw_addr)))

```

which is trivially true.

This completes the proof of `linear_resolve_unchanged_pm_phy.1.1.3.3.2.2.1`.

linear_resolve_unchanged_pm_phy.1.1.3.3.2.2.2:

```

{-1} pm' 'states(s'_1)
{-2} pm' 'states(s'_2)
{-3}  $\forall (s: (\text{pm\_phy}' \text{states})):$ 
      OK?(read_data(pm_phy, pdir_data_type)(PDBR)(s))  $\wedge$ 
      data(read_data(pm_phy, pdir_data_type)(PDBR)(s)) = d''
{-4}  $\forall (s: (\text{pm}' \text{states})):$ 
      OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s))  $\wedge$ 
      d' = data(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-5}  $\forall (cs: \text{Segment\_Reg\_type}, pdir: \text{Pdir\_type}):$ 
       $\forall (x_1: \text{Physical\_memory}):$ 
        every[Physical_memory, [bool, Address]]
          ( $\lambda (x: \text{Physical\_memory}): \text{TRUE},$ 
            $\lambda (t: [\text{bool}, \text{Address}]):$ 
             Mem?(t'2'type_of)  $\wedge$ 
             (0  $\leq$  t'2'offset  $\wedge$  t'2'offset < max_linear_offset)  $\wedge$ 
             IF t'1
               THEN aligned?(22)(offset(t'2))
               ELSE aligned?(12)(offset(t'2))
               ENDIF)
           (translate[Physical_memory, pm_phy]
            (0, pdir'base_addr, addr', access', segment_to_priv(cs))(x_1)))
{-6}  $\forall (cs: \text{Segment\_Reg\_type}, pdir: \text{Pdir\_type}, \text{leaf}: \text{bool},$ 
      base:
        {b: Memory_Address_4G |
          IF leaf
            THEN aligned?(-1  $\times$  (pdir_lvl [Physical_memory, pm_phy]  $\times$  bits_per_level)
                          (offset(b)))
            ELSE aligned?(bits_per_level[Physical_memory, pm_phy] + pe_size)
                          (offset(b)))
          ENDIF}):
       $\neg \text{leaf} \supset$ 
      ( $\forall (x_1: \text{Physical\_memory}):$ 
       every[Physical_memory, [bool, Address]]
         ( $\lambda (x: \text{Physical\_memory}): \text{TRUE},$ 
           $\lambda (t: [\text{bool}, \text{Address}]):$ 
            Mem?(t'2'type_of)  $\wedge$  0  $\leq$  t'2'offset  $\wedge$  t'2'offset < max_linear_offset)
         (translate[Physical_memory, pm_phy]
          (1, base, addr', access', segment_to_priv(cs))(x_1)))
{-7} pm' 'mem = linear_pm
{-8}  $\forall (s: (\text{pm}' \text{states})): \text{OK?}(\text{read\_data}(\text{pm\_phy}, \text{segment\_reg\_data\_type})(\text{CS})(s))$ 
{-9}  $\forall (s_1, s_2: (\text{pm}' \text{states})):$ 
      data(read_data(pm_phy, segment_reg_data_type)(CS)(s_1)) =
      data(read_data(pm_phy, segment_reg_data_type)(CS)(s_2))
{-10}  $\forall (s: (\text{pm}' \text{states})): \text{OK?}(\text{read\_data}(\text{pm\_phy}, \text{pdir\_data\_type})(\text{PDBR})(s))$ 
{-11}  $\forall (s_1, s_2: (\text{pm}' \text{states})):$ 
      data(read_data(pm_phy, pdir_data_type)(PDBR)(s_1)) =
      data(read_data(pm_phy, pdir_data_type)(PDBR)(s_2))
{-12}  $\forall (s_1, s_2: (\text{pm}' \text{states})):$ 
      pe_in_pdir_range?(s_1,
                        restrict[Address, Memory_Address_4G, boolean]
                          ((pm'ro_addr  $\cup$  pm'rw_addr)))
      =
      pe_in_pdir_range?(s_2,
                        restrict[Address, Memory_Address_4G, boolean]
                          ((pm'ro_addr  $\cup$  pm'rw_addr)))
       $\wedge$ 
      pe_in_ptab_range?(s_1,
                        restrict[Address, Memory_Address_4G, boolean]
                          ((pm'ro_addr  $\cup$  pm'rw_addr)))
      =
      pe_in_ptab_range?(s_2,
                        restrict[Address, Memory_Address_4G, boolean]
                          ((pm'ro_addr  $\cup$  pm'rw_addr)))
{-13}  $\forall (s_1, s_2: (\text{pm}' \text{states})): \text{OK?}(\text{read\_data}(\text{pm\_phy}, \text{segment\_reg\_data\_type})(\text{CS})(s_1))$ 

```

C.117 Proofs for Linear_Memory_Properties (challenge-linear.pvs)

Adding type constraints for `d!2'base_addr`,

Instantiating the top quantifier in `-7` with the terms: `(s1!1)`,

Using lemma `xlat_pe_in_pdir_range`,

Expanding the definition of `pe_in_pt_range?`,

Using lemma `xlat_idx_memory_address`,

Using lemma `xlat_idx_memory_address2`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `linear_resolve_unchanged_pm_phy.1.1.3.3.2.2.2`.

linear_resolve_unchanged_pm_phy.1.1.3.3.2.3:

```

{-1}  ∀ (s: (pm_phy' states)):
      OK?(read_data(pm_phy, pabr_data_type)(PDBR)(s)) ∧
      data(read_data(pm_phy, pabr_data_type)(PDBR)(s)) = d''
{-2}  ∀ (s: (pm' states)):
      OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s)) ∧
      d' = data(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-3}  ∀ (cs: Segment_Reg_type, pabr: Pabr_type):
      ∀ (x1: Physical_memory):
        every[Physical_memory, [bool, Address]]
          (λ (x: Physical_memory): TRUE,
           λ (t: [bool, Address]):
             Mem?(t'2'type_of) ∧
             (0 ≤ t'2'offset ∧ t'2'offset < max_linear_offset) ∧
             IF t'1
               THEN aligned?(22)(offset(t'2))
               ELSE aligned?(12)(offset(t'2))
             ENDIF)
          (translate[Physical_memory, pm_phy]
           (0, pabr'base_addr, addr', access', segment_to_priv(cs))(x1))
{-4}  ∀ (cs: Segment_Reg_type, pabr: Pabr_type, leaf: bool,
      base:
        {b: Memory_Address_4G |
          IF leaf
            THEN aligned?(-1 × (pdir_lvl [Physical_memory, pm_phy] × bits_per_level
              (offset(b))
            ELSE aligned?(bits_per_level[Physical_memory, pm_phy] + pe_size)
              (offset(b))
            ENDIF}):
      ¬ leaf ⊃
      (∀ (x1: Physical_memory):
        every[Physical_memory, [bool, Address]]
          (λ (x: Physical_memory): TRUE,
           λ (t: [bool, Address]):
             Mem?(t'2'type_of) ∧ 0 ≤ t'2'offset ∧ t'2'offset < max_linear_offset)
          (translate[Physical_memory, pm_phy]
           (1, base, addr', access', segment_to_priv(cs))(x1)))
{-5}  pm' mem = linear_pm
{-6}  ∀ (s: (pm' states)): OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-7}  ∀ (s1, s2: (pm' states)):
      data(read_data(pm_phy, segment_reg_data_type)(CS)(s1)) =
      data(read_data(pm_phy, segment_reg_data_type)(CS)(s2))
{-8}  ∀ (s: (pm' states)): OK?(read_data(pm_phy, pabr_data_type)(PDBR)(s))
{-9}  ∀ (s1, s2: (pm' states)):
      data(read_data(pm_phy, pabr_data_type)(PDBR)(s1)) =
      data(read_data(pm_phy, pabr_data_type)(PDBR)(s2))
{-10} ∀ (s1, s2: (pm' states)):
      pe_in_pdir_range?(s1,
        restrict[Address, Memory_Address_4G, boolean]
          ((pm'ro_addr ∪ pm'rw_addr)))
      =
      pe_in_pdir_range?(s2,
        restrict[Address, Memory_Address_4G, boolean]
          ((pm'ro_addr ∪ pm'rw_addr)))
      ∧
      pe_in_ptab_range?(s1,
        restrict[Address, Memory_Address_4G, boolean]
          ((pm'ro_addr ∪ pm'rw_addr)))
      =
      pe_in_ptab_range?(s2,
        restrict[Address, Memory_Address_4G, boolean]
          ((pm'ro_addr ∪ pm'rw_addr)))
{-11} ∀ (s: (pm' states), lvl: Level,
      a:

```

Hiding formulas: (1 3),

Repeatedly Skolemizing and flattening,

Instantiating the top quantifier in -12 with the terms: (s!2 0 xlat_idx(0, d!2'base_addr, addr!1)),

we get 2 subgoals:

linear_resolve_unchanged_pm_phy.1.1.3.3.2.3.1:

```

{-1} pm' 'states(s'')
{-2} ∀ (s: (pm_phy 'states)):
      OK?(read_data(pm_phy, pdir_data_type)(PDBR)(s)) ∧
      data(read_data(pm_phy, pdir_data_type)(PDBR)(s)) = d''
{-3} ∀ (s: (pm' 'states)):
      OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s)) ∧
      d' = data(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-4} ∀ (cs: Segment_Reg_type, pdir: Pdir_type):
      ∀ (x1: Physical_memory):
        every[Physical_memory, [bool, Address]]
          (λ (x: Physical_memory): TRUE,
           λ (t: [bool, Address]):
             Mem?(t'2'type_of) ∧
             (0 ≤ t'2'offset ∧ t'2'offset < max_linear_offset) ∧
             IF t'1
               THEN aligned?(22)(offset(t'2))
               ELSE aligned?(12)(offset(t'2))
             ENDIF)
          (translate[Physical_memory, pm_phy]
           (0, pdir'base_addr, addr', access', segment_to_priv(cs))(x1))
{-5} ∀ (cs: Segment_Reg_type, pdir: Pdir_type, leaf: bool,
      base:
        {b: Memory_Address_4G |
          IF leaf
            THEN aligned?(-1 × (pdir_lvl [Physical_memory, pm_phy] × bits_per_level
              (offset(b))
            ELSE aligned?(bits_per_level[Physical_memory, pm_phy] + pe_size)
              (offset(b))
            ENDIF}):
      ¬ leaf ⊃
      (∀ (x1: Physical_memory):
        every[Physical_memory, [bool, Address]]
          (λ (x: Physical_memory): TRUE,
           λ (t: [bool, Address]):
             Mem?(t'2'type_of) ∧ 0 ≤ t'2'offset ∧ t'2'offset < max_linear_offset)
          (translate[Physical_memory, pm_phy]
           (1, base, addr', access', segment_to_priv(cs))(x1)))
{-6} pm' 'mem = linear_pm
{-7} ∀ (s: (pm' 'states)): OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-8} ∀ (s1, s2: (pm' 'states)):
      data(read_data(pm_phy, segment_reg_data_type)(CS)(s1)) =
      data(read_data(pm_phy, segment_reg_data_type)(CS)(s2))
{-9} ∀ (s: (pm' 'states)): OK?(read_data(pm_phy, pdir_data_type)(PDBR)(s))
{-10} ∀ (s1, s2: (pm' 'states)):
      data(read_data(pm_phy, pdir_data_type)(PDBR)(s1)) =
      data(read_data(pm_phy, pdir_data_type)(PDBR)(s2))
{-11} ∀ (s1, s2: (pm' 'states)):
      pe_in_pdir_range?(s1,
        restrict[Address, Memory_Address_4G, boolean]
          ((pm'ro_addr ∪ pm'rw_addr)))
      =
      pe_in_pdir_range?(s2,
        restrict[Address, Memory_Address_4G, boolean]
          ((pm'ro_addr ∪ pm'rw_addr)))
      ∧
      pe_in_ptab_range?(s1,
        restrict[Address, Memory_Address_4G, boolean]
          ((pm'ro_addr ∪ pm'rw_addr)))
      =
      pe_in_ptab_range?(s2,
        restrict[Address, Memory_Address_4G, boolean]
          ((pm'ro_addr ∪ pm'rw_addr)))
{-12} OK?(read_data(pm_phy, paging_data_type(0))(xlat_idx(0, d''base_addr, addr'))(s''))

```

which is trivially true.

This completes the proof of `linear_resolve_unchanged_pm_phy.1.1.3.3.2.3.1`.

linear_resolve_unchanged_pm_phy.1.1.3.3.2.3.2:

```

{-1} pm' 'states(s'')
{-2}  $\forall (s: (\text{pm\_phy}' \text{states})):$ 
      OK?(read_data(pm_phy, pdir_data_type)(PDBR)(s))  $\wedge$ 
      data(read_data(pm_phy, pdir_data_type)(PDBR)(s)) = d'
{-3}  $\forall (s: (\text{pm}' \text{states})):$ 
      OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s))  $\wedge$ 
      d' = data(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-4}  $\forall (cs: \text{Segment\_Reg\_type}, pdir: \text{Pdir\_type}):$ 
       $\forall (x_1: \text{Physical\_memory}):$ 
        every[Physical_memory, [bool, Address]]
          ( $\lambda (x: \text{Physical\_memory}): \text{TRUE},$ 
            $\lambda (t: [\text{bool}, \text{Address}]):$ 
             Mem?(t'2'type_of)  $\wedge$ 
             (0  $\leq$  t'2'offset  $\wedge$  t'2'offset < max_linear_offset)  $\wedge$ 
             IF t'1
               THEN aligned?(22)(offset(t'2))
               ELSE aligned?(12)(offset(t'2))
             ENDIF)
          (translate[Physical_memory, pm_phy]
           (0, pdir'base_addr, addr', access', segment_to_priv(cs))(x_1))
{-5}  $\forall (cs: \text{Segment\_Reg\_type}, pdir: \text{Pdir\_type}, \text{leaf}: \text{bool},$ 
      base:
        {b: Memory_Address_4G |
          IF leaf
            THEN aligned?(-1  $\times$  (pdir_lvl [Physical_memory, pm_phy]  $\times$  bits_per_level
              (offset(b)))
            ELSE aligned?(bits_per_level[Physical_memory, pm_phy] + pe_size)
              (offset(b))
            ENDIF}):
       $\neg \text{leaf} \supset$ 
      ( $\forall (x_1: \text{Physical\_memory}):$ 
        every[Physical_memory, [bool, Address]]
          ( $\lambda (x: \text{Physical\_memory}): \text{TRUE},$ 
            $\lambda (t: [\text{bool}, \text{Address}]):$ 
             Mem?(t'2'type_of)  $\wedge$  0  $\leq$  t'2'offset  $\wedge$  t'2'offset < max_linear_offset)
          (translate[Physical_memory, pm_phy]
           (1, base, addr', access', segment_to_priv(cs))(x_1)))
{-6} pm' 'mem = linear_pm
{-7}  $\forall (s: (\text{pm}' \text{states})): \text{OK?}(\text{read\_data}(\text{pm\_phy}, \text{segment\_reg\_data\_type})(\text{CS})(s))$ 
{-8}  $\forall (s_1, s_2: (\text{pm}' \text{states})):$ 
      data(read_data(pm_phy, segment_reg_data_type)(CS)(s_1)) =
      data(read_data(pm_phy, segment_reg_data_type)(CS)(s_2))
{-9}  $\forall (s: (\text{pm}' \text{states})): \text{OK?}(\text{read\_data}(\text{pm\_phy}, \text{pdir\_data\_type})(\text{PDBR})(s))$ 
{-10}  $\forall (s_1, s_2: (\text{pm}' \text{states})):$ 
      data(read_data(pm_phy, pdir_data_type)(PDBR)(s_1)) =
      data(read_data(pm_phy, pdir_data_type)(PDBR)(s_2))
{-11}  $\forall (s_1, s_2: (\text{pm}' \text{states})):$ 
      pe_in_pdir_range?(s_1,
        restrict[Address, Memory_Address_4G, boolean]
          ((pm'ro_addr  $\cup$  pm'rw_addr)))
      =
      pe_in_pdir_range?(s_2,
        restrict[Address, Memory_Address_4G, boolean]
          ((pm'ro_addr  $\cup$  pm'rw_addr)))
       $\wedge$ 
      pe_in_ptab_range?(s_1,
        restrict[Address, Memory_Address_4G, boolean]
          ((pm'ro_addr  $\cup$  pm'rw_addr)))
      =
      pe_in_ptab_range?(s_2,
        restrict[Address, Memory_Address_4G, boolean]
          ((pm'ro_addr  $\cup$  pm'rw_addr)))
{-12}  $\forall (s_1, s_2: (\text{pm}' \text{states}), \text{lvl}: \text{Level},$ 

```

C.117 Proofs for Linear_Memory_Properties (challenge-linear.pvs)

Adding type constraints for `d!2'base_addr`,

Instantiating quantified variables,

Using lemma `xlat_pe_in_pdir_range`,

Using lemma `xlat_idx_memory_address`,

Using lemma `xlat_idx_memory_address2`,

Expanding the definition of `pe_in_pt_range?`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `linear_resolve_unchanged_pm_phy.1.1.3.3.2.3.2`.

linear_resolve_unchanged_pm_phy.1.1.3.3.2.4:

```

{-1}  ∀ (s: (pm_phy' states)):
      OK?(read_data(pm_phy, pabr_data_type)(PDBR)(s)) ∧
      data(read_data(pm_phy, pabr_data_type)(PDBR)(s)) = d''
{-2}  ∀ (s: (pm' states)):
      OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s)) ∧
      d' = data(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-3}  ∀ (cs: Segment_Reg_type, pabr: Pabr_type):
      ∀ (x1: Physical_memory):
        every[Physical_memory, [bool, Address]]
          (λ (x: Physical_memory): TRUE,
           λ (t: [bool, Address]):
             Mem?(t'2'type_of) ∧
             (0 ≤ t'2'offset ∧ t'2'offset < max_linear_offset) ∧
             IF t'1
               THEN aligned?(22)(offset(t'2))
               ELSE aligned?(12)(offset(t'2))
             ENDIF)
          (translate[Physical_memory, pm_phy]
           (0, pabr'base_addr, addr', access', segment_to_priv(cs))(x1))
{-4}  ∀ (cs: Segment_Reg_type, pabr: Pabr_type, leaf: bool,
      base:
        {b: Memory_Address_4G |
          IF leaf
            THEN aligned?(-1 × (pdir_lvl [Physical_memory, pm_phy] × bits_per_level
              (offset(b))
            ELSE aligned?(bits_per_level[Physical_memory, pm_phy] + pe_size)
              (offset(b))
            ENDIF}):
      ¬ leaf ⊃
      (∀ (x1: Physical_memory):
        every[Physical_memory, [bool, Address]]
          (λ (x: Physical_memory): TRUE,
           λ (t: [bool, Address]):
             Mem?(t'2'type_of) ∧ 0 ≤ t'2'offset ∧ t'2'offset < max_linear_offset)
          (translate[Physical_memory, pm_phy]
           (1, base, addr', access', segment_to_priv(cs))(x1)))
{-5}  pm' mem = linear_pm
{-6}  ∀ (s: (pm' states)): OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-7}  ∀ (s1, s2: (pm' states)):
      data(read_data(pm_phy, segment_reg_data_type)(CS)(s1)) =
      data(read_data(pm_phy, segment_reg_data_type)(CS)(s2))
{-8}  ∀ (s: (pm' states)): OK?(read_data(pm_phy, pabr_data_type)(PDBR)(s))
{-9}  ∀ (s1, s2: (pm' states)):
      data(read_data(pm_phy, pabr_data_type)(PDBR)(s1)) =
      data(read_data(pm_phy, pabr_data_type)(PDBR)(s2))
{-10} ∀ (s1, s2: (pm' states)):
      pe_in_pdir_range?(s1,
        restrict[Address, Memory_Address_4G, boolean]
          ((pm'ro_addr ∪ pm'rw_addr)))
      =
      pe_in_pdir_range?(s2,
        restrict[Address, Memory_Address_4G, boolean]
          ((pm'ro_addr ∪ pm'rw_addr)))
      ∧
      pe_in_ptab_range?(s1,
        restrict[Address, Memory_Address_4G, boolean]
          ((pm'ro_addr ∪ pm'rw_addr)))
      =
      pe_in_ptab_range?(s2,
        restrict[Address, Memory_Address_4G, boolean]
          ((pm'ro_addr ∪ pm'rw_addr)))
{-11} ∀ (s: (pm' states), lvl: Level,
      a:

```

C.117 Proofs for Linear_Memory_Properties (challenge-linear.pvs)

Using lemma `subset_ptab_address_rw_addr`,

we get 2 subgoals:

linear_resolve_unchanged_pm_phy.1.1.3.3.2.4.1:

```

{-1} (address_in_pt_range?(s', (restrict [Address, Memory_Address_4G, bool](pm'ro_addr) ∪ restrict
    ^
    (address_block(xlat_idx(0, d''base_addr, addr'), expt(2, pe_size))) ⊆ extend [Address, Memory_Address_4G, bool]
    ⊃
    (address_block(xlat_idx(0, d''base_addr, addr'), expt(2, pe_size))) ⊆ pm_phy'rw_addr)
{-2} ∀ (s: (pm_phy'states)):
    OK?(read_data(pm_phy, pdb_data_type)(PDBR)(s)) ∧
    data(read_data(pm_phy, pdb_data_type)(PDBR)(s)) = d''
{-3} ∀ (s: (pm'states)):
    OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s)) ∧
    d' = data(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-4} ∀ (cs: Segment_Reg_type, pdb: Pdb_type):
    ∀ (x1: Physical_memory):
        every [Physical_memory, [bool, Address]]
            (λ (x: Physical_memory): TRUE,
             λ (t: [bool, Address]):
                 Mem?(t'2'type_of) ∧
                 (0 ≤ t'2'offset ∧ t'2'offset < max_linear_offset) ∧
                 IF t'1
                     THEN aligned?(22)(offset(t'2))
                     ELSE aligned?(12)(offset(t'2))
                 ENDIF)
            (translate [Physical_memory, pm_phy]
                (0, pdb'base_addr, addr', access', segment_to_priv(cs))(x1))
{-5} ∀ (cs: Segment_Reg_type, pdb: Pdb_type, leaf: bool,
    base:
        {b: Memory_Address_4G |
            IF leaf
                THEN aligned?(-1 × (pdir_lvl [Physical_memory, pm_phy] × bits_per_level
                    (offset(b)))
                ELSE aligned?(bits_per_level [Physical_memory, pm_phy] + pe_size)
                    (offset(b))
            ENDIF}):
    ¬ leaf ⊃
    (∀ (x1: Physical_memory):
        every [Physical_memory, [bool, Address]]
            (λ (x: Physical_memory): TRUE,
             λ (t: [bool, Address]):
                 Mem?(t'2'type_of) ∧ 0 ≤ t'2'offset ∧ t'2'offset < max_linear_offset)
            (translate [Physical_memory, pm_phy]
                (1, base, addr', access', segment_to_priv(cs))(x1)))
{-6} pm'mem = linear_pm
{-7} ∀ (s: (pm'states)): OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-8} ∀ (s1, s2: (pm'states)):
    data(read_data(pm_phy, segment_reg_data_type)(CS)(s1)) =
    data(read_data(pm_phy, segment_reg_data_type)(CS)(s2))
{-9} ∀ (s: (pm'states)): OK?(read_data(pm_phy, pdb_data_type)(PDBR)(s))
{-10} ∀ (s1, s2: (pm'states)):
    data(read_data(pm_phy, pdb_data_type)(PDBR)(s1)) =
    data(read_data(pm_phy, pdb_data_type)(PDBR)(s2))
{-11} ∀ (s1, s2: (pm'states)):
    pe_in_pdir_range?(s1,
        restrict [Address, Memory_Address_4G, boolean]
            ((pm'ro_addr ∪ pm'rw_addr)))
    =
    pe_in_pdir_range?(s2,
        restrict [Address, Memory_Address_4G, boolean]
            ((pm'ro_addr ∪ pm'rw_addr)))
    ∧
    pe_in_ptab_range?(s1,
        restrict [Address, Memory_Address_4G, boolean]
            ((pm'ro_addr ∪ pm'rw_addr)))
    =

```

C.117 Proofs for Linear_Memory_Properties (challenge-linear.pvs)

Simplifying, rewriting, and recording with decision procedures,

Hiding formulas: (2 3 4),

Applying propositional simplification,

we get 2 subgoals:

linear_resolve_unchanged_pm_phy.1.1.3.3.2.4.1.1:

```

{-1}  ∀ (s: (pm_phy' states)):
      OK?(read_data(pm_phy, pabr_data_type)(PDBR)(s)) ∧
      data(read_data(pm_phy, pabr_data_type)(PDBR)(s)) = d''
{-2}  ∀ (s: (pm' states)):
      OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s)) ∧
      d' = data(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-3}  ∀ (cs: Segment_Reg_type, pabr: Pabr_type):
      ∀ (x1: Physical_memory):
        every[Physical_memory, [bool, Address]]
          (λ (x: Physical_memory): TRUE,
           λ (t: [bool, Address]):
             Mem?(t'2'type_of) ∧
             (0 ≤ t'2'offset ∧ t'2'offset < max_linear_offset) ∧
             IF t'1
               THEN aligned?(22)(offset(t'2))
               ELSE aligned?(12)(offset(t'2))
             ENDIF)
          (translate[Physical_memory, pm_phy]
           (0, pabr'base_addr, addr', access', segment_to_priv(cs))(x1))
{-4}  ∀ (cs: Segment_Reg_type, pabr: Pabr_type, leaf: bool,
      base:
        {b: Memory_Address_4G |
          IF leaf
            THEN aligned?(-1 × (pdir_lvl [Physical_memory, pm_phy] × bits_per_level
              (offset(b))
            ELSE aligned?(bits_per_level[Physical_memory, pm_phy] + pe_size)
              (offset(b))
            ENDIF}):
      ¬ leaf ⊃
      (∀ (x1: Physical_memory):
        every[Physical_memory, [bool, Address]]
          (λ (x: Physical_memory): TRUE,
           λ (t: [bool, Address]):
             Mem?(t'2'type_of) ∧ 0 ≤ t'2'offset ∧ t'2'offset < max_linear_offset)
          (translate[Physical_memory, pm_phy]
           (1, base, addr', access', segment_to_priv(cs))(x1)))
{-5}  pm' mem = linear_pm
{-6}  ∀ (s: (pm' states)): OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-7}  ∀ (s1, s2: (pm' states)):
      data(read_data(pm_phy, segment_reg_data_type)(CS)(s1)) =
      data(read_data(pm_phy, segment_reg_data_type)(CS)(s2))
{-8}  ∀ (s: (pm' states)): OK?(read_data(pm_phy, pabr_data_type)(PDBR)(s))
{-9}  ∀ (s1, s2: (pm' states)):
      data(read_data(pm_phy, pabr_data_type)(PDBR)(s1)) =
      data(read_data(pm_phy, pabr_data_type)(PDBR)(s2))
{-10} ∀ (s1, s2: (pm' states)):
      pe_in_pdir_range?(s1,
        restrict[Address, Memory_Address_4G, boolean]
          ((pm'ro_addr ∪ pm'rw_addr)))
      =
      pe_in_pdir_range?(s2,
        restrict[Address, Memory_Address_4G, boolean]
          ((pm'ro_addr ∪ pm'rw_addr)))
      ∧
      pe_in_ptab_range?(s1,
        restrict[Address, Memory_Address_4G, boolean]
          ((pm'ro_addr ∪ pm'rw_addr)))
      =
      pe_in_ptab_range?(s2,
        restrict[Address, Memory_Address_4G, boolean]
          ((pm'ro_addr ∪ pm'rw_addr)))
{-11} ∀ (s: (pm' states), lvl: Level,
      a:

```

Using lemma `pm_address_in_pt_range_in_rw_addr`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Case splitting on `restrict[Address, Memory_Address_4G, boolean] (union(pm!1'ro_addr, pm!1'rw_addr)) = union(restrict[Address, Memory_Address_4G, boolean](pm!1'ro_addr), restrict[Address, Memory_Address_4G, boolean](pm!1'rw_addr))`,

we get 2 subgoals:

linear_resolve_unchanged_pm_phy.1.1.3.3.2.4.1.1.1:

```

{-1} restrict[Address, Memory_Address_4G, boolean]((pm'ro_addr ∪ pm'rw_addr)) =
      (restrict[Address, Memory_Address_4G, boolean](pm'ro_addr) ∪ restrict [Address, Memory_Address_4G, boolean](pm'rw_addr))
{-2} is_linear_plain_memory?(pm')
{-3} (address_in_pt_range?(s', restrict [Address, Memory_Address_4G, boolean]((pm'ro_addr ∪ pm'rw_addr)))
{-4} ∀ (s: (pm_phy'states)):
      OK?(read_data(pm_phy, pabr_data_type)(PDBR)(s)) ∧
      data(read_data(pm_phy, pabr_data_type)(PDBR)(s)) = d''
{-5} ∀ (s: (pm'states)):
      OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s)) ∧
      d' = data(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-6} ∀ (cs: Segment_Reg_type, pabr: Pabr_type):
      ∀ (x1: Physical_memory):
        every[Physical_memory, [bool, Address]]
          (λ (x: Physical_memory): TRUE,
           λ (t: [bool, Address]):
             Mem?(t'2'type_of) ∧
             (0 ≤ t'2'offset ∧ t'2'offset < max_linear_offset) ∧
             IF t'1
               THEN aligned?(22)(offset(t'2))
               ELSE aligned?(12)(offset(t'2))
             ENDIF)
          (translate[Physical_memory, pm_phy]
           (0, pabr'base_addr, addr', access', segment_to_priv(cs))(x1))
{-7} ∀ (cs: Segment_Reg_type, pabr: Pabr_type, leaf: bool,
      base:
        {b: Memory_Address_4G |
          IF leaf
            THEN aligned?(-1 × (pdir_lvl [Physical_memory, pm_phy] × bits_per_level
              (offset(b))
            ELSE aligned?(bits_per_level[Physical_memory, pm_phy] + pe_size)
              (offset(b))
            ENDIF}):
      ¬ leaf ⊃
      (∀ (x1: Physical_memory):
        every[Physical_memory, [bool, Address]]
          (λ (x: Physical_memory): TRUE,
           λ (t: [bool, Address]):
             Mem?(t'2'type_of) ∧ 0 ≤ t'2'offset ∧ t'2'offset < max_linear_offset)
          (translate[Physical_memory, pm_phy]
           (1, base, addr', access', segment_to_priv(cs))(x1)))
{-8} pm'mem = linear_pm
{-9} ∀ (s: (pm'states)): OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-10} ∀ (s1, s2: (pm'states)):
      data(read_data(pm_phy, segment_reg_data_type)(CS)(s1)) =
      data(read_data(pm_phy, segment_reg_data_type)(CS)(s2))
{-11} ∀ (s: (pm'states)): OK?(read_data(pm_phy, pabr_data_type)(PDBR)(s))
{-12} ∀ (s1, s2: (pm'states)):
      data(read_data(pm_phy, pabr_data_type)(PDBR)(s1)) =
      data(read_data(pm_phy, pabr_data_type)(PDBR)(s2))
{-13} ∀ (s1, s2: (pm'states)):
      pe_in_pdir_range?(s1,
        restrict[Address, Memory_Address_4G, boolean]
          ((pm'ro_addr ∪ pm'rw_addr)))
      =
      pe_in_pdir_range?(s2,
        restrict[Address, Memory_Address_4G, boolean]
          ((pm'ro_addr ∪ pm'rw_addr)))
      ∧
      pe_in_ptab_range?(s1,
        restrict[Address, Memory_Address_4G, boolean]
          ((pm'ro_addr ∪ pm'rw_addr)))
      =
      pe_in_ptab_range?(s2,

```

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `linear_resolve_unchanged_pm_phy.1.1.3.3.2.4.1.1.1.`

linear_resolve_unchanged_pm_phy.1.1.3.3.2.4.1.1.2:

```

{-1}  is_linear_plain_memory?(pm')
{-2}  (address_in_pt_range?(s', restrict [Address, Memory_Address_4G, boolean])((pm'ro_addr ∪ pm'rw_addr)))
{-3}  ∀ (s: (pm_phy' states)):
      OK?(read_data(pm_phy, pabr_data_type)(PDBR)(s)) ∧
      data(read_data(pm_phy, pabr_data_type)(PDBR)(s)) = d''
{-4}  ∀ (s: (pm' states)):
      OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s)) ∧
      d' = data(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-5}  ∀ (cs: Segment_Reg_type, pabr: Pabr_type):
      ∀ (x1: Physical_memory):
        every[Physical_memory, [bool, Address]]
          (λ (x: Physical_memory): TRUE,
           λ (t: [bool, Address]):
             Mem?(t'2'type_of) ∧
             (0 ≤ t'2'offset ∧ t'2'offset < max_linear_offset) ∧
             IF t'1
               THEN aligned?(22)(offset(t'2))
               ELSE aligned?(12)(offset(t'2))
             ENDIF)
          (translate[Physical_memory, pm_phy]
           (0, pabr'base_addr, addr', access', segment_to_priv(cs))(x1))
{-6}  ∀ (cs: Segment_Reg_type, pabr: Pabr_type, leaf: bool,
      base:
        {b: Memory_Address_4G |
          IF leaf
            THEN aligned?(-1 × (pdir_lvl [Physical_memory, pm_phy] × bits_per_level
              (offset(b)))
            ELSE aligned?(bits_per_level[Physical_memory, pm_phy] + pe_size)
              (offset(b))
            ENDIF}):
      ¬ leaf ⊃
      (∀ (x1: Physical_memory):
        every[Physical_memory, [bool, Address]]
          (λ (x: Physical_memory): TRUE,
           λ (t: [bool, Address]):
             Mem?(t'2'type_of) ∧ 0 ≤ t'2'offset ∧ t'2'offset < max_linear_offset)
          (translate[Physical_memory, pm_phy]
           (1, base, addr', access', segment_to_priv(cs))(x1)))
{-7}  pm'mem = linear_pm
{-8}  ∀ (s: (pm' states)): OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-9}  ∀ (s1, s2: (pm' states)):
      data(read_data(pm_phy, segment_reg_data_type)(CS)(s1)) =
      data(read_data(pm_phy, segment_reg_data_type)(CS)(s2))
{-10} ∀ (s: (pm' states)): OK?(read_data(pm_phy, pabr_data_type)(PDBR)(s))
{-11} ∀ (s1, s2: (pm' states)):
      data(read_data(pm_phy, pabr_data_type)(PDBR)(s1)) =
      data(read_data(pm_phy, pabr_data_type)(PDBR)(s2))
{-12} ∀ (s1, s2: (pm' states)):
      pe_in_pdir_range?(s1,
        restrict[Address, Memory_Address_4G, boolean]
          ((pm'ro_addr ∪ pm'rw_addr)))
      =
      pe_in_pdir_range?(s2,
        restrict[Address, Memory_Address_4G, boolean]
          ((pm'ro_addr ∪ pm'rw_addr)))
      ∧
      pe_in_ptab_range?(s1,
        restrict[Address, Memory_Address_4G, boolean]
          ((pm'ro_addr ∪ pm'rw_addr)))
      =
      pe_in_ptab_range?(s2,
        restrict[Address, Memory_Address_4G, boolean]
          ((pm'ro_addr ∪ pm'rw_addr)))
{-13} ∀ (s: (pm' states)):
      pe_in_pdir_range?(s,
        restrict[Address, Memory_Address_4G, boolean]
          ((pm'ro_addr ∪ pm'rw_addr)))
      =
      pe_in_ptab_range?(s,
        restrict[Address, Memory_Address_4G, boolean]
          ((pm'ro_addr ∪ pm'rw_addr)))

```

Keeping (1) and hiding *,

Applying decompose-equality,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `linear_resolve_unchanged_pm_phy.1.1.3.3.2.4.1.1.2.`

linear_resolve_unchanged_pm_phy.1.1.3.3.2.4.1.2:

```

{-1}  ∀ (s: (pm_phy' states)):
      OK?(read_data(pm_phy, pabr_data_type)(PDBR)(s)) ∧
      data(read_data(pm_phy, pabr_data_type)(PDBR)(s)) = d''
{-2}  ∀ (s: (pm' states)):
      OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s)) ∧
      d' = data(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-3}  ∀ (cs: Segment_Reg_type, pabr: Pabr_type):
      ∀ (x1: Physical_memory):
        every[Physical_memory, [bool, Address]]
          (λ (x: Physical_memory): TRUE,
           λ (t: [bool, Address]):
             Mem?(t'2'type_of) ∧
             (0 ≤ t'2'offset ∧ t'2'offset < max_linear_offset) ∧
             IF t'1
               THEN aligned?(22)(offset(t'2))
               ELSE aligned?(12)(offset(t'2))
             ENDIF)
          (translate[Physical_memory, pm_phy]
           (0, pabr'base_addr, addr', access', segment_to_priv(cs))(x1))
{-4}  ∀ (cs: Segment_Reg_type, pabr: Pabr_type, leaf: bool,
      base:
        {b: Memory_Address_4G |
          IF leaf
            THEN aligned?(-1 × (pdir_lvl [Physical_memory, pm_phy] × bits_per_level
              (offset(b))
            ELSE aligned?(bits_per_level[Physical_memory, pm_phy] + pe_size)
              (offset(b))
            ENDIF}):
      ¬ leaf ⊃
      (∀ (x1: Physical_memory):
        every[Physical_memory, [bool, Address]]
          (λ (x: Physical_memory): TRUE,
           λ (t: [bool, Address]):
             Mem?(t'2'type_of) ∧ 0 ≤ t'2'offset ∧ t'2'offset < max_linear_offset)
          (translate[Physical_memory, pm_phy]
           (1, base, addr', access', segment_to_priv(cs))(x1)))
{-5}  pm' mem = linear_pm
{-6}  ∀ (s: (pm' states)): OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-7}  ∀ (s1, s2: (pm' states)):
      data(read_data(pm_phy, segment_reg_data_type)(CS)(s1)) =
      data(read_data(pm_phy, segment_reg_data_type)(CS)(s2))
{-8}  ∀ (s: (pm' states)): OK?(read_data(pm_phy, pabr_data_type)(PDBR)(s))
{-9}  ∀ (s1, s2: (pm' states)):
      data(read_data(pm_phy, pabr_data_type)(PDBR)(s1)) =
      data(read_data(pm_phy, pabr_data_type)(PDBR)(s2))
{-10} ∀ (s1, s2: (pm' states)):
      pe_in_pdir_range?(s1,
        restrict[Address, Memory_Address_4G, boolean]
          ((pm'ro_addr ∪ pm'rw_addr)))
      =
      pe_in_pdir_range?(s2,
        restrict[Address, Memory_Address_4G, boolean]
          ((pm'ro_addr ∪ pm'rw_addr)))
      ∧
      pe_in_ptab_range?(s1,
        restrict[Address, Memory_Address_4G, boolean]
          ((pm'ro_addr ∪ pm'rw_addr)))
      =
      pe_in_ptab_range?(s2,
        restrict[Address, Memory_Address_4G, boolean]
          ((pm'ro_addr ∪ pm'rw_addr)))
{-11} ∀ (s: (pm' states), lvl: Level,
      a:

```

C.117 Proofs for Linear_Memory_Properties (challenge-linear.pvs)

Using lemma `pe_in_pt_address_in_pt`,

we get 2 subgoals:

linear_resolve_unchanged_pm_phy.1.1.3.3.2.4.1.2.1:

```

{-1} pe_in_pt_range?(s',
      restrict[Address, Memory_Address_4G, bool]
        ((pm'ro_addr ∪ pm'rw_addr)),
      0)
      (xlat_idx(0, d''base_addr, addr'))
  ⊃
  (address_block(xlat_idx(0, d''base_addr, addr'), expt(2, pe_size)) ⊆ extend [Address, Memory_Address_4G, bool]
    (address_block(xlat_idx(0, d''base_addr, addr'), expt(2, pe_size)))
  {-2} ∇ (s: (pm_phy'states)):
      OK?(read_data(pm_phy, pdir_data_type)(PDIR)(s)) ∧
      data(read_data(pm_phy, pdir_data_type)(PDIR)(s)) = d''
  {-3} ∇ (s: (pm'states)):
      OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s)) ∧
      d' = data(read_data(pm_phy, segment_reg_data_type)(CS)(s))
  {-4} ∇ (cs: Segment_Reg_type, pdir: Pdir_type):
      ∇ (x₁: Physical_memory):
        every[Physical_memory, [bool, Address]]
          (λ (x: Physical_memory): TRUE,
           λ (t: [bool, Address]):
             Mem?(t'2'type_of) ∧
             (0 ≤ t'2'offset ∧ t'2'offset < max_linear_offset) ∧
             IF t'1
               THEN aligned?(22)(offset(t'2))
               ELSE aligned?(12)(offset(t'2))
             ENDIF)
          (translate[Physical_memory, pm_phy]
            (0, pdir'base_addr, addr', access', segment_to_priv(cs))(x₁))
  {-5} ∇ (cs: Segment_Reg_type, pdir: Pdir_type, leaf: bool,
        base:
          {b: Memory_Address_4G |
            IF leaf
              THEN aligned?(-1 × (pdir_lvl [Physical_memory, pm_phy] × bits_per_level)
                (offset(b))
            ELSE aligned?(bits_per_level[Physical_memory, pm_phy] + pe_size)
                (offset(b))
            ENDIF}):
      ¬ leaf ⊃
      (∇ (x₁: Physical_memory):
        every[Physical_memory, [bool, Address]]
          (λ (x: Physical_memory): TRUE,
           λ (t: [bool, Address]):
             Mem?(t'2'type_of) ∧ 0 ≤ t'2'offset ∧ t'2'offset < max_linear_offset)
          (translate[Physical_memory, pm_phy]
            (1, base, addr', access', segment_to_priv(cs))(x₁)))
  {-6} pm'mem = linear_pm
  {-7} ∇ (s: (pm'states)): OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s))
  {-8} ∇ (s₁, s₂: (pm'states)):
      data(read_data(pm_phy, segment_reg_data_type)(CS)(s₁)) =
      data(read_data(pm_phy, segment_reg_data_type)(CS)(s₂))
  {-9} ∇ (s: (pm'states)): OK?(read_data(pm_phy, pdir_data_type)(PDIR)(s))
  {-10} ∇ (s₁, s₂: (pm'states)):
      data(read_data(pm_phy, pdir_data_type)(PDIR)(s₁)) =
      data(read_data(pm_phy, pdir_data_type)(PDIR)(s₂))
  {-11} ∇ (s₁, s₂: (pm'states)):
      pe_in_pdir_range?(s₁,
        restrict[Address, Memory_Address_4G, boolean]
          ((pm'ro_addr ∪ pm'rw_addr)))
      =
      pe_in_pdir_range?(s₂,
        restrict[Address, Memory_Address_4G, boolean]
          ((pm'ro_addr ∪ pm'rw_addr)))
      ∧
      pe_in_ptab_range?(s₁,
        restrict[Address, Memory_Address_4G, boolean]
          ((pm'ro_addr ∪ pm'rw_addr)))

```

C.117 Proofs for Linear_Memory_Properties (challenge-linear.pvs)

Using lemma `xlat_pe_in_pdir_range`,

Instantiating quantified variables,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Expanding the definition of `pe_in_pt_range?`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `linear_resolve_unchanged_pm_phy.1.1.3.3.2.4.1.2.1`.

linear_resolve_unchanged_pm_phy.1.1.3.3.2.4.1.2.2:

```

{-1}  ∀ (s: (pm_phy' states)):
      OK?(read_data(pm_phy, pdir_data_type)(PDBR)(s)) ∧
      data(read_data(pm_phy, pdir_data_type)(PDBR)(s)) = d''
{-2}  ∀ (s: (pm' states)):
      OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s)) ∧
      d' = data(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-3}  ∀ (cs: Segment_Reg_type, pdir: Pdir_type):
      ∀ (x1: Physical_memory):
        every[Physical_memory, [bool, Address]]
          (λ (x: Physical_memory): TRUE,
           λ (t: [bool, Address]):
             Mem?(t'2'type_of) ∧
             (0 ≤ t'2'offset ∧ t'2'offset < max_linear_offset) ∧
             IF t'1
               THEN aligned?(22)(offset(t'2))
               ELSE aligned?(12)(offset(t'2))
             ENDIF)
          (translate[Physical_memory, pm_phy]
           (0, pdir'base_addr, addr', access', segment_to_priv(cs))(x1))
{-4}  ∀ (cs: Segment_Reg_type, pdir: Pdir_type, leaf: bool,
      base:
        {b: Memory_Address_4G |
          IF leaf
            THEN aligned?(-1 × (pdir_lvl [Physical_memory, pm_phy] × bits_per_level
              (offset(b))
            ELSE aligned?(bits_per_level[Physical_memory, pm_phy] + pe_size)
              (offset(b))
            ENDIF}):
      ¬ leaf ⊃
      (∀ (x1: Physical_memory):
        every[Physical_memory, [bool, Address]]
          (λ (x: Physical_memory): TRUE,
           λ (t: [bool, Address]):
             Mem?(t'2'type_of) ∧ 0 ≤ t'2'offset ∧ t'2'offset < max_linear_offset)
          (translate[Physical_memory, pm_phy]
           (1, base, addr', access', segment_to_priv(cs))(x1)))
{-5}  pm' mem = linear_pm
{-6}  ∀ (s: (pm' states)): OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-7}  ∀ (s1, s2: (pm' states)):
      data(read_data(pm_phy, segment_reg_data_type)(CS)(s1)) =
      data(read_data(pm_phy, segment_reg_data_type)(CS)(s2))
{-8}  ∀ (s: (pm' states)): OK?(read_data(pm_phy, pdir_data_type)(PDBR)(s))
{-9}  ∀ (s1, s2: (pm' states)):
      data(read_data(pm_phy, pdir_data_type)(PDBR)(s1)) =
      data(read_data(pm_phy, pdir_data_type)(PDBR)(s2))
{-10} ∀ (s1, s2: (pm' states)):
      pe_in_pdir_range?(s1,
        restrict[Address, Memory_Address_4G, boolean]
          ((pm'ro_addr ∪ pm'rw_addr)))
      =
      pe_in_pdir_range?(s2,
        restrict[Address, Memory_Address_4G, boolean]
          ((pm'ro_addr ∪ pm'rw_addr)))
      ∧
      pe_in_ptab_range?(s1,
        restrict[Address, Memory_Address_4G, boolean]
          ((pm'ro_addr ∪ pm'rw_addr)))
      =
      pe_in_ptab_range?(s2,
        restrict[Address, Memory_Address_4G, boolean]
          ((pm'ro_addr ∪ pm'rw_addr)))
{-11} ∀ (s: (pm' states), lvl: Level,
      a:

```

C.117 Proofs for Linear_Memory_Properties (challenge-linear.pvs)

Using lemma `xlat_idx_memory_address`,

Using lemma `xlat_idx_memory_address2`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `linear_resolve_unchanged_pm_phy.1.1.3.3.2.4.1.2.2`.

linear_resolve_unchanged_pm_phy.1.1.3.3.2.4.2:

```

{-1}  ∀ (s: (pm_phy' states)):
      OK?(read_data(pm_phy, pabr_data_type)(PDBR)(s)) ∧
      data(read_data(pm_phy, pabr_data_type)(PDBR)(s)) = d''
{-2}  ∀ (s: (pm' states)):
      OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s)) ∧
      d' = data(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-3}  ∀ (cs: Segment_Reg_type, pabr: Pabr_type):
      ∀ (x1: Physical_memory):
        every[Physical_memory, [bool, Address]]
          (λ (x: Physical_memory): TRUE,
           λ (t: [bool, Address]):
             Mem?(t'2'type_of) ∧
             (0 ≤ t'2'offset ∧ t'2'offset < max_linear_offset) ∧
             IF t'1
               THEN aligned?(22)(offset(t'2))
               ELSE aligned?(12)(offset(t'2))
             ENDIF)
          (translate[Physical_memory, pm_phy]
           (0, pabr'base_addr, addr', access', segment_to_priv(cs))(x1))
{-4}  ∀ (cs: Segment_Reg_type, pabr: Pabr_type, leaf: bool,
      base:
        {b: Memory_Address_4G |
          IF leaf
            THEN aligned?(-1 × (pdir_lvl [Physical_memory, pm_phy] × bits_per_level
              (offset(b))
            ELSE aligned?(bits_per_level[Physical_memory, pm_phy] + pe_size)
              (offset(b))
            ENDIF}):
      ¬ leaf ⊃
      (∀ (x1: Physical_memory):
        every[Physical_memory, [bool, Address]]
          (λ (x: Physical_memory): TRUE,
           λ (t: [bool, Address]):
             Mem?(t'2'type_of) ∧ 0 ≤ t'2'offset ∧ t'2'offset < max_linear_offset)
          (translate[Physical_memory, pm_phy]
           (1, base, addr', access', segment_to_priv(cs))(x1)))
{-5}  pm' mem = linear_pm
{-6}  ∀ (s: (pm' states)): OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-7}  ∀ (s1, s2: (pm' states)):
      data(read_data(pm_phy, segment_reg_data_type)(CS)(s1)) =
      data(read_data(pm_phy, segment_reg_data_type)(CS)(s2))
{-8}  ∀ (s: (pm' states)): OK?(read_data(pm_phy, pabr_data_type)(PDBR)(s))
{-9}  ∀ (s1, s2: (pm' states)):
      data(read_data(pm_phy, pabr_data_type)(PDBR)(s1)) =
      data(read_data(pm_phy, pabr_data_type)(PDBR)(s2))
{-10} ∀ (s1, s2: (pm' states)):
      pe_in_pdir_range?(s1,
        restrict[Address, Memory_Address_4G, boolean]
          ((pm'ro_addr ∪ pm'rw_addr)))
      =
      pe_in_pdir_range?(s2,
        restrict[Address, Memory_Address_4G, boolean]
          ((pm'ro_addr ∪ pm'rw_addr)))
      ∧
      pe_in_ptab_range?(s1,
        restrict[Address, Memory_Address_4G, boolean]
          ((pm'ro_addr ∪ pm'rw_addr)))
      =
      pe_in_ptab_range?(s2,
        restrict[Address, Memory_Address_4G, boolean]
          ((pm'ro_addr ∪ pm'rw_addr)))
{-11} ∀ (s: (pm' states), lvl: Level,
      a:

```

C.117 Proofs for Linear_Memory_Properties (challenge-linear.pvs)

Using lemma `xlat_idx_memory_address`,

Using lemma `xlat_idx_memory_address2`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `linear_resolve_unchanged_pm_phy.1.1.3.3.2.4.2`.

linear_resolve_unchanged_pm_phy.1.1.3.3.2.5:

```

{-1}  unchanged_memory_invariant?(pm_phy' mem, pm_phy' states,
                                         singleton(expr_2_super(translate(0,
                                                                                   d''' base_addr,
                                                                                   addr',
                                                                                   access',
                                                                                   seg-
                                                                                   ment_to_priv(d')))),
                                         ((pm_phy' ro_addr ∪ pm_phy' rw_addr) \ address_block(xlat_idx(
{-2}  ∀ (s: (pm_phy' states)):
      OK?(read_data(pm_phy, pdbr_data_type)(PDBR)(s)) ∧
      data(read_data(pm_phy, pdbr_data_type)(PDBR)(s)) = d''
{-3}  ∀ (s: (pm' states)):
      OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s)) ∧
      d' = data(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-4}  ∀ (cs: Segment_Reg_type, pdbr: Pdbr_type):
      ∀ (x1: Physical_memory):
        every[Physical_memory, [bool, Address]]
          (λ (x: Physical_memory): TRUE,
           λ (t: [bool, Address]):
             Mem?(t'2'type_of) ∧
             (0 ≤ t'2'offset ∧ t'2'offset < max_linear_offset) ∧
             IF t'1
               THEN aligned?(22)(offset(t'2))
               ELSE aligned?(12)(offset(t'2))
             ENDIF)
          (translate[Physical_memory, pm_phy]
            (0, pdbr' base_addr, addr', access', segment_to_priv(cs))(x1))
{-5}  ∀ (cs: Segment_Reg_type, pdbr: Pdbr_type, leaf: bool,
      base:
        {b: Memory_Address_4G |
          IF leaf
            THEN aligned?(-1 × (pdir_lvl [Physical_memory, pm_phy] × bits_per_level
              (offset(b))
            ELSE aligned?(bits_per_level[Physical_memory, pm_phy] + pe_size)
              (offset(b))
            ENDIF}):
      ¬ leaf ⊃
      (∀ (x1: Physical_memory):
        every[Physical_memory, [bool, Address]]
          (λ (x: Physical_memory): TRUE,
           λ (t: [bool, Address]):
             Mem?(t'2'type_of) ∧ 0 ≤ t'2'offset ∧ t'2'offset < max_linear_offset)
          (translate[Physical_memory, pm_phy]
            (1, base, addr', access', segment_to_priv(cs))(x1)))
{-6}  pm' mem = linear_pm
{-7}  ∀ (s: (pm' states)): OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-8}  ∀ (s1, s2: (pm' states)):
      data(read_data(pm_phy, segment_reg_data_type)(CS)(s1)) =
      data(read_data(pm_phy, segment_reg_data_type)(CS)(s2))
{-9}  ∀ (s: (pm' states)): OK?(read_data(pm_phy, pdbr_data_type)(PDBR)(s))
{-10} ∀ (s1, s2: (pm' states)):
      data(read_data(pm_phy, pdbr_data_type)(PDBR)(s1)) =
      data(read_data(pm_phy, pdbr_data_type)(PDBR)(s2))
{-11} ∀ (s1, s2: (pm' states)):
      pe_in_pdir_range?(s1,
        restrict[Address, Memory_Address_4G, boolean]
          ((pm' ro_addr ∪ pm' rw_addr)))
      =
      pe_in_pdir_range?(s2,
        restrict[Address, Memory_Address_4G, boolean]
          ((pm' ro_addr ∪ pm' rw_addr)))
      ∧
      pe_in_ptab_range?(s1,
        restrict[Address, Memory_Address_4G, boolean]
          ((pm' ro_addr ∪ pm' rw_addr)))

```

Replacing using formula -21,

Rewriting using subset_of_differences, matching in *,

Using lemma pe_in_pt_address_in_pt,

we get 2 subgoals:

linear_resolve_unchanged_pm_phy.1.1.3.3.2.5.1:

```

{-1} pe_in_pt_range?(s',
      restrict[Address, Memory_Address_4G, boolean]
        ((pm'ro_addr ∪ pm'rw_addr)),
      0)
      (xlat_idx(0, d''base_addr, addr'))
    ⊃
      (address_block(xlat_idx(0, d''base_addr, addr'), expt(2, pe_size)) ⊆ extend [Address, Memory_Address_4G, boolean]
        (singleton(expr_2_super(translate(0,
                                                    d''base_addr,
                                                    addr',
                                                    access',
                                                    segment_to_priv(d'))),
                                                    ((pm_phy'ro_addr ∪ pm_phy'rw_addr) \ address_block(xlat_idx(0, d''base_addr, addr'), expt(2, pe_size))))))
{-2} unchanged_memory_invariant?(pm_phy'mem, pm_phy'states,
      singleton(expr_2_super(translate(0,
                                                    d''base_addr,
                                                    addr',
                                                    access',
                                                    segment_to_priv(d'))),
                                                    ((pm_phy'ro_addr ∪ pm_phy'rw_addr) \ address_block(xlat_idx(0, d''base_addr, addr'), expt(2, pe_size))))))
{-3} ∀ (s: (pm_phy'states)):
      OK?(read_data(pm_phy, pdir_data_type)(PDIR)(s)) ∧
      data(read_data(pm_phy, pdir_data_type)(PDIR)(s)) = d''
{-4} ∀ (s: (pm'states)):
      OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s)) ∧
      d' = data(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-5} ∀ (cs: Segment_Reg_type, pdir: Pdir_type):
      ∀ (x1: Physical_memory):
        every[Physical_memory, [bool, Address]]
          (λ (x: Physical_memory): TRUE,
           λ (t: [bool, Address]):
             Mem?(t'2'type_of) ∧
             (0 ≤ t'2'offset ∧ t'2'offset < max_linear_offset) ∧
             IF t'1
               THEN aligned?(22)(offset(t'2))
               ELSE aligned?(12)(offset(t'2))
             ENDIF)
          (translate[Physical_memory, pm_phy]
            (0, pdir'base_addr, addr', access', segment_to_priv(cs))(x1))
{-6} ∀ (cs: Segment_Reg_type, pdir: Pdir_type, leaf: bool,
      base:
        {b: Memory_Address_4G |
          IF leaf
            THEN aligned?(-1 × (pdir_lvl [Physical_memory, pm_phy] × bits_per_level)
              (offset(b))
            ELSE aligned?(bits_per_level[Physical_memory, pm_phy] + pe_size)
              (offset(b))
            ENDIF}):
      ¬ leaf ⊃
      (∀ (x1: Physical_memory):
        every[Physical_memory, [bool, Address]]
          (λ (x: Physical_memory): TRUE,
           λ (t: [bool, Address]):
             Mem?(t'2'type_of) ∧ 0 ≤ t'2'offset ∧ t'2'offset < max_linear_offset)
          (translate[Physical_memory, pm_phy]
            (1, base, addr', access', segment_to_priv(cs))(x1)))
{-7} pm'mem = linear_pm
{-8} ∀ (s: (pm'states)): OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-9} ∀ (s1, s2: (pm'states)):
      data(read_data(pm_phy, segment_reg_data_type)(CS)(s1)) =
      data(read_data(pm_phy, segment_reg_data_type)(CS)(s2))
1802 {-10} ∀ (s: (pm'states)): OK?(read_data(pm_phy, pdir_data_type)(PDIR)(s))
{-11} ∀ (s1, s2: (pm'states)):
      data(read_data(pm_phy, pdir_data_type)(PDIR)(s1)) =
      data(read_data(pm_phy, pdir_data_type)(PDIR)(s2))
{-12} ∀ (s1, s2: (pm'states)):
      pe_in_pdir_range?(s1,
        restrict[Address, Memory_Address_4G, boolean]
          ((pm_phy'ro_addr ∪ pm_phy'rw_addr)))

```

C.117 Proofs for Linear_Memory_Properties (challenge-linear.pvs)

Using lemma `xlat_pe_in_pdir_range`,

Expanding the definition of `pe_in_pt_range?`,

Instantiating the top quantifier in -4 with the terms: `(s!1)`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `linear_resolve_unchanged_pm_phy.1.1.3.3.2.5.1`.

linear_resolve_unchanged_pm_phy.1.1.3.3.2.5.2:

```

{-1}  unchanged_memory_invariant?(pm_phy' mem, pm_phy' states,
                                         singleton(expr_2_super(translate(0,
                                                                                   d''' base_addr,
                                                                                   addr',
                                                                                   access',
                                                                                   seg-
                                                                                   ment_to_priv(d')))),
                                         ((pm_phy'ro_addr ∪ pm_phy'rw_addr) \ address_block(xlat_idx(
{-2}  ∀ (s: (pm_phy' states)):
      OK?(read_data(pm_phy, pdb_r_data_type)(PDBR)(s)) ∧
      data(read_data(pm_phy, pdb_r_data_type)(PDBR)(s)) = d''
{-3}  ∀ (s: (pm'' states)):
      OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s)) ∧
      d' = data(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-4}  ∀ (cs: Segment_Reg_type, pdb_r: Pdb_r_type):
      ∀ (x1: Physical_memory):
        every[Physical_memory, [bool, Address]]
          (λ (x: Physical_memory): TRUE,
           λ (t: [bool, Address]):
             Mem?(t'2'type_of) ∧
             (0 ≤ t'2'offset ∧ t'2'offset < max_linear_offset) ∧
             IF t'1
               THEN aligned?(22)(offset(t'2))
               ELSE aligned?(12)(offset(t'2))
             ENDIF)
          (translate[Physical_memory, pm_phy]
            (0, pdb_r'base_addr, addr', access', segment_to_priv(cs))(x1))
{-5}  ∀ (cs: Segment_Reg_type, pdb_r: Pdb_r_type, leaf: bool,
      base:
        {b: Memory_Address_4G |
          IF leaf
            THEN aligned?(-1 × (pdir_lvl [Physical_memory, pm_phy] × bits_per_level
              (offset(b))
            ELSE aligned?(bits_per_level[Physical_memory, pm_phy] + pe_size)
              (offset(b))
            ENDIF}):
      ¬ leaf ⊃
      (∀ (x1: Physical_memory):
        every[Physical_memory, [bool, Address]]
          (λ (x: Physical_memory): TRUE,
           λ (t: [bool, Address]):
             Mem?(t'2'type_of) ∧ 0 ≤ t'2'offset ∧ t'2'offset < max_linear_offset)
          (translate[Physical_memory, pm_phy]
            (1, base, addr', access', segment_to_priv(cs))(x1))
{-6}  pm'' mem = linear_pm
{-7}  ∀ (s: (pm'' states)): OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-8}  ∀ (s1, s2: (pm'' states)):
      data(read_data(pm_phy, segment_reg_data_type)(CS)(s1)) =
      data(read_data(pm_phy, segment_reg_data_type)(CS)(s2))
{-9}  ∀ (s: (pm'' states)): OK?(read_data(pm_phy, pdb_r_data_type)(PDBR)(s))
{-10} ∀ (s1, s2: (pm'' states)):
      data(read_data(pm_phy, pdb_r_data_type)(PDBR)(s1)) =
      data(read_data(pm_phy, pdb_r_data_type)(PDBR)(s2))
{-11} ∀ (s1, s2: (pm'' states)):
      pe_in_pdir_range?(s1,
        restrict[Address, Memory_Address_4G, boolean]
          ((pm''ro_addr ∪ pm''rw_addr)))
      =
      pe_in_pdir_range?(s2,
        restrict[Address, Memory_Address_4G, boolean]
          ((pm''ro_addr ∪ pm''rw_addr)))
      ∧
      pe_in_ptab_range?(s1,
        restrict[Address, Memory_Address_4G, boolean]
          ((pm''ro_addr ∪ pm''rw_addr)))

```

C.117 Proofs for Linear_Memory_Properties (challenge-linear.pvs)

Using lemma `xlat_idx_memory_address`,

Using lemma `xlat_idx_memory_address2`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `linear_resolve_unchanged_pm_phy.1.1.3.3.2.5.2`.

linear_resolve_unchanged_pm_phy.1.1.3.3.3:

```

{-1}  ∀ (s: (pm_phy' states)):
      OK?(read_data(pm_phy, pabr_data_type)(PDBR)(s)) ∧
      data(read_data(pm_phy, pabr_data_type)(PDBR)(s)) = d''
{-2}  ∀ (s: (pm' states)):
      OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s)) ∧
      d' = data(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-3}  ∀ (cs: Segment_Reg_type, pabr: Pabr_type):
      ∀ (x1: Physical_memory):
        every[Physical_memory, [bool, Address]]
          (λ (x: Physical_memory): TRUE,
           λ (t: [bool, Address]):
             Mem?(t'2'type_of) ∧
             (0 ≤ t'2'offset ∧ t'2'offset < max_linear_offset) ∧
             IF t'1
               THEN aligned?(22)(offset(t'2))
               ELSE aligned?(12)(offset(t'2))
             ENDIF)
          (translate[Physical_memory, pm_phy]
           (0, pabr'base_addr, addr', access', segment_to_priv(cs))(x1))
{-4}  ∀ (cs: Segment_Reg_type, pabr: Pabr_type, leaf: bool,
      base:
        {b: Memory_Address_4G |
          IF leaf
            THEN aligned?(-1 × (pdir_lvl [Physical_memory, pm_phy] × bits_per_level
              (offset(b))
            ELSE aligned?(bits_per_level[Physical_memory, pm_phy] + pe_size)
              (offset(b))
            ENDIF}):
      ¬ leaf ⊃
      (∀ (x1: Physical_memory):
        every[Physical_memory, [bool, Address]]
          (λ (x: Physical_memory): TRUE,
           λ (t: [bool, Address]):
             Mem?(t'2'type_of) ∧ 0 ≤ t'2'offset ∧ t'2'offset < max_linear_offset)
          (translate[Physical_memory, pm_phy]
           (1, base, addr', access', segment_to_priv(cs))(x1)))
{-5}  pm' mem = linear_pm
{-6}  ∀ (s: (pm' states)): OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-7}  ∀ (s1, s2: (pm' states)):
      data(read_data(pm_phy, segment_reg_data_type)(CS)(s1)) =
      data(read_data(pm_phy, segment_reg_data_type)(CS)(s2))
{-8}  ∀ (s: (pm' states)): OK?(read_data(pm_phy, pabr_data_type)(PDBR)(s))
{-9}  ∀ (s1, s2: (pm' states)):
      data(read_data(pm_phy, pabr_data_type)(PDBR)(s1)) =
      data(read_data(pm_phy, pabr_data_type)(PDBR)(s2))
{-10} ∀ (s1, s2: (pm' states)):
      pe_in_pdir_range?(s1,
        restrict[Address, Memory_Address_4G, boolean]
          ((pm'ro_addr ∪ pm'rw_addr)))
      =
      pe_in_pdir_range?(s2,
        restrict[Address, Memory_Address_4G, boolean]
          ((pm'ro_addr ∪ pm'rw_addr)))
      ∧
      pe_in_ptab_range?(s1,
        restrict[Address, Memory_Address_4G, boolean]
          ((pm'ro_addr ∪ pm'rw_addr)))
      =
      pe_in_ptab_range?(s2,
        restrict[Address, Memory_Address_4G, boolean]
          ((pm'ro_addr ∪ pm'rw_addr)))
{-11} ∀ (s: (pm' states), lvl: Level,
      a:

```

C.117 Proofs for Linear_Memory_Properties (challenge-linear.pvs)

Hiding formulas: 2,

Repeatedly Skolemizing and flattening,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Rewriting using `plain_memory_unchanged_composition` [Physical_memory, [bool, Memory_Address_4G], Address], matching in * where P gets LAMBDA (d: [bool, Memory_Address_4G]): TRUE,

Hiding formulas: 3,

Case splitting on `aligned?(12)(offset(d!3'2))`,

we get 2 subgoals:

linear_resolve_unchanged_pm_phy.1.1.3.3.1:

```

{-1}  aligned?(12)(offset(d'''2))
{-2}  ∀ (s: (pm_phy'states)):
      OK?(translate(0, d'''base_addr, addr', access', segment_to_priv(d'))(s)) ∧
      d''' = data(translate(0, d'''base_addr, addr', access', segment_to_priv(d'))(s))
{-3}  ∀ (s: (pm_phy'states)):
      OK?(read_data(pm_phy, pddb_data_type)(PDBR)(s)) ∧
      data(read_data(pm_phy, pddb_data_type)(PDBR)(s)) = d''
{-4}  ∀ (s: (pm''states)):
      OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s)) ∧
      d' = data(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-5}  ∀ (cs: Segment_Reg_type, pddb: Pddb_type):
      ∀ (x1: Physical_memory):
        every[Physical_memory, [bool, Address]]
          (λ (x: Physical_memory): TRUE,
           λ (t: [bool, Address]):
             Mem?(t'2'type_of) ∧
             (0 ≤ t'2'offset ∧ t'2'offset < max_linear_offset) ∧
             IF t'1
               THEN aligned?(22)(offset(t'2))
               ELSE aligned?(12)(offset(t'2))
             ENDIF)
          (translate[Physical_memory, pm_phy]
            (0, pddb'base_addr, addr', access', segment_to_priv(cs))(x1))
{-6}  ∀ (cs: Segment_Reg_type, pddb: Pddb_type, leaf: bool,
      base:
        {b: Memory_Address_4G |
          IF leaf
            THEN aligned?(-1 × (pdir_lvl [Physical_memory, pm_phy] × bits_per_level
              (offset(b))
            ELSE aligned?(bits_per_level[Physical_memory, pm_phy] + pe_size)
              (offset(b))
            ENDIF}):
      ¬ leaf ⊃
      (∀ (x1: Physical_memory):
        every[Physical_memory, [bool, Address]]
          (λ (x: Physical_memory): TRUE,
           λ (t: [bool, Address]):
             Mem?(t'2'type_of) ∧ 0 ≤ t'2'offset ∧ t'2'offset < max_linear_offset)
          (translate[Physical_memory, pm_phy]
            (1, base, addr', access', segment_to_priv(cs))(x1)))
{-7}  pm''mem = linear_pm
{-8}  ∀ (s: (pm''states)): OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-9}  ∀ (s1, s2: (pm''states)):
      data(read_data(pm_phy, segment_reg_data_type)(CS)(s1)) =
      data(read_data(pm_phy, segment_reg_data_type)(CS)(s2))
{-10} ∀ (s: (pm''states)): OK?(read_data(pm_phy, pddb_data_type)(PDBR)(s))
{-11} ∀ (s1, s2: (pm''states)):
      data(read_data(pm_phy, pddb_data_type)(PDBR)(s1)) =
      data(read_data(pm_phy, pddb_data_type)(PDBR)(s2))
{-12} ∀ (s1, s2: (pm''states)):
      pe_in_pdir_range?(s1,
        restrict[Address, Memory_Address_4G, boolean]
          ((pm''ro_addr ∪ pm''rw_addr)))
      =
      pe_in_pdir_range?(s2,
        restrict[Address, Memory_Address_4G, boolean]
          ((pm''ro_addr ∪ pm''rw_addr)))
      ∧
      pe_in_ptab_range?(s1,
        restrict[Address, Memory_Address_4G, boolean]
          ((pm''ro_addr ∪ pm''rw_addr)))
      =
      pe_in_ptab_range?(s2,
        restrict[Address, Memory_Address_4G, boolean]
          ((pm''ro_addr ∪ pm''rw_addr)))

```

C.117 Proofs for Linear_Memory_Properties (challenge-linear.pvs)

Using lemma `unchanged_memory_invariant_mono`,

Using lemma `xlat_idx_memory_address`,

Using lemma `translate_unchanged_pm_phy_except_pe`,

Simplifying, rewriting, and recording with decision procedures,

Case splitting on `subset?(addresses, difference(union(pm_phy'ro_addr, pm_phy'rw_addr), address_block(xlat_idx(1, d!3'2, addr!1), expt(2, 2))))`,

we get 2 subgoals:

linear_resolve_unchanged_pm_phy.1.1.3.3.3.1.1:

```

{-1} (addresses  $\subseteq$  ((pm_phy'ro_addr  $\cup$  pm_phy'rw_addr) \ address_block(xlat_idx(1, d'''2, addr'), exp
{-2} ((address_block(xlat_idx(1, d'''2, addr'), expt(2, 2))  $\subseteq$  pm_phy'rw_addr)  $\wedge$ 
  ( $\forall$  (s: (pm'states)):
    OK?(read_data(pm_phy, paging_data_type(1))(xlat_idx(1, d'''2, addr'))(s)))
   $\wedge$ 
  ( $\forall$  (s1, s2: (pm'states)):
    set_reference(data(read_data(pm_phy, paging_data_type(1))
      (xlat_idx(1, d'''2, addr'))(s1)),
      Write)
    =
    set_reference(data(read_data(pm_phy, paging_data_type(1))
      (xlat_idx(1, d'''2, addr'))(s2)),
      Write)))
 $\supset$ 
unchanged_memory_invariant?(pm_phy'mem, pm_phy'states,
  singleton(expr_2_super(translate(1,
    d'''2,
    addr',
    access',
    seg-
    ment_to_priv(d')))),
  ((pm_phy'ro_addr  $\cup$  pm_phy'rw_addr) \ address_block(xlat_idx
{-3} xlat_idx(1, d'''2, addr')'type_of = Mem_  $\wedge$ 
  xlat_idx(1, d'''2, addr')'offset < max_linear_offset
{-4} aligned?(12)(offset(d'''2))
{-5}  $\forall$  (s: (pm_phy'states)):
  OK?(translate(0, d''base_addr, addr', access', segment_to_priv(d'))(s))  $\wedge$ 
  d''' = data(translate(0, d''base_addr, addr', access', segment_to_priv(d'))(s))
{-6}  $\forall$  (s: (pm_phy'states)):
  OK?(read_data(pm_phy, pdb_r_data_type)(PDBR)(s))  $\wedge$ 
  data(read_data(pm_phy, pdb_r_data_type)(PDBR)(s)) = d''
{-7}  $\forall$  (s: (pm'states)):
  OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s))  $\wedge$ 
  d' = data(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-8}  $\forall$  (cs: Segment_Reg_type, pdb_r: Pdb_r_type):
   $\forall$  (x1: Physical_memory):
    every[Physical_memory, [bool, Address]]
      ( $\lambda$  (x: Physical_memory): TRUE,
        $\lambda$  (t: [bool, Address]):
         Mem?(t'2'type_of)  $\wedge$ 
         (0  $\leq$  t'2'offset  $\wedge$  t'2'offset < max_linear_offset)  $\wedge$ 
         IF t'1
           THEN aligned?(22)(offset(t'2))
           ELSE aligned?(12)(offset(t'2))
           ENDIF)
      (translate[Physical_memory, pm_phy]
        (0, pdb_r'base_addr, addr', access', segment_to_priv(cs))(x1))
{-9}  $\forall$  (cs: Segment_Reg_type, pdb_r: Pdb_r_type, leaf: bool,
  base:
    {b: Memory_Address_4G |
      IF leaf
        THEN aligned?(-1  $\times$  (pdir_lvl [Physical_memory, pm_phy]  $\times$  bits_per_level
          (offset(b)))
        ELSE aligned?(bits_per_level[Physical_memory, pm_phy] + pe_size)
          (offset(b))
        ENDIF}):
 $\neg$  leaf  $\supset$ 
  ( $\forall$  (x1: Physical_memory):
    every[Physical_memory, [bool, Address]]
      ( $\lambda$  (x: Physical_memory): TRUE,
        $\lambda$  (t: [bool, Address]):
         Mem?(t'2'type_of)  $\wedge$  0  $\leq$  t'2'offset  $\wedge$  t'2'offset < max_linear_offset)
      (translate[Physical_memory, pm_phy]
        (0, pdb_r'base_addr, addr', access', segment_to_priv(cs))(x1))

```

C.117 Proofs for Linear_Memory_Properties (challenge-linear.pvs)

Rewriting using `subset_reflexive`, matching in `*`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 4 subgoals:

linear_resolve_unchanged_pm_phy.1.1.3.3.3.1.1.1:

```

{-1}  (addresses  $\subseteq$  ((pm_phy'ro_addr  $\cup$  pm_phy'rw_addr) \ address_block(xlat_idx(1, d'''2, addr'), exp
{-2}  unchanged_memory_invariant?(pm_phy'mem, pm_phy'states,
                                     singleton(expr_2_super(translate(1,
                                                                    d'''2,
                                                                    addr',
                                                                    access',
                                                                    seg-
ment_to_priv(d')))),
                                     ((pm_phy'ro_addr  $\cup$  pm_phy'rw_addr) \ address_block(xlat_idx(
{-3}  Mem?(xlat_idx(1, d'''2, addr')'type_of)
{-4}  xlat_idx(1, d'''2, addr')'offset < max_linear_offset
{-5}  aligned?(12)(offset(d'''2))
{-6}   $\forall$  (s: (pm_phy'states)):
      OK?(translate(0, d''base_addr, addr', access', segment_to_priv(d'))(s))  $\wedge$ 
      d''' = data(translate(0, d''base_addr, addr', access', segment_to_priv(d'))(s))
{-7}   $\forall$  (s: (pm_phy'states)):
      OK?(read_data(pm_phy, pdbr_data_type)(PDBR)(s))  $\wedge$ 
      data(read_data(pm_phy, pdbr_data_type)(PDBR)(s)) = d''
{-8}   $\forall$  (s: (pm' states)):
      OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s))  $\wedge$ 
      d' = data(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-9}   $\forall$  (cs: Segment_Reg_type, pdbr: Pdbr_type):
       $\forall$  (x1: Physical_memory):
        every[Physical_memory, [bool, Address]]
          ( $\lambda$  (x: Physical_memory): TRUE,
            $\lambda$  (t: [bool, Address]):
             Mem?(t'2'type_of)  $\wedge$ 
             (0  $\leq$  t'2'offset  $\wedge$  t'2'offset < max_linear_offset)  $\wedge$ 
             IF t'1
               THEN aligned?(22)(offset(t'2))
               ELSE aligned?(12)(offset(t'2))
               ENDIF)
          (translate[Physical_memory, pm_phy]
            (0, pdbr'base_addr, addr', access', segment_to_priv(cs))(x1))
{-10}  $\forall$  (cs: Segment_Reg_type, pdbr: Pdbr_type, leaf: bool,
        base:
          {b: Memory_Address_4G |
            IF leaf
              THEN aligned?(-1  $\times$  (pdir_lvl [Physical_memory, pm_phy]  $\times$  bits_per_level
                (offset(b)))
              ELSE aligned?(bits_per_level[Physical_memory, pm_phy] + pe_size)
                (offset(b))
              ENDIF}):
       $\neg$  leaf  $\supset$ 
      ( $\forall$  (x1: Physical_memory):
        every[Physical_memory, [bool, Address]]
          ( $\lambda$  (x: Physical_memory): TRUE,
            $\lambda$  (t: [bool, Address]):
             Mem?(t'2'type_of)  $\wedge$  0  $\leq$  t'2'offset  $\wedge$  t'2'offset < max_linear_offset)
          (translate[Physical_memory, pm_phy]
            (1, base, addr', access', segment_to_priv(cs))(x1)))
{-11} pm'mem = linear_pm
{-12}  $\forall$  (s: (pm' states)): OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-13}  $\forall$  (s1, s2: (pm' states)):
      data(read_data(pm_phy, segment_reg_data_type)(CS)(s1)) =
      data(read_data(pm_phy, segment_reg_data_type)(CS)(s2))
1812 {-14}  $\forall$  (s: (pm' states)): OK?(read_data(pm_phy, pdbr_data_type)(PDBR)(s))
{-15}  $\forall$  (s1, s2: (pm' states)):
      data(read_data(pm_phy, pdbr_data_type)(PDBR)(s1)) =
      data(read_data(pm_phy, pdbr_data_type)(PDBR)(s2))
{-16}  $\forall$  (s1, s2: (pm' states)):
      pe_in_pdir_range?(s1,
                        restrict[Address, Memory_Address_4G, boolean]
                          ((pm'ro_addr  $\cup$  pm'rw_addr)))

```

Keeping (-2 1) and hiding *,

Case splitting on Transformer_Super_Embedding_Expr [Physical_memory, [bool, Address_Helpers.Address]].expr_2_super (Linear_Memory[Physical_memory, pm_phy].translate (1, d!3'2, addr!1, access!1, Segment_Types.segment_to_priv(d!1))) = Transformer_Super_Embedding_Expr [Physical_memory, [bool, IA32.Memory_Address_4G]].expr_2_super (Linear_Memory[Physical_memory, pm_phy].translate (1, d!3'2, addr!1, access!1, Segment_Types.segment_to_priv(d!1))),

we get 2 subgoals:

linear_resolve_unchanged_pm_phy.1.1.3.3.3.1.1.1.1:

{-1}	Transformer_Super_Embedding_Expr [Physical_memory, [bool, Address_Helpers.Address]].expr_2_super (Linear_Memory [Physical_memory, pm_phy].translate (1, d'''^2, addr', access', Segment_Types.segment_to_priv(d')))	
	=	
	Transformer_Super_Embedding_Expr [Physical_memory, [bool, IA32.Memory_Address_4G]].expr_2_super (Linear_Memory [Physical_memory, pm_phy].translate (1, d'''^2, addr', access', Segment_Types.segment_to_priv(d')))	
{-2}	unchanged_memory_invariant?(pm_phy'mem, pm_phy'states, singleton(expr_2_super(translate(1, mem_to_priv(d'))), d'''^2, addr', access', seg- ment_to_priv(d'))), ((pm_phy'ro_addr ∪ pm_phy'rw_addr) \ address_block(xlat_idx(1, d'''^2, ad	
{1}	unchanged_memory_invariant?(pm_phy'mem, pm_phy'states, singleton(expr_2_super(translate(1, mem_to_priv(d'))), d'''^2, addr', access', seg- ment_to_priv(d'))), ((pm_phy'ro_addr ∪ pm_phy'rw_addr) \ address_block(xlat_idx(1, d'''^2, ad	

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of linear_resolve_unchanged_pm_phy.1.1.3.3.3.1.1.1.1.

C Proof scripts

linear_resolve_unchanged_pm_phy.1.1.3.3.3.1.1.1.2:

{-1}	$\text{unchanged_memory_invariant?}(\text{pm_phy}'\text{mem}, \text{pm_phy}'\text{states},$ $\text{singleton}(\text{expr_2_super}(\text{translate}(1,$ $\text{ment_to_priv}(d')))),$	$d''''2,$ $\text{addr}',$ $\text{access}',$ seg-
	$((\text{pm_phy}'\text{ro_addr} \cup \text{pm_phy}'\text{rw_addr}) \setminus \text{address_block}(\text{xlat_idx}(1$	
{1}	$\text{Transformer_Super_Embedding_Expr}[\text{Physical_memory}, [\text{bool}, \text{Ad-}$ $\text{dress_Helpers.Address}]].\text{expr_2_super}$ $(\text{Linear_Memory}[\text{Physical_memory}, \text{pm_phy}].\text{translate}$ $(1, d''''2, \text{addr}', \text{access}', \text{Segment_Types.segment_to_priv}(d')))$ $=$ $\text{Transformer_Super_Embedding_Expr}[\text{Physical_memory}, [\text{bool}, \text{IA32.Memory_Address_4G}]].\text{ex}$ $(\text{Linear_Memory}[\text{Physical_memory}, \text{pm_phy}].\text{translate}$ $(1, d''''2, \text{addr}', \text{access}', \text{Segment_Types.segment_to_priv}(d')))$	
{2}	$\text{unchanged_memory_invariant?}(\text{pm_phy}'\text{mem}, \text{pm_phy}'\text{states},$ $\text{singleton}(\text{expr_2_super}(\text{translate}(1,$ $\text{ment_to_priv}(d')))),$	$d''''2,$ $\text{addr}',$ $\text{access}',$ seg-
	$((\text{pm_phy}'\text{ro_addr} \cup \text{pm_phy}'\text{rw_addr}) \setminus \text{address_block}(\text{xlat_idx}(1$	

Keeping (1) and hiding *,

Applying extensionality,

Expanding the definition(s) of (expr_2_super expr_2_super_res has_next_state),

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of linear_resolve_unchanged_pm_phy.1.1.3.3.3.1.1.1.2.

linear_resolve_unchanged_pm_phy.1.1.3.3.3.1.1.2:

```

{-1} (addresses ⊆ ((pm_phy'ro_addr ∪ pm_phy'rw_addr) \ address_block(xlat_idx(1, d'''2, addr'), expt(2, 2))))
{-2} Mem?(xlat_idx(1, d'''2, addr')'type_of)
{-3} xlat_idx(1, d'''2, addr')'offset < max_linear_offset
{-4} aligned?(12)(offset(d'''2))
{-5} ∀ (s: (pm_phy'states)):
      OK?(translate(0, d'''base_addr, addr', access', segment_to_priv(d'))(s) ∧
            d''' = data(translate(0, d'''base_addr, addr', access', segment_to_priv(d'))(s))
{-6} ∀ (s: (pm_phy'states)):
      OK?(read_data(pm_phy, pdir_data_type)(PDBR)(s) ∧
            data(read_data(pm_phy, pdir_data_type)(PDBR)(s)) = d''
{-7} ∀ (s: (pm' states)):
      OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s) ∧
            d' = data(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-8} ∀ (cs: Segment_Reg_type, pdir: Pdir_type):
      ∀ (x1: Physical_memory):
        every[Physical_memory, [bool, Address]]
          (λ (x: Physical_memory): TRUE,
            λ (t: [bool, Address]):
              Mem?(t'2'type_of) ∧
              (0 ≤ t'2'offset ∧ t'2'offset < max_linear_offset) ∧
              IF t'1
                THEN aligned?(22)(offset(t'2))
                ELSE aligned?(12)(offset(t'2))
              ENDIF)
          (translate[Physical_memory, pm_phy]
            (0, pdir'base_addr, addr', access', segment_to_priv(cs))(x1))
{-9} ∀ (cs: Segment_Reg_type, pdir: Pdir_type, leaf: bool,
      base:
        {b: Memory_Address_4G |
          IF leaf
            THEN aligned?(-1 × (pdir_lvl [Physical_memory, pm_phy] × bits_per_level [Physical
              (offset(b))
            ELSE aligned?(bits_per_level[Physical_memory, pm_phy] + pe_size)
              (offset(b))
            ENDIF}):
        ¬ leaf ⊃
      (∀ (x1: Physical_memory):
        every[Physical_memory, [bool, Address]]
          (λ (x: Physical_memory): TRUE,
            λ (t: [bool, Address]):
              Mem?(t'2'type_of) ∧ 0 ≤ t'2'offset ∧ t'2'offset < max_linear_offset)
          (translate[Physical_memory, pm_phy]
            (1, base, addr', access', segment_to_priv(cs))(x1)))
{-10} pm'mem = linear_pm
{-11} ∀ (s: (pm' states)): OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-12} ∀ (s1, s2: (pm' states)):
      data(read_data(pm_phy, segment_reg_data_type)(CS)(s1)) =
      data(read_data(pm_phy, segment_reg_data_type)(CS)(s2))
{-13} ∀ (s: (pm' states)): OK?(read_data(pm_phy, pdir_data_type)(PDBR)(s))
{-14} ∀ (s1, s2: (pm' states)):
      data(read_data(pm_phy, pdir_data_type)(PDBR)(s1)) =
      data(read_data(pm_phy, pdir_data_type)(PDBR)(s2))
{-15} ∀ (s1, s2: (pm' states)):
      pe_in_pdir_range?(s1,
        restrict[Address, Memory_Address_4G, boolean]
          ((pm'ro_addr ∪ pm'rw_addr)))
      =
      pe_in_pdir_range?(s2,
        restrict[Address, Memory_Address_4G, boolean]
          ((pm'ro_addr ∪ pm'rw_addr)))
      ∧
      pe_in_ptab_range?(s1,
        restrict[Address, Memory_Address_4G, boolean]
          ((pm'ro_addr ∪ pm'rw_addr)))

```


C Proof scripts

Hiding formulas: (2 3),

Repeatedly Skolemizing and flattening,

Instantiating the top quantifier in -19 with the terms: (s1!1 s2!1 1 xlat_idx(1, d!3'2, addr!1)),

we get 2 subgoals:

linear_resolve_unchanged_pm_phy.1.1.3.3.3.1.1.2.1:

```

{-1}  pm' states(s1)
{-2}  pm' states(s2)
{-3}  (addresses ⊆ ((pm_phy'ro_addr ∪ pm_phy'rw_addr) \ address_block(xlat_idx(1, d'''2, addr'), expt(2, 2))))
{-4}  Mem?(xlat_idx(1, d'''2, addr')'type_of)
{-5}  xlat_idx(1, d'''2, addr')'offset < max_linear_offset
{-6}  aligned?(12)(offset(d'''2))
{-7}  ∀ (s: (pm_phy' states)):
      OK?(translate(0, d'''base_addr, addr', access', segment_to_priv(d'))(s) ∧
          d''' = data(translate(0, d'''base_addr, addr', access', segment_to_priv(d'))(s))
{-8}  ∀ (s: (pm_phy' states)):
      OK?(read_data(pm_phy, pdbr_data_type)(PDBR)(s) ∧
          data(read_data(pm_phy, pdbr_data_type)(PDBR)(s)) = d''
{-9}  ∀ (s: (pm' states)):
      OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s) ∧
          d' = data(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-10} ∀ (cs: Segment_Reg_type, pdbr: Pdbr_type):
      ∀ (x1: Physical_memory):
          every[Physical_memory, [bool, Address]]
            (λ (x: Physical_memory): TRUE,
              λ (t: [bool, Address]):
                Mem?(t'2'type_of) ∧
                (0 ≤ t'2'offset ∧ t'2'offset < max_linear_offset) ∧
                IF t'1
                  THEN aligned?(22)(offset(t'2))
                  ELSE aligned?(12)(offset(t'2))
                ENDIF)
            (translate[Physical_memory, pm_phy]
              (0, pdbr'base_addr, addr', access', segment_to_priv(cs))(x1))
{-11} ∀ (cs: Segment_Reg_type, pdbr: Pdbr_type, leaf: bool,
      base:
        {b: Memory_Address_4G |
          IF leaf
            THEN aligned?(-1 × (pdir_lvl [Physical_memory, pm_phy] × bits_per_level [Physical
              (offset(b))
            ELSE aligned?(bits_per_level[Physical_memory, pm_phy] + pe_size)
              (offset(b))
            ENDIF}):
        ¬ leaf ⊃
        (∀ (x1: Physical_memory):
          every[Physical_memory, [bool, Address]]
            (λ (x: Physical_memory): TRUE,
              λ (t: [bool, Address]):
                Mem?(t'2'type_of) ∧ 0 ≤ t'2'offset ∧ t'2'offset < max_linear_offset)
            (translate[Physical_memory, pm_phy]
              (1, base, addr', access', segment_to_priv(cs))(x1)))
{-12} pm' mem = linear_pm
{-13} ∀ (s: (pm' states)): OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-14} ∀ (s1, s2: (pm' states)):
      data(read_data(pm_phy, segment_reg_data_type)(CS)(s1)) =
        data(read_data(pm_phy, segment_reg_data_type)(CS)(s2))
{-15} ∀ (s: (pm' states)): OK?(read_data(pm_phy, pdbr_data_type)(PDBR)(s))
{-16} ∀ (s1, s2: (pm' states)):
      data(read_data(pm_phy, pdbr_data_type)(PDBR)(s1)) =
        data(read_data(pm_phy, pdbr_data_type)(PDBR)(s2))
{-17} ∀ (s1, s2: (pm' states)):
      pe_in_pdir_range?(s1,
        restrict[Address, Memory_Address_4G, boolean]
          ((pm'ro_addr ∪ pm'rw_addr)))
      =
      pe_in_pdir_range?(s2,
        restrict[Address, Memory_Address_4G, boolean]
          ((pm'ro_addr ∪ pm'rw_addr)))
      ∧

```

C Proof scripts

which is trivially true.

This completes the proof of `linear_resolve_unchanged_pm_phy.1.1.3.3.3.1.1.2.1.`

linear_resolve_unchanged_pm_phy.1.1.3.3.3.1.1.2.2:

```

{-1}  pm' states(s1)
{-2}  pm' states(s2)
{-3}  (addresses ⊆ ((pm_phy'ro_addr ∪ pm_phy'rw_addr) \ address_block(xlat_idx(1, d'''2, addr'), expt(2, 2))))
{-4}  Mem?(xlat_idx(1, d'''2, addr')'type_of)
{-5}  xlat_idx(1, d'''2, addr')'offset < max_linear_offset
{-6}  aligned?(12)(offset(d'''2))
{-7}  ∀ (s: (pm_phy' states)):
      OK?(translate(0, d'''base_addr, addr', access', segment_to_priv(d'))(s) ∧
          d''' = data(translate(0, d'''base_addr, addr', access', segment_to_priv(d'))(s))
{-8}  ∀ (s: (pm_phy' states)):
      OK?(read_data(pm_phy, pdb_data_type)(PDBR)(s) ∧
          data(read_data(pm_phy, pdb_data_type)(PDBR)(s)) = d''
{-9}  ∀ (s: (pm' states)):
      OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s) ∧
          d' = data(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-10} ∀ (cs: Segment_Reg_type, pdb: Pdb_type):
      ∀ (x1: Physical_memory):
          every[Physical_memory, [bool, Address]]
            (λ (x: Physical_memory): TRUE,
              λ (t: [bool, Address]):
                Mem?(t'2'type_of) ∧
                (0 ≤ t'2'offset ∧ t'2'offset < max_linear_offset) ∧
                IF t'1
                  THEN aligned?(22)(offset(t'2))
                  ELSE aligned?(12)(offset(t'2))
                ENDIF)
            (translate[Physical_memory, pm_phy]
              (0, pdb'base_addr, addr', access', segment_to_priv(cs))(x1))
{-11} ∀ (cs: Segment_Reg_type, pdb: Pdb_type, leaf: bool,
      base:
        {b: Memory_Address_4G |
          IF leaf
            THEN aligned?(-1 × (pdir_lvl [Physical_memory, pm_phy] × bits_per_level [Physical
              (offset(b))
            ELSE aligned?(bits_per_level[Physical_memory, pm_phy] + pe_size)
              (offset(b))
            ENDIF}):
      ¬ leaf ⊃
      (∀ (x1: Physical_memory):
        every[Physical_memory, [bool, Address]]
          (λ (x: Physical_memory): TRUE,
            λ (t: [bool, Address]):
              Mem?(t'2'type_of) ∧ 0 ≤ t'2'offset ∧ t'2'offset < max_linear_offset)
          (translate[Physical_memory, pm_phy]
            (1, base, addr', access', segment_to_priv(cs))(x1)))
{-12} pm' mem = linear_pm
{-13} ∀ (s: (pm' states)): OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-14} ∀ (s1, s2: (pm' states)):
      data(read_data(pm_phy, segment_reg_data_type)(CS)(s1)) =
      data(read_data(pm_phy, segment_reg_data_type)(CS)(s2))
{-15} ∀ (s: (pm' states)): OK?(read_data(pm_phy, pdb_data_type)(PDBR)(s))
{-16} ∀ (s1, s2: (pm' states)):
      data(read_data(pm_phy, pdb_data_type)(PDBR)(s1)) =
      data(read_data(pm_phy, pdb_data_type)(PDBR)(s2))
{-17} ∀ (s1, s2: (pm' states)):
      pe_in_pdir_range?(s1,
        restrict[Address, Memory_Address_4G, boolean]
          ((pm'ro_addr ∪ pm'rw_addr)))
      =
      pe_in_pdir_range?(s2,
        restrict[Address, Memory_Address_4G, boolean]
          ((pm'ro_addr ∪ pm'rw_addr)))
      ∧

```

C Proof scripts

Using lemma `xlat_pe_in_pt_range`,

Instantiating quantified variables,

Case splitting on `pm!1'states(state(read_data(pm_phy, pdbr_data_type)(PDBR)(s1!1)))`,

we get 3 subgoals:

linear_resolve_unchanged_pm_phy.1.1.3.3.3.1.1.2.2.1:

```

{-1} pm' states(state(read_data(pm_phy, pdir_data_type)(PDBR)(s'_1)))
{-2} union(pm' ro_addr, pm' rw_addr)(addr') ∧
      is_linear_plain_memory?(pm') ∧
      pm' states(s'_1) ∧
      pm' states(s'_1) ∧
      OK?(translate(pdir_lvl,
                    data(read_data(pm_phy, pdir_data_type)(PDBR)(s'_1))'base_addr, addr',
                    access', segment_to_priv(d'))
            (state(read_data(pm_phy, pdir_data_type)(PDBR)(s'_1))))
      ∧
      ¬ data(translate(pdir_lvl,
                      data(read_data(pm_phy, pdir_data_type)(PDBR)(s'_1))'base_addr,
                      addr', access', segment_to_priv(d'))
            (state(read_data(pm_phy, pdir_data_type)(PDBR)(s'_1))))'1
      ⊃
      pe_in_pt_range?(s'_1,
                      restrict[Address, Memory_Address_4G, boolean]
                        ((pm' ro_addr ∪ pm' rw_addr),
                         ptab_lvl)
                      (xlat_idx(ptab_lvl,
                                data(translate(pdir_lvl,
                                                data(read_data(pm_phy, pdir_data_type)
                                                            (PDBR)
                                                            (s'_1))'base_addr,
                                                            addr', access', seg-
ment_to_priv(d'))
                                (state(read_data(pm_phy, pdir_data_type)
                                        (PDBR)
                                        (s'_1))))'2,
                                addr'))
{-3} pm' states(s'_1)
{-4} pm' states(s'_2)
{-5} (addresses ⊆ ((pm_phy'ro_addr ∪ pm_phy'rw_addr) \ address_block(xlat_idx(1, d'''2, addr'), expt(2, 2))))
{-6} Mem?(xlat_idx(1, d'''2, addr')'type_of)
{-7} xlat_idx(1, d'''2, addr')'offset < max_linear_offset
{-8} aligned?(12)(offset(d'''2))
{-9} ∀ (s: (pm_phy'states)):
      OK?(translate(0, d'''base_addr, addr', access', segment_to_priv(d'))(s)) ∧
      d''' = data(translate(0, d'''base_addr, addr', access', segment_to_priv(d'))(s))
{-10} OK?(read_data(pm_phy, pdir_data_type)(PDBR)(s'_1)) ∧
      data(read_data(pm_phy, pdir_data_type)(PDBR)(s'_1)) = d''
{-11} ∀ (s: (pm' states)):
      OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s)) ∧
      d' = data(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-12} ∀ (cs: Segment_Reg_type, pdir: Pdir_type):
      ∀ (x1: Physical_memory):
        every[Physical_memory, [bool, Address]]
          (λ (x: Physical_memory): TRUE,
           λ (t: [bool, Address]):
             Mem?(t'2'type_of) ∧
             (0 ≤ t'2'offset ∧ t'2'offset < max_linear_offset) ∧
             IF t'1
               THEN aligned?(22)(offset(t'2))
               ELSE aligned?(12)(offset(t'2))
             ENDIF)
          (translate[Physical_memory, pm_phy]
            (0, pdir'base_addr, addr', access', segment_to_priv(cs))(x1))
{-13} ∀ (cs: Segment_Reg_type, pdir: Pdir_type, leaf: bool,
        base:
          {b: Memory_Address_4G |
            IF leaf
              THEN aligned?(-1 × (pdir_lvl [Physical_memory, pm_phy] × bits_per_level [Physical
            (offset(b))

```

C Proof scripts

Instantiating quantified variables,

we get 3 subgoals:

linear_resolve_unchanged_pm_phy.1.1.3.3.3.1.1.2.2.1.1:

```

{-1} pm' states(state(read_data(pm_phy, pabr_data_type)(PDBR)(s'_1)))
{-2} union(pm' ro_addr, pm' rw_addr)(addr') ^
      is_linear_plain_memory?(pm') ^
      pm' states(s'_1) ^
      pm' states(s'_1) ^
      OK?(translate(pdir_lvl,
                    data(read_data(pm_phy, pabr_data_type)(PDBR)(s'_1))'base_addr, addr',
                    access', segment_to_priv(d'))
            (state(read_data(pm_phy, pabr_data_type)(PDBR)(s'_1))))
      ^
      ¬ data(translate(pdir_lvl,
                       data(read_data(pm_phy, pabr_data_type)(PDBR)(s'_1))'base_addr,
                       addr', access', segment_to_priv(d'))
             (state(read_data(pm_phy, pabr_data_type)(PDBR)(s'_1))))'1
      ⊃
      pe_in_pt_range?(s'_1,
                      restrict[Address, Memory_Address_4G, boolean]
                        ((pm' ro_addr ∪ pm' rw_addr),
                         ptab_lvl)
                      (xlat_idx(ptab_lvl,
                                data(translate(pdir_lvl,
                                                data(read_data(pm_phy, pabr_data_type)
                                                            (PDBR)
                                                            (s'_1))'base_addr,
                                                            addr', access', seg-
ment_to_priv(d'))
                                (state(read_data(pm_phy, pabr_data_type)
                                            (PDBR)
                                            (s'_1))))'2,
                                addr'))
{-3} pm' states(s'_1)
{-4} pm' states(s'_2)
{-5} (addresses ⊆ ((pm_phy'ro_addr ∪ pm_phy'rw_addr) \ address_block(xlat_idx(1, d'''2, addr'), expt(2, 2))))
{-6} Mem?(xlat_idx(1, d'''2, addr')'type_of)
{-7} xlat_idx(1, d'''2, addr')'offset < max_linear_offset
{-8} aligned?(12)(offset(d'''2))
{-9} OK?(translate(0, d''base_addr, addr', access', segment_to_priv(d'))
            (state(read_data(pm_phy, pabr_data_type)(PDBR)(s'_1))))
      ^
      d''' =
      data(translate(0, d''base_addr, addr', access', segment_to_priv(d'))
            (state(read_data(pm_phy, pabr_data_type)(PDBR)(s'_1))))
{-10} OK?(read_data(pm_phy, pabr_data_type)(PDBR)(s'_1)) ^
      data(read_data(pm_phy, pabr_data_type)(PDBR)(s'_1)) = d''
{-11} ∀ (s: (pm' states)):
      OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s)) ^
      d' = data(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-12} ∀ (cs: Segment_Reg_type, pabr: Pabr_type):
      ∀ (x1: Physical_memory):
      every[Physical_memory, [bool, Address]]
        (λ (x: Physical_memory): TRUE,
         λ (t: [bool, Address]):
         Mem?(t'2'type_of) ^
         (0 ≤ t'2'offset ^ t'2'offset < max_linear_offset) ^
         IF t'1
           THEN aligned?(22)(offset(t'2))
           ELSE aligned?(12)(offset(t'2))
         ENDIF)
      (translate[Physical_memory, pm_phy]
       (0, pabr'base_addr, addr', access', segment_to_priv(cs))(x1))
{-13} ∀ (cs: Segment_Reg_type, pabr: Pabr_type, leaf: bool,
        base:
        {b: Memory_Address_4G |

```


C Proof scripts

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Using lemma `xlat_idx_memory_address2`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `linear_resolve_unchanged_pm_phy.1.1.3.3.3.1.1.2.2.1.1`.

linear_resolve_unchanged_pm_phy.1.1.3.3.3.1.1.2.2.1.2:

```

{-1} pm' states(state(read_data(pm_phy, pdir_data_type)(PDBR)(s'_1)))
{-2} union(pm' ro_addr, pm' rw_addr)(addr') ∧
      is_linear_plain_memory?(pm') ∧
      pm' states(s'_1) ∧
      pm' states(s'_1) ∧
      OK?(translate(pdir_lvl,
                    data(read_data(pm_phy, pdir_data_type)(PDBR)(s'_1))'base_addr, addr',
                    access', segment_to_priv(d'))
            (state(read_data(pm_phy, pdir_data_type)(PDBR)(s'_1))))
      ∧
      ¬ data(translate(pdir_lvl,
                      data(read_data(pm_phy, pdir_data_type)(PDBR)(s'_1))'base_addr,
                      addr', access', segment_to_priv(d'))
              (state(read_data(pm_phy, pdir_data_type)(PDBR)(s'_1))))'1
      ⊃
      pe_in_pt_range?(s'_1,
                      restrict[Address, Memory_Address_4G, boolean]
                        ((pm' ro_addr ∪ pm' rw_addr),
                         ptab_lvl)
                      (xlat_idx(ptab_lvl,
                                data(translate(pdir_lvl,
                                                data(read_data(pm_phy, pdir_data_type)
                                                            (PDBR)
                                                            (s'_1))'base_addr,
                                                            addr', access', seg-
ment_to_priv(d'))
                                (state(read_data(pm_phy, pdir_data_type)
                                        (PDBR)
                                        (s'_1))))'2,
                                addr'))
{-3} pm' states(s'_1)
{-4} pm' states(s'_2)
{-5} (addresses ⊆ ((pm_phy'ro_addr ∪ pm_phy'rw_addr) \ address_block(xlat_idx(1, d'''2, addr'), expt(2, 2))))
{-6} Mem?(xlat_idx(1, d'''2, addr')'type_of)
{-7} xlat_idx(1, d'''2, addr')'offset < max_linear_offset
{-8} aligned?(12)(offset(d'''2))
{-9} OK?(read_data(pm_phy, pdir_data_type)(PDBR)(s'_1) ∧
        data(read_data(pm_phy, pdir_data_type)(PDBR)(s'_1)) = d'')
{-10} ∀ (s: (pm' states)):
      OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s) ∧
          d' = data(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-11} ∀ (cs: Segment_Reg_type, pdir: Pdir_type):
      ∀ (x1: Physical_memory):
        every[Physical_memory, [bool, Address]]
          (λ (x: Physical_memory): TRUE,
           λ (t: [bool, Address]):
             Mem?(t'2'type_of) ∧
             (0 ≤ t'2'offset ∧ t'2'offset < max_linear_offset) ∧
             IF t'1
               THEN aligned?(22)(offset(t'2))
               ELSE aligned?(12)(offset(t'2))
             ENDIF)
          (translate[Physical_memory, pm_phy]
            (0, pdir'base_addr, addr', access', segment_to_priv(cs))(x1))
{-12} ∀ (cs: Segment_Reg_type, pdir: Pdir_type, leaf: bool,
        base:
          {b: Memory_Address_4G |
            IF leaf
              THEN aligned?(-1 × (pdir_lvl [Physical_memory, pm_phy] × bits_per_level [Physical
              (offset(b))
            ELSE aligned?(bits_per_level[Physical_memory, pm_phy] + pe_size)
              (offset(b))
            ENDIF}):

```

C Proof scripts

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `linear_resolve_unchanged_pm_phy.1.1.3.3.3.1.1.2.2.1.2.`

linear_resolve_unchanged_pm_phy.1.1.3.3.3.1.1.2.2.1.3:

```

{-1} pm' states(state(read_data(pm_phy, pdir_data_type)(PDBR)(s1)))
{-2} union(pm' ro_addr, pm' rw_addr)(addr') ^
      is_linear_plain_memory?(pm') ^
      pm' states(s1) ^
      pm' states(s1) ^
      OK?(translate(pdir_lvl,
                    data(read_data(pm_phy, pdir_data_type)(PDBR)(s1))'base_addr, addr',
                    access', segment_to_priv(d'))
            (state(read_data(pm_phy, pdir_data_type)(PDBR)(s1))))
      ^
      ¬ data(translate(pdir_lvl,
                       data(read_data(pm_phy, pdir_data_type)(PDBR)(s1))'base_addr,
                       addr', access', segment_to_priv(d'))
              (state(read_data(pm_phy, pdir_data_type)(PDBR)(s1))))'1
      ⊃
      pe_in_pt_range?(s1,
                      restrict[Address, Memory_Address_4G, boolean]
                        ((pm' ro_addr ∪ pm' rw_addr),
                         ptab_lvl)
                      (xlat_idx(ptab_lvl,
                                data(translate(pdir_lvl,
                                                data(read_data(pm_phy, pdir_data_type)
                                                            (PDBR)
                                                            (s1))'base_addr,
                                                            addr', access', seg-
ment_to_priv(d'))
                                (state(read_data(pm_phy, pdir_data_type)
                                            (PDBR)
                                            (s1))))'2,
                                addr'))
{-3} pm' states(s1)
{-4} pm' states(s2)
{-5} (addresses ⊆ ((pm_phy'ro_addr ∪ pm_phy'rw_addr) \ address_block(xlat_idx(1, d'''2, addr'), expt(2, 2))))
{-6} Mem?(xlat_idx(1, d'''2, addr')'type_of)
{-7} xlat_idx(1, d'''2, addr')'offset < max_linear_offset
{-8} aligned?(12)(offset(d'''2))
{-9} ∀ (s: (pm_phy'states)):
      OK?(translate(0, d'''base_addr, addr', access', segment_to_priv(d'))(s)) ^
      d''' = data(translate(0, d'''base_addr, addr', access', segment_to_priv(d'))(s))
{-10} OK?(read_data(pm_phy, pdir_data_type)(PDBR)(s1)) ^
      data(read_data(pm_phy, pdir_data_type)(PDBR)(s1)) = d''
{-11} ∀ (s: (pm' states)):
      OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s)) ^
      d' = data(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-12} ∀ (cs: Segment_Reg_type, pdir: Pdir_type):
      ∀ (x1: Physical_memory):
        every[Physical_memory, [bool, Address]]
          (λ (x: Physical_memory): TRUE,
           λ (t: [bool, Address]):
             Mem?(t'2'type_of) ^
             (0 ≤ t'2'offset ^ t'2'offset < max_linear_offset) ^
             IF t'1
               THEN aligned?(22)(offset(t'2))
               ELSE aligned?(12)(offset(t'2))
             ENDIF)
          (translate[Physical_memory, pm_phy]
            (0, pdir'base_addr, addr', access', segment_to_priv(cs))(x1))
{-13} ∀ (cs: Segment_Reg_type, pdir: Pdir_type, leaf: bool,
        base:
          {b: Memory_Address_4G |
            IF leaf
              THEN aligned?(-1 × (pdir_lvl [Physical_memory, pm_phy] × bits_per_level [Physical
            (offset(b))

```

C Proof scripts

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `linear_resolve_unchanged_pm_phy.1.1.3.3.3.1.1.2.2.1.3.`

linear_resolve_unchanged_pm_phy.1.1.3.3.3.1.1.2.2.2:

```

{-1}  union(pm'ro_addr, pm'rw_addr)(addr') ∧
      is_linear_plain_memory?(pm') ∧
      pm'states(s'_1) ∧
      pm'states(s'_1) ∧
      OK?(translate(pdir_lvl,
                    data(read_data(pm_phy, pdir_data_type)(PDBR)(s'_1))'base_addr, addr',
                    access', segment_to_priv(d'))
            (state(read_data(pm_phy, pdir_data_type)(PDBR)(s'_1))))
      ∧
      ¬ data(translate(pdir_lvl,
                      data(read_data(pm_phy, pdir_data_type)(PDBR)(s'_1))'base_addr,
                      addr', access', segment_to_priv(d'))
            (state(read_data(pm_phy, pdir_data_type)(PDBR)(s'_1))))'1
      ⊃
      pe_in_pt_range?(s'_1,
                      restrict[Address, Memory_Address_4G, boolean]
                        ((pm'ro_addr ∪ pm'rw_addr)),
                      ptab_lvl)
      (xlat_idx(ptab_lvl,
                data(translate(pdir_lvl,
                              data(read_data(pm_phy, pdir_data_type)
                                    (PDBR)
                                    (s'_1))'base_addr,
                              addr', access', seg-
ment_to_priv(d'))
                    (state(read_data(pm_phy, pdir_data_type)
                              (PDBR)
                              (s'_1))))'2,
                addr'))
{-2}  pm'states(s'_1)
{-3}  pm'states(s'_2)
{-4}  (addresses ⊆ ((pm_phy'ro_addr ∪ pm_phy'rw_addr) \ address_block(xlat_idx(1, d'''2, addr'), expt(2, 2))))
{-5}  Mem?(xlat_idx(1, d'''2, addr')'type_of)
{-6}  xlat_idx(1, d'''2, addr')'offset < max_linear_offset
{-7}  aligned?(12)(offset(d'''2))
{-8}  ∀ (s: (pm_phy'states)):
      OK?(translate(0, d'''base_addr, addr', access', segment_to_priv(d'))(s)) ∧
      d''' = data(translate(0, d'''base_addr, addr', access', segment_to_priv(d'))(s))
{-9}  OK?(read_data(pm_phy, pdir_data_type)(PDBR)(s'_1)) ∧
      data(read_data(pm_phy, pdir_data_type)(PDBR)(s'_1)) = d''
{-10} ∀ (s: (pm'states)):
      OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s)) ∧
      d' = data(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-11} ∀ (cs: Segment_Reg_type, pdir: Pdir_type):
      ∀ (x1: Physical_memory):
        every[Physical_memory, [bool, Address]]
          (λ (x: Physical_memory): TRUE,
           λ (t: [bool, Address]):
             Mem?(t'2'type_of) ∧
             (0 ≤ t'2'offset ∧ t'2'offset < max_linear_offset) ∧
             IF t'1
               THEN aligned?(22)(offset(t'2))
               ELSE aligned?(12)(offset(t'2))
             ENDIF
          (translate[Physical_memory, pm_phy]
            (0, pdir'base_addr, addr', access', segment_to_priv(cs))(x1))1829
{-12} ∀ (cs: Segment_Reg_type, pdir: Pdir_type, leaf: bool,
      base:
        {b: Memory_Address_4G |
          IF leaf
            THEN aligned?(−1 × (pdir_lvl [Physical_memory, pm_phy] × bits_per_level [Physical
              (offset(b))
            ELSE aligned?(bits_per_level[Physical_memory, pm_phy] + pe_size)

```

C Proof scripts

Hiding formulas: -1,

Using lemma `pm_unchanged_singleton_linear_resolve_reg_transformers`,

Expanding the definition of `linear_resolve_register_transformers`,

Using lemma `unchanged_memory_invariant_invariant`,

Using lemma `expr_transformer_invariant_next_ok`,

Using lemma `pm_states`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `linear_resolve_unchanged_pm_phy.1.1.3.3.3.1.1.2.2.2`.

linear_resolve_unchanged_pm_phy.1.1.3.3.3.1.1.2.2.3:

```

{-1} union(pm'ro_addr, pm'rw_addr)(addr') ∧
      is_linear_plain_memory?(pm') ∧
      pm'states(s'_1) ∧
      pm'states(s'_1) ∧
      OK?(translate(pdir_lvl,
                    data(read_data(pm_phy, pdir_data_type)(PDBR)(s'_1))'base_addr, addr',
                    access', segment_to_priv(d'))
              (state(read_data(pm_phy, pdir_data_type)(PDBR)(s'_1))))
      ∧
      ¬ data(translate(pdir_lvl,
                      data(read_data(pm_phy, pdir_data_type)(PDBR)(s'_1))'base_addr,
                      addr', access', segment_to_priv(d'))
              (state(read_data(pm_phy, pdir_data_type)(PDBR)(s'_1))))'1
      ⊃
      pe_in_pt_range?(s'_1,
                      restrict[Address, Memory_Address_4G, boolean]
                        ((pm'ro_addr ∪ pm'rw_addr),
                         ptab_lvl)
                      (xlat_idx(ptab_lvl,
                                data(translate(pdir_lvl,
                                                data(read_data(pm_phy, pdir_data_type)
                                                            (PDBR)
                                                            (s'_1))'base_addr,
                                                addr', access', seg-
ment_to_priv(d'))
                                (state(read_data(pm_phy, pdir_data_type)
                                        (PDBR)
                                        (s'_1))))'2,
                                addr'))
{-2} pm'states(s'_1)
{-3} pm'states(s'_2)
{-4} (addresses ⊆ ((pm_phy'ro_addr ∪ pm_phy'rw_addr) \ address_block(xlat_idx(1, d'''2, addr'), expt(2, 2))))
{-5} Mem?(xlat_idx(1, d'''2, addr')'type_of)
{-6} xlat_idx(1, d'''2, addr')'offset < max_linear_offset
{-7} aligned?(12)(offset(d'''2))
{-8} ∀ (s: (pm_phy'states)):
      OK?(translate(0, d'''base_addr, addr', access', segment_to_priv(d'))(s)) ∧
      d''' = data(translate(0, d'''base_addr, addr', access', segment_to_priv(d'))(s))
{-9} OK?(read_data(pm_phy, pdir_data_type)(PDBR)(s'_1)) ∧
      data(read_data(pm_phy, pdir_data_type)(PDBR)(s'_1)) = d''
{-10} ∀ (s: (pm'states)):
      OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s)) ∧
      d' = data(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-11} ∀ (cs: Segment_Reg_type, pdir: Pdir_type):
      ∀ (x1: Physical_memory):
        every[Physical_memory, [bool, Address]]
          (λ (x: Physical_memory): TRUE,
           λ (t: [bool, Address]):
             Mem?(t'2'type_of) ∧
             (0 ≤ t'2'offset ∧ t'2'offset < max_linear_offset) ∧
             IF t'1
               THEN aligned?(22)(offset(t'2))
               ELSE aligned?(12)(offset(t'2))
             ENDIF
          (translate[Physical_memory, pm_phy]
            (0, pdir'base_addr, addr', access', segment_to_priv(cs))(x1))1831
{-12} ∀ (cs: Segment_Reg_type, pdir: Pdir_type, leaf: bool,
        base:
          {b: Memory_Address_4G |
            IF leaf
              THEN aligned?(-1 × (pdir_lvl [Physical_memory, pm_phy] × bits_per_level [Physical
              (offset(b))
            ELSE aligned?(bits_per_level[Physical_memory, pm_phy] + pe_size)

```


C Proof scripts

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `linear_resolve_unchanged_pm_phy.1.1.3.3.3.1.1.2.2.3`.

linear_resolve_unchanged_pm_phy.1.1.3.3.3.1.1.3:

```

{-1} (addresses ⊆ ((pm_phy'ro_addr ∪ pm_phy'rw_addr) \ address_block(xlat_idx(1, d'''2, addr'), expt(2, 2))))
{-2} Mem?(xlat_idx(1, d'''2, addr')'type_of)
{-3} xlat_idx(1, d'''2, addr')'offset < max_linear_offset
{-4} aligned?(12)(offset(d'''2))
{-5} ∀ (s: (pm_phy'states)):
      OK?(translate(0, d'''base_addr, addr', access', segment_to_priv(d'))(s) ∧
            d''' = data(translate(0, d'''base_addr, addr', access', segment_to_priv(d'))(s))
{-6} ∀ (s: (pm_phy'states)):
      OK?(read_data(pm_phy, pdir_data_type)(PDBR)(s) ∧
            data(read_data(pm_phy, pdir_data_type)(PDBR)(s)) = d''
{-7} ∀ (s: (pm' states)):
      OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s) ∧
            d' = data(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-8} ∀ (cs: Segment_Reg_type, pdir: Pdir_type):
      ∀ (x1: Physical_memory):
        every[Physical_memory, [bool, Address]]
          (λ (x: Physical_memory): TRUE,
            λ (t: [bool, Address]):
              Mem?(t'2'type_of) ∧
              (0 ≤ t'2'offset ∧ t'2'offset < max_linear_offset) ∧
              IF t'1
                THEN aligned?(22)(offset(t'2))
                ELSE aligned?(12)(offset(t'2))
              ENDIF)
          (translate[Physical_memory, pm_phy]
            (0, pdir'base_addr, addr', access', segment_to_priv(cs))(x1))
{-9} ∀ (cs: Segment_Reg_type, pdir: Pdir_type, leaf: bool,
      base:
        {b: Memory_Address_4G |
          IF leaf
            THEN aligned?(-1 × (pdir_lvl [Physical_memory, pm_phy] × bits_per_level [Physical
              (offset(b))
            ELSE aligned?(bits_per_level[Physical_memory, pm_phy] + pe_size)
              (offset(b))
            ENDIF}):
      ¬ leaf ⊃
      (∀ (x1: Physical_memory):
        every[Physical_memory, [bool, Address]]
          (λ (x: Physical_memory): TRUE,
            λ (t: [bool, Address]):
              Mem?(t'2'type_of) ∧ 0 ≤ t'2'offset ∧ t'2'offset < max_linear_offset)
          (translate[Physical_memory, pm_phy]
            (1, base, addr', access', segment_to_priv(cs))(x1)))
{-10} pm'mem = linear_pm
{-11} ∀ (s: (pm' states)): OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-12} ∀ (s1, s2: (pm' states)):
      data(read_data(pm_phy, segment_reg_data_type)(CS)(s1)) =
      data(read_data(pm_phy, segment_reg_data_type)(CS)(s2))
{-13} ∀ (s: (pm' states)): OK?(read_data(pm_phy, pdir_data_type)(PDBR)(s))
{-14} ∀ (s1, s2: (pm' states)):
      data(read_data(pm_phy, pdir_data_type)(PDBR)(s1)) =
      data(read_data(pm_phy, pdir_data_type)(PDBR)(s2))
{-15} ∀ (s1, s2: (pm' states)):
      pe_in_pdir_range?(s1,
        restrict[Address, Memory_Address_4G, boolean]
          ((pm'ro_addr ∪ pm'rw_addr)))
      =
      pe_in_pdir_range?(s2,
        restrict[Address, Memory_Address_4G, boolean]
          ((pm'ro_addr ∪ pm'rw_addr)))
      ∧
      pe_in_ptab_range?(s1,
        restrict[Address, Memory_Address_4G, boolean]
          ((pm'ro_addr ∪ pm'rw_addr)))

```

C Proof scripts

Hiding formulas: (3 2),

Repeatedly Skolemizing and flattening,

Instantiating the top quantifier in -17 with the terms: (s!2 1 xlat_idx(1, d!3'2, addr!1)),

we get 2 subgoals:

linear_resolve_unchanged_pm_phy.1.1.3.3.3.1.1.3.1:

```

{-1} pm' states(s'')
{-2} (addresses ⊆ ((pm_phy'ro_addr ∪ pm_phy'rw_addr) \ address_block(xlat_idx(1, d'''2, addr'), expt(2, 2))))
{-3} Mem?(xlat_idx(1, d'''2, addr')'type_of)
{-4} xlat_idx(1, d'''2, addr')'offset < max_linear_offset
{-5} aligned?(12)(offset(d'''2))
{-6} ∀ (s: (pm_phy' states)):
      OK?(translate(0, d''base_addr, addr', access', segment_to_priv(d'))(s) ∧
          d''' = data(translate(0, d''base_addr, addr', access', segment_to_priv(d'))(s))
{-7} ∀ (s: (pm_phy' states)):
      OK?(read_data(pm_phy, pdir_data_type)(PDBR)(s) ∧
          data(read_data(pm_phy, pdir_data_type)(PDBR)(s)) = d''
{-8} ∀ (s: (pm' states)):
      OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s) ∧
          d' = data(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-9} ∀ (cs: Segment_Reg_type, pdir: Pdir_type):
      ∀ (x1: Physical_memory):
          every[Physical_memory, [bool, Address]]
            (λ (x: Physical_memory): TRUE,
              λ (t: [bool, Address]):
                Mem?(t'2'type_of) ∧
                (0 ≤ t'2'offset ∧ t'2'offset < max_linear_offset) ∧
                IF t'1
                  THEN aligned?(22)(offset(t'2))
                  ELSE aligned?(12)(offset(t'2))
                ENDIF)
            (translate[Physical_memory, pm_phy]
              (0, pdir'base_addr, addr', access', segment_to_priv(cs))(x1))
{-10} ∀ (cs: Segment_Reg_type, pdir: Pdir_type, leaf: bool,
        base:
          {b: Memory_Address_4G |
            IF leaf
              THEN aligned?(-1 × (pdir_lvl [Physical_memory, pm_phy] × bits_per_level [Physical
                (offset(b))
              ELSE aligned?(bits_per_level[Physical_memory, pm_phy] + pe_size)
                (offset(b))
            ENDIF}):
        ¬ leaf ⊃
        (∀ (x1: Physical_memory):
          every[Physical_memory, [bool, Address]]
            (λ (x: Physical_memory): TRUE,
              λ (t: [bool, Address]):
                Mem?(t'2'type_of) ∧ 0 ≤ t'2'offset ∧ t'2'offset < max_linear_offset)
            (translate[Physical_memory, pm_phy]
              (1, base, addr', access', segment_to_priv(cs))(x1)))
{-11} pm' mem = linear_pm
{-12} ∀ (s: (pm' states)): OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-13} ∀ (s1, s2: (pm' states)):
      data(read_data(pm_phy, segment_reg_data_type)(CS)(s1)) =
        data(read_data(pm_phy, segment_reg_data_type)(CS)(s2))
{-14} ∀ (s: (pm' states)): OK?(read_data(pm_phy, pdir_data_type)(PDBR)(s))
{-15} ∀ (s1, s2: (pm' states)):
      data(read_data(pm_phy, pdir_data_type)(PDBR)(s1)) =
        data(read_data(pm_phy, pdir_data_type)(PDBR)(s2))
{-16} ∀ (s1, s2: (pm' states)):
      pe_in_pdir_range?(s1,
        restrict[Address, Memory_Address_4G, boolean]
          ((pm'ro_addr ∪ pm'rw_addr)))
      =
      pe_in_pdir_range?(s2,
        restrict[Address, Memory_Address_4G, boolean]
          ((pm'ro_addr ∪ pm'rw_addr)))
      ∧
      pe_in_ptab_range?(s1,

```

C Proof scripts

which is trivially true.

This completes the proof of `linear_resolve_unchanged_pm_phy.1.1.3.3.3.1.1.3.1.`

linear_resolve_unchanged_pm_phy.1.1.3.3.3.1.1.3.2:

```

{-1} pm' states(s'')
{-2} (addresses ⊆ ((pm_phy'ro_addr ∪ pm_phy'rw_addr) \ address_block(xlat_idx(1, d'''2, addr'), expt(2, 2))))
{-3} Mem?(xlat_idx(1, d'''2, addr')'type_of)
{-4} xlat_idx(1, d'''2, addr')'offset < max_linear_offset
{-5} aligned?(12)(offset(d'''2))
{-6} ∀ (s: (pm_phy' states)):
      OK?(translate(0, d''base_addr, addr', access', segment_to_priv(d'))(s) ∧
          d''' = data(translate(0, d''base_addr, addr', access', segment_to_priv(d'))(s))
{-7} ∀ (s: (pm_phy' states)):
      OK?(read_data(pm_phy, pdb_data_type)(PDBR)(s) ∧
          data(read_data(pm_phy, pdb_data_type)(PDBR)(s)) = d''
{-8} ∀ (s: (pm' states)):
      OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s) ∧
          d' = data(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-9} ∀ (cs: Segment_Reg_type, pdb: Pdb_type):
      ∀ (x1: Physical_memory):
          every[Physical_memory, [bool, Address]]
            (λ (x: Physical_memory): TRUE,
              λ (t: [bool, Address]):
                Mem?(t'2'type_of) ∧
                (0 ≤ t'2'offset ∧ t'2'offset < max_linear_offset) ∧
                IF t'1
                  THEN aligned?(22)(offset(t'2))
                  ELSE aligned?(12)(offset(t'2))
                ENDIF)
            (translate[Physical_memory, pm_phy]
              (0, pdb'base_addr, addr', access', segment_to_priv(cs))(x1))
{-10} ∀ (cs: Segment_Reg_type, pdb: Pdb_type, leaf: bool,
        base:
          {b: Memory_Address_4G |
            IF leaf
              THEN aligned?(-1 × (pdir_lvl [Physical_memory, pm_phy] × bits_per_level [Physical
                (offset(b))
              ELSE aligned?(bits_per_level[Physical_memory, pm_phy] + pe_size)
                (offset(b))
            ENDIF}):
        ¬ leaf ⊃
        (∀ (x1: Physical_memory):
          every[Physical_memory, [bool, Address]]
            (λ (x: Physical_memory): TRUE,
              λ (t: [bool, Address]):
                Mem?(t'2'type_of) ∧ 0 ≤ t'2'offset ∧ t'2'offset < max_linear_offset)
            (translate[Physical_memory, pm_phy]
              (1, base, addr', access', segment_to_priv(cs))(x1)))
{-11} pm' mem = linear_pm
{-12} ∀ (s: (pm' states)): OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-13} ∀ (s1, s2: (pm' states)):
      data(read_data(pm_phy, segment_reg_data_type)(CS)(s1)) =
        data(read_data(pm_phy, segment_reg_data_type)(CS)(s2))
{-14} ∀ (s: (pm' states)): OK?(read_data(pm_phy, pdb_data_type)(PDBR)(s))
{-15} ∀ (s1, s2: (pm' states)):
      data(read_data(pm_phy, pdb_data_type)(PDBR)(s1)) =
        data(read_data(pm_phy, pdb_data_type)(PDBR)(s2))
{-16} ∀ (s1, s2: (pm' states)):
      pe_in_pdir_range?(s1,
        restrict[Address, Memory_Address_4G, boolean]
          ((pm'ro_addr ∪ pm'rw_addr)))
      =
      pe_in_pdir_range?(s2,
        restrict[Address, Memory_Address_4G, boolean]
          ((pm'ro_addr ∪ pm'rw_addr)))
      ∧
      pe_in_ptab_range?(s1,

```

C Proof scripts

Simplifying, rewriting, and recording with decision procedures,

Using lemma `xlat_idx_memory_address2`,

Simplifying, rewriting, and recording with decision procedures,

Expanding the definition of `pe_in_pt_range?`,

Simplifying, rewriting, and recording with decision procedures,

Using lemma `xlat_pe_in_pt_range`,

Instantiating quantified variables,

Case splitting on `pm!1'states(state(read_data(pm_phy, pabr_data_type)(PDBR)(s!2)))`,

we get 3 subgoals:

linear_resolve_unchanged_pm_phy.1.1.3.3.3.1.1.3.2.1:

```

{-1} pm' states(state(read_data(pm_phy, pdir_data_type)(PDBR)(s'')))
{-2} union(pm' ro_addr, pm' rw_addr)(addr') ^
      is_linear_plain_memory?(pm') ^
      pm' states(s'') ^
      pm' states(s'') ^
      OK?(translate(pdir_lvl, data(read_data(pm_phy, pdir_data_type)(PDBR)(s''))'base_addr,
                    addr', access', segment_to_priv(d'))
            (state(read_data(pm_phy, pdir_data_type)(PDBR)(s''))))
      ^
      ¬ data(translate(pdir_lvl,
                      data(read_data(pm_phy, pdir_data_type)(PDBR)(s''))'base_addr,
                      addr', access', segment_to_priv(d'))
            (state(read_data(pm_phy, pdir_data_type)(PDBR)(s''))))'1
      ⊃
      pe_in_pt_range?(s'',
                    restrict[Address, Memory_Address_4G, boolean]
                      ((pm' ro_addr ∪ pm' rw_addr)),
                      ptab_lvl)
                    (xlat_idx(ptab_lvl,
                              data(translate(pdir_lvl,
                                              data(read_data(pm_phy, pdir_data_type)
                                                    (PDBR)
                                                    (s''))'base_addr,
                                              addr', access', seg-
ment_to_priv(d'))
                              (state(read_data(pm_phy, pdir_data_type)
                                      (PDBR)
                                      (s''))))'2,
                              addr'))
{-3} 0 ≤ offset(xlat_idx(1, d'''2, addr'))
{-4} pm' states(s'')
{-5} (addresses ⊆ ((pm_phy'ro_addr ∪ pm_phy'rw_addr) \ address_block(xlat_idx(1, d'''2, addr'), expt(2, 2))))
{-6} Mem?(xlat_idx(1, d'''2, addr'))'type_of
{-7} xlat_idx(1, d'''2, addr'))'offset < max_linear_offset
{-8} aligned?(12)(offset(d'''2))
{-9} ∀ (s: (pm_phy'states)):
      OK?(translate(0, d'''base_addr, addr', access', segment_to_priv(d''))(s)) ^
      d''' = data(translate(0, d'''base_addr, addr', access', segment_to_priv(d''))(s))
{-10} OK?(read_data(pm_phy, pdir_data_type)(PDBR)(s'')) ^
      data(read_data(pm_phy, pdir_data_type)(PDBR)(s'')) = d''
{-11} ∀ (s: (pm' states)):
      OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s)) ^
      d' = data(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-12} ∀ (cs: Segment_Reg_type, pdir: Pdir_type):
      ∀ (x1: Physical_memory):
        every[Physical_memory, [bool, Address]]
          (λ (x: Physical_memory): TRUE,
           λ (t: [bool, Address]):
             Mem?(t'2'type_of) ^
             (0 ≤ t'2'offset ^ t'2'offset < max_linear_offset) ^
             IF t'1
               THEN aligned?(22)(offset(t'2))
               ELSE aligned?(12)(offset(t'2))
             ENDIF)
          (translate[Physical_memory, pm_phy]
            (0, pdir'base_addr, addr', access', segment_to_priv(cs))(x1))1839
{-13} ∀ (cs: Segment_Reg_type, pdir: Pdir_type, leaf: bool,
      base:
        {b: Memory_Address_4G |
          IF leaf
            THEN aligned?(−1 × (pdir_lvl [Physical_memory, pm_phy] × bits_per_level [Physical
            (offset(b))
            ELSE aligned?(bits_per_level[Physical_memory, pm_phy] + pe_size)

```


C Proof scripts

Instantiating quantified variables,

we get 3 subgoals:

linear_resolve_unchanged_pm_phy.1.1.3.3.3.1.1.3.2.1.1:

```

{-1} pm' states(state(read_data(pm_phy, pabr_data_type)(PDBR)(s'')))
{-2} union(pm' ro_addr, pm' rw_addr)(addr') ∧
      is_linear_plain_memory?(pm') ∧
      pm' states(s'') ∧
      pm' states(s'') ∧
      OK?(translate(pdir_lvl, data(read_data(pm_phy, pabr_data_type)(PDBR)(s''))'base_addr,
                    addr', access', segment_to_priv(d'))
            (state(read_data(pm_phy, pabr_data_type)(PDBR)(s''))))
      ∧
      ¬ data(translate(pdir_lvl,
                      data(read_data(pm_phy, pabr_data_type)(PDBR)(s''))'base_addr,
                      addr', access', segment_to_priv(d'))
            (state(read_data(pm_phy, pabr_data_type)(PDBR)(s''))))'1
      ⊃
      pe_in_pt_range?(s'',
                      restrict[Address, Memory_Address_4G, boolean]
                        ((pm' ro_addr ∪ pm' rw_addr),
                         ptab_lvl)
                      (xlat_idx(ptab_lvl,
                                data(translate(pdir_lvl,
                                                data(read_data(pm_phy, pabr_data_type)
                                                              (PDBR)
                                                              (s''))'base_addr,
                                                              addr', access', seg-
ment_to_priv(d'))
                                (state(read_data(pm_phy, pabr_data_type)
                                              (PDBR)
                                              (s''))))'2,
                                addr'))
                      0 ≤ offset(xlat_idx(1, d'''2, addr'))
{-3} pm' states(s'')
{-4} (addresses ⊆ ((pm_phy ro_addr ∪ pm_phy rw_addr) \ address_block(xlat_idx(1, d'''2, addr'), expt(2, 2))))
{-5} Mem?(xlat_idx(1, d'''2, addr'))'type_of
{-6} xlat_idx(1, d'''2, addr'))'offset < max_linear_offset
{-7} aligned?(12)(offset(d'''2))
{-8} OK?(translate(0, d''base_addr, addr', access', segment_to_priv(d'))
                (state(read_data(pm_phy, pabr_data_type)(PDBR)(s''))))
      ∧
      d''' =
      data(translate(0, d''base_addr, addr', access', segment_to_priv(d'))
            (state(read_data(pm_phy, pabr_data_type)(PDBR)(s''))))
{-10} OK?(read_data(pm_phy, pabr_data_type)(PDBR)(s'')) ∧
      data(read_data(pm_phy, pabr_data_type)(PDBR)(s'')) = d''
{-11} ∀ (s: (pm' states)):
      OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s)) ∧
      d' = data(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-12} ∀ (cs: Segment_Reg_type, pabr: Pabr_type):
      ∀ (x1: Physical_memory):
      every[Physical_memory, [bool, Address]]
        (λ (x: Physical_memory): TRUE,
         λ (t: [bool, Address]):
         Mem?(t'2'type_of) ∧
         (0 ≤ t'2'offset ∧ t'2'offset < max_linear_offset) ∧
         IF t'1
           THEN aligned?(22)(offset(t'2))
           ELSE aligned?(12)(offset(t'2))
         ENDIF
        (translate[Physical_memory, pm_phy]
          (0, pabr'base_addr, addr', access', segment_to_priv(cs))(x1))
{-13} ∀ (cs: Segment_Reg_type, pabr: Pabr_type, leaf: bool,
      base:
      {b: Memory_Address_4G |
      IF leaf

```

C Proof scripts

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `linear_resolve_unchanged_pm_phy.1.1.3.3.3.1.1.3.2.1.1.`

linear_resolve_unchanged_pm_phy.1.1.3.3.3.1.1.3.2.1.2:

```

{-1} pm' states(state(read_data(pm_phy, pdir_data_type)(PDBR)(s'')))
{-2} union(pm' ro_addr, pm' rw_addr)(addr') ∧
      is_linear_plain_memory?(pm') ∧
      pm' states(s'') ∧
      pm' states(s'') ∧
      OK?(translate(pdir_lvl, data(read_data(pm_phy, pdir_data_type)(PDBR)(s''))'base_addr,
                    addr', access', segment_to_priv(d'))
            (state(read_data(pm_phy, pdir_data_type)(PDBR)(s''))))
      ∧
      ¬ data(translate(pdir_lvl,
                      data(read_data(pm_phy, pdir_data_type)(PDBR)(s''))'base_addr,
                      addr', access', segment_to_priv(d'))
            (state(read_data(pm_phy, pdir_data_type)(PDBR)(s''))))'1
      ⊃
      pe_in_pt_range?(s'',
                    restrict[Address, Memory_Address_4G, boolean]
                      ((pm' ro_addr ∪ pm' rw_addr),
                      ptab_lvl)
                    (xlat_idx(ptab_lvl,
                              data(translate(pdir_lvl,
                                              data(read_data(pm_phy, pdir_data_type)
                                                    (PDBR)
                                                    (s''))'base_addr,
                                              addr', access', seg-
ment_to_priv(d'))
                              (state(read_data(pm_phy, pdir_data_type)
                                        (PDBR)
                                        (s''))))'2,
                              addr'))
{-3} 0 ≤ offset(xlat_idx(1, d'''2, addr'))
{-4} pm' states(s'')
{-5} (addresses ⊆ ((pm_phy'ro_addr ∪ pm_phy'rw_addr) \ address_block(xlat_idx(1, d'''2, addr'), expt(2, 2))))
{-6} Mem?(xlat_idx(1, d'''2, addr'))'type_of
{-7} xlat_idx(1, d'''2, addr'))'offset < max_linear_offset
{-8} aligned?(12)(offset(d'''2))
{-9} OK?(read_data(pm_phy, pdir_data_type)(PDBR)(s'')) ∧
      data(read_data(pm_phy, pdir_data_type)(PDBR)(s'')) = d''
{-10} ∀ (s: (pm' states)):
      OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s)) ∧
      d' = data(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-11} ∀ (cs: Segment_Reg_type, pdir: Pdir_type):
      ∀ (x1: Physical_memory):
      every[Physical_memory, [bool, Address]]
        (λ (x: Physical_memory): TRUE,
         λ (t: [bool, Address]):
           Mem?(t'2'type_of) ∧
           (0 ≤ t'2'offset ∧ t'2'offset < max_linear_offset) ∧
           IF t'1
             THEN aligned?(22)(offset(t'2))
             ELSE aligned?(12)(offset(t'2))
           ENDIF)
        (translate[Physical_memory, pm_phy]
          (0, pdir'base_addr, addr', access', segment_to_priv(cs))(x1))
{-12} ∀ (cs: Segment_Reg_type, pdir: Pdir_type, leaf: bool,
      base:
        {b: Memory_Address_4G |
          IF leaf
            THEN aligned?(−1 × (pdir_lvl [Physical_memory, pm_phy] × bits_per_level [Physical
              (offset(b))
            ELSE aligned?(bits_per_level[Physical_memory, pm_phy] + pe_size)
              (offset(b))
            ENDIF}):
      ¬ leaf ⊃

```

C Proof scripts

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `linear_resolve_unchanged_pm_phy.1.1.3.3.3.1.1.3.2.1.2`.

linear_resolve_unchanged_pm_phy.1.1.3.3.3.1.1.3.2.1.3:

```

{-1} pm' states(state(read_data(pm_phy, pabr_data_type)(PDBR)(s'')))
{-2} union(pm' ro_addr, pm' rw_addr)(addr') ^
      is_linear_plain_memory?(pm') ^
      pm' states(s'') ^
      pm' states(s'') ^
      OK?(translate(pdir_lvl, data(read_data(pm_phy, pabr_data_type)(PDBR)(s''))'base_addr,
                    addr', access', segment_to_priv(d'))
            (state(read_data(pm_phy, pabr_data_type)(PDBR)(s''))))
      ^
      ¬ data(translate(pdir_lvl,
                       data(read_data(pm_phy, pabr_data_type)(PDBR)(s''))'base_addr,
                       addr', access', segment_to_priv(d'))
              (state(read_data(pm_phy, pabr_data_type)(PDBR)(s''))))'1
      ⊃
      pe_in_pt_range?(s'',
                      restrict[Address, Memory_Address_4G, boolean]
                        ((pm' ro_addr ∪ pm' rw_addr),
                         ptab_lvl)
                      (xlat_idx(ptab_lvl,
                                data(translate(pdir_lvl,
                                                data(read_data(pm_phy, pabr_data_type)
                                                            (PDBR)
                                                            (s''))'base_addr,
                                                            addr', access', seg-
ment_to_priv(d'))
                                (state(read_data(pm_phy, pabr_data_type)
                                            (PDBR)
                                            (s''))))'2,
                                addr'))
{-3} 0 ≤ offset(xlat_idx(1, d'''2, addr'))
{-4} pm' states(s'')
{-5} (addresses ⊆ ((pm_phy'ro_addr ∪ pm_phy'rw_addr) \ address_block(xlat_idx(1, d'''2, addr'), expt(2, 2))))
{-6} Mem?(xlat_idx(1, d'''2, addr'))'type_of
{-7} xlat_idx(1, d'''2, addr'))'offset < max_linear_offset
{-8} aligned?(12)(offset(d'''2))
{-9} ∀ (s: (pm_phy'states)):
      OK?(translate(0, d'''base_addr, addr', access', segment_to_priv(d'))(s)) ^
      d''' = data(translate(0, d'''base_addr, addr', access', segment_to_priv(d'))(s))
{-10} OK?(read_data(pm_phy, pabr_data_type)(PDBR)(s'')) ^
      data(read_data(pm_phy, pabr_data_type)(PDBR)(s'')) = d''
{-11} ∀ (s: (pm' states)):
      OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s)) ^
      d' = data(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-12} ∀ (cs: Segment_Reg_type, pabr: Pabr_type):
      ∀ (x1: Physical_memory):
        every[Physical_memory, [bool, Address]]
          (λ (x: Physical_memory): TRUE,
           λ (t: [bool, Address]):
             Mem?(t'2'type_of) ^
             (0 ≤ t'2'offset ^ t'2'offset < max_linear_offset) ^
             IF t'1
               THEN aligned?(22)(offset(t'2))
               ELSE aligned?(12)(offset(t'2))
             ENDIF)
          (translate[Physical_memory, pm_phy]
           (0, pabr'base_addr, addr', access', segment_to_priv(cs))(x1))1845
{-13} ∀ (cs: Segment_Reg_type, pabr: Pabr_type, leaf: bool,
        base:
          {b: Memory_Address_4G |
            IF leaf
              THEN aligned?(-1 × (pdir_lvl [Physical_memory, pm_phy] × bits_per_level [Physical
              (offset(b))
            ELSE aligned?(bits_per_level[Physical_memory, pm_phy] + pe_size)

```

C Proof scripts

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `linear_resolve_unchanged_pm_phy.1.1.3.3.3.1.1.3.2.1.3`.

linear_resolve_unchanged_pm_phy.1.1.3.3.3.1.1.3.2.2:

```

{-1} union(pm'ro_addr, pm'rw_addr)(addr') ∧
      is_linear_plain_memory?(pm') ∧
      pm'states(s'') ∧
      pm'states(s'') ∧
      OK?(translate(pdir_lvl, data(read_data(pm_phy, pdir_data_type)(PDBR)(s''))'base_addr,
                    addr', access', segment_to_priv(d'))
              (state(read_data(pm_phy, pdir_data_type)(PDBR)(s''))))
      ∧
      ¬ data(translate(pdir_lvl,
                      data(read_data(pm_phy, pdir_data_type)(PDBR)(s''))'base_addr,
                      addr', access', segment_to_priv(d'))
              (state(read_data(pm_phy, pdir_data_type)(PDBR)(s''))))'1
      ⊃
      pe_in_pt_range?(s'',
                      restrict[Address, Memory_Address_4G, boolean]
                        ((pm'ro_addr ∪ pm'rw_addr),
                         ptab_lvl)
                      (xlat_idx(ptab_lvl,
                                data(translate(pdir_lvl,
                                                data(read_data(pm_phy, pdir_data_type)
                                                          (PDBR)
                                                          (s''))'base_addr,
                                                          addr', access', seg-
ment_to_priv(d'))
                                (state(read_data(pm_phy, pdir_data_type)
                                              (PDBR)
                                              (s''))))'2,
                                addr'))
{-2} 0 ≤ offset(xlat_idx(1, d'''2, addr'))
{-3} pm'states(s'')
{-4} (addresses ⊆ ((pm_phy'ro_addr ∪ pm_phy'rw_addr) \ address_block(xlat_idx(1, d'''2, addr'), expt(2, 2))))
{-5} Mem?(xlat_idx(1, d'''2, addr'))'type_of
{-6} xlat_idx(1, d'''2, addr'))'offset < max_linear_offset
{-7} aligned?(12)(offset(d'''2))
{-8} ∀ (s: (pm_phy'states)):
      OK?(translate(0, d''base_addr, addr', access', segment_to_priv(d''))(s)) ∧
      d''' = data(translate(0, d''base_addr, addr', access', segment_to_priv(d''))(s))
{-9} OK?(read_data(pm_phy, pdir_data_type)(PDBR)(s'')) ∧
      data(read_data(pm_phy, pdir_data_type)(PDBR)(s'')) = d''
{-10} ∀ (s: (pm'states)):
      OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s)) ∧
      d' = data(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-11} ∀ (cs: Segment_Reg_type, pdir: Pdir_type):
      ∀ (x1: Physical_memory):
        every[Physical_memory, [bool, Address]]
          (λ (x: Physical_memory): TRUE,
           λ (t: [bool, Address]):
             Mem?(t'2'type_of) ∧
             (0 ≤ t'2'offset ∧ t'2'offset < max_linear_offset) ∧
             IF t'1
               THEN aligned?(22)(offset(t'2))
               ELSE aligned?(12)(offset(t'2))
             ENDIF
          (translate[Physical_memory, pm_phy]
            (0, pdir'base_addr, addr', access', segment_to_priv(cs))(x1))
{-12} ∀ (cs: Segment_Reg_type, pdir: Pdir_type, leaf: bool,
        base:
          {b: Memory_Address_4G |
            IF leaf
              THEN aligned?(−1 × (pdir_lvl [Physical_memory, pm_phy] × bits_per_level [Physical
                (offset(b))
              ELSE aligned?(bits_per_level[Physical_memory, pm_phy] + pe_size)
                (offset(b))

```


C Proof scripts

Using lemma `pm_unchanged_singleton_linear_resolve_reg_transformers`,

Expanding the definition of `linear_resolve_register_transformers`,

Using lemma `unchanged_memory_invariant_invariant`,

Using lemma `expr_transformer_invariant_next_ok`,

Using lemma `pm_states`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `linear_resolve_unchanged_pm_phy.1.1.3.3.3.1.1.3.2.2`.

linear_resolve_unchanged_pm_phy.1.1.3.3.3.1.1.3.2.3:

```

{-1}  union(pm'ro_addr, pm'rw_addr)(addr') ∧
      is_linear_plain_memory?(pm') ∧
      pm'states(s'') ∧
      pm'states(s'') ∧
      OK?(translate(pdir_lvl, data(read_data(pm_phy, pdir_data_type)(PDBR)(s''))'base_addr,
                    addr', access', segment_to_priv(d'))
              (state(read_data(pm_phy, pdir_data_type)(PDBR)(s''))))
      ∧
      ¬ data(translate(pdir_lvl,
                      data(read_data(pm_phy, pdir_data_type)(PDBR)(s''))'base_addr,
                      addr', access', segment_to_priv(d'))
              (state(read_data(pm_phy, pdir_data_type)(PDBR)(s''))))'1
      ⊃
      pe_in_pt_range?(s'',
                      restrict[Address, Memory_Address_4G, boolean]
                        ((pm'ro_addr ∪ pm'rw_addr)),
                      ptab_lvl)
      (xlat_idx(ptab_lvl,
                data(translate(pdir_lvl,
                              data(read_data(pm_phy, pdir_data_type)
                                      (PDBR)
                                      (s''))'base_addr,
                              addr', access', seg-
ment_to_priv(d'))
                              (state(read_data(pm_phy, pdir_data_type)
                                      (PDBR)
                                      (s''))))'2,
                              addr'))
{-2}  0 ≤ offset(xlat_idx(1, d'''2, addr'))
{-3}  pm'states(s'')
{-4}  (addresses ⊆ ((pm_phy'ro_addr ∪ pm_phy'rw_addr) \ address_block(xlat_idx(1, d'''2, addr'), expt(2, 2))))
{-5}  Mem?(xlat_idx(1, d'''2, addr'))'type_of
{-6}  xlat_idx(1, d'''2, addr'))'offset < max_linear_offset
{-7}  aligned?(12)(offset(d'''2))
{-8}  ∀ (s: (pm_phy'states)):
      OK?(translate(0, d''base_addr, addr', access', segment_to_priv(d''))(s)) ∧
      d''' = data(translate(0, d''base_addr, addr', access', segment_to_priv(d''))(s))
{-9}  OK?(read_data(pm_phy, pdir_data_type)(PDBR)(s'')) ∧
      data(read_data(pm_phy, pdir_data_type)(PDBR)(s'')) = d''
{-10} ∀ (s: (pm'states)):
      OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s)) ∧
      d' = data(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-11} ∀ (cs: Segment_Reg_type, pdir: Pdir_type):
      ∀ (x1: Physical_memory):
        every[Physical_memory, [bool, Address]]
          (λ (x: Physical_memory): TRUE,
           λ (t: [bool, Address]):
             Mem?(t'2'type_of) ∧
             (0 ≤ t'2'offset ∧ t'2'offset < max_linear_offset) ∧
             IF t'1
               THEN aligned?(22)(offset(t'2))
               ELSE aligned?(12)(offset(t'2))
             ENDIF
          (translate[Physical_memory, pm_phy]
            (0, pdir'base_addr, addr', access', segment_to_priv(cs))(x1))
{-12} ∀ (cs: Segment_Reg_type, pdir: Pdir_type, leaf: bool,
        base:
          {b: Memory_Address_4G |
            IF leaf
              THEN aligned?(−1 × (pdir_lvl [Physical_memory, pm_phy] × bits_per_level [Physical
                (offset(b))
              ELSE aligned?(bits_per_level[Physical_memory, pm_phy] + pe_size)
                (offset(b))

```

C Proof scripts

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `linear_resolve_unchanged_pm_phy.1.1.3.3.3.1.1.3.2.3`.

linear_resolve_unchanged_pm_phy.1.1.3.3.3.1.1.4:

```

{-1}  (addresses ⊆ ((pm_phy'ro_addr ∪ pm_phy'rw_addr) \ address_block(xlat_idx(1, d'''2, addr'), expt(2, 2))))
{-2}  Mem?(xlat_idx(1, d'''2, addr')'type_of)
{-3}  xlat_idx(1, d'''2, addr')'offset < max_linear_offset
{-4}  aligned?(12)(offset(d'''2))
{-5}  ∀ (s: (pm_phy'states)):
      OK?(translate(0, d''base_addr, addr', access', segment_to_priv(d'))(s) ∧
          d''' = data(translate(0, d''base_addr, addr', access', segment_to_priv(d'))(s))
{-6}  ∀ (s: (pm_phy'states)):
      OK?(read_data(pm_phy, pdir_data_type)(PDBR)(s) ∧
          data(read_data(pm_phy, pdir_data_type)(PDBR)(s)) = d''
{-7}  ∀ (s: (pm' states)):
      OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s) ∧
          d' = data(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-8}  ∀ (cs: Segment_Reg_type, pdir: Pdir_type):
      ∀ (x1: Physical_memory):
          every[Physical_memory, [bool, Address]]
            (λ (x: Physical_memory): TRUE,
              λ (t: [bool, Address]):
                Mem?(t'2'type_of) ∧
                (0 ≤ t'2'offset ∧ t'2'offset < max_linear_offset) ∧
                IF t'1
                  THEN aligned?(22)(offset(t'2))
                  ELSE aligned?(12)(offset(t'2))
                ENDIF)
            (translate[Physical_memory, pm_phy]
              (0, pdir'base_addr, addr', access', segment_to_priv(cs))(x1))
{-9}  ∀ (cs: Segment_Reg_type, pdir: Pdir_type, leaf: bool,
      base:
        {b: Memory_Address_4G |
          IF leaf
            THEN aligned?(-1 × (pdir_lvl [Physical_memory, pm_phy] × bits_per_level [Physical
              (offset(b))
            ELSE aligned?(bits_per_level[Physical_memory, pm_phy] + pe_size)
              (offset(b))
            ENDIF}):
        ¬ leaf ⊃
      (∀ (x1: Physical_memory):
        every[Physical_memory, [bool, Address]]
          (λ (x: Physical_memory): TRUE,
            λ (t: [bool, Address]):
              Mem?(t'2'type_of) ∧ 0 ≤ t'2'offset ∧ t'2'offset < max_linear_offset)
          (translate[Physical_memory, pm_phy]
            (1, base, addr', access', segment_to_priv(cs))(x1)))
{-10} pm'mem = linear_pm
{-11} ∀ (s: (pm' states)): OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-12} ∀ (s1, s2: (pm' states)):
      data(read_data(pm_phy, segment_reg_data_type)(CS)(s1)) =
        data(read_data(pm_phy, segment_reg_data_type)(CS)(s2))
{-13} ∀ (s: (pm' states)): OK?(read_data(pm_phy, pdir_data_type)(PDBR)(s))
{-14} ∀ (s1, s2: (pm' states)):
      data(read_data(pm_phy, pdir_data_type)(PDBR)(s1)) =
        data(read_data(pm_phy, pdir_data_type)(PDBR)(s2))
{-15} ∀ (s1, s2: (pm' states)):
      pe_in_pdir_range?(s1,
        restrict[Address, Memory_Address_4G, boolean]
          ((pm'ro_addr ∪ pm'rw_addr)))
      =
      pe_in_pdir_range?(s2,
        restrict[Address, Memory_Address_4G, boolean]
          ((pm'ro_addr ∪ pm'rw_addr)))
      ∧
      pe_in_ptab_range?(s1,
        restrict[Address, Memory_Address_4G, boolean]
          ((pm'ro_addr ∪ pm'rw_addr)))

```

C Proof scripts

Hiding formulas: 2, 3,

Using lemma `subset_ptab_address_rw_addr`,

we get 2 subgoals:

linear_resolve_unchanged_pm_phy.1.1.3.3.3.1.1.4.1:

- {-1} (address_in_pt_range?(s', (restrict [Address, Memory_Address_4G, bool](pm'ro_addr) ∪ restrict [Address,
 \wedge
 (address_block(xlat_idx(1, d'''2, addr'), expt(2, pe_size)) \subseteq extend [Address, Memory_Address_4G, bo
 \supseteq (address_block(xlat_idx(1, d'''2, addr'), expt(2, pe_size)) \subseteq pm_phy'rw_addr)
- {-2} (addresses \subseteq ((pm_phy'ro_addr ∪ pm_phy'rw_addr) \ address_block(xlat_idx(1, d'''2, addr'), expt(2, 2))))
- {-3} Mem?(xlat_idx(1, d'''2, addr')'type_of)
- {-4} xlat_idx(1, d'''2, addr')'offset < max_linear_offset
- {-5} aligned?(12)(offset(d'''2))
- {-6} $\forall (s: (pm_phy'states)):$
 - OK?(translate(0, d'''base_addr, addr', access', segment_to_priv(d'''))(s) \wedge
 - d''' = data(translate(0, d'''base_addr, addr', access', segment_to_priv(d'''))(s))
- {-7} $\forall (s: (pm_phy'states)):$
 - OK?(read_data(pm_phy, pdbr_data_type)(PDBR)(s) \wedge
 - data(read_data(pm_phy, pdbr_data_type)(PDBR)(s)) = d'''
- {-8} $\forall (s: (pm'states)):$
 - OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s) \wedge
 - d' = data(read_data(pm_phy, segment_reg_data_type)(CS)(s))
- {-9} $\forall (cs: Segment_Reg_type, pdbr: Pdbr_type):$
 - $\forall (x_1: Physical_memory):$
 - every[Physical_memory, [bool, Address]]
 - ($\lambda (x: Physical_memory): TRUE,$
 - $\lambda (t: [bool, Address]):$
 - Mem?(t'2'type_of) \wedge
 - (0 \leq t'2'offset \wedge t'2'offset < max_linear_offset) \wedge
 - IF t'1
 - THEN aligned?(22)(offset(t'2))
 - ELSE aligned?(12)(offset(t'2))
 - ENDIF)
 - (translate[Physical_memory, pm_phy]
 - (0, pdbr'base_addr, addr', access', segment_to_priv(cs))(x_1))
- {-10} $\forall (cs: Segment_Reg_type, pdbr: Pdbr_type, leaf: bool,$
 - base:
 - {b: Memory_Address_4G |
 - IF leaf
 - THEN aligned?(-1 \times (pdir_lvl [Physical_memory, pm_phy] \times bits_per_level [Physical
 - (offset(b))
 - ELSE aligned?(bits_per_level[Physical_memory, pm_phy] + pe_size)
 - (offset(b))
 - ENDIF}):
 - \neg leaf \supseteq
 - ($\forall (x_1: Physical_memory):$
 - every[Physical_memory, [bool, Address]]
 - ($\lambda (x: Physical_memory): TRUE,$
 - $\lambda (t: [bool, Address]):$
 - Mem?(t'2'type_of) \wedge 0 \leq t'2'offset \wedge t'2'offset < max_linear_offset)
 - (translate[Physical_memory, pm_phy]
 - (1, base, addr', access', segment_to_priv(cs))(x_1)))
- {-11} pm'mem = linear_pm
- {-12} $\forall (s: (pm'states)):$ OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s))
- {-13} $\forall (s_1, s_2: (pm'states)):$
 - data(read_data(pm_phy, segment_reg_data_type)(CS)(s_1)) =
 - data(read_data(pm_phy, segment_reg_data_type)(CS)(s_2))
- {-14} $\forall (s: (pm'states)):$ OK?(read_data(pm_phy, pdbr_data_type)(PDBR)(s))
- {-15} $\forall (s_1, s_2: (pm'states)):$
 - data(read_data(pm_phy, pdbr_data_type)(PDBR)(s_1)) =
 - data(read_data(pm_phy, pdbr_data_type)(PDBR)(s_2))
- {-16} $\forall (s_1, s_2: (pm'states)):$
 - pe_in_pdir_range?(s_1,
 - restrict[Address, Memory_Address_4G, boolean]
 - ((pm'ro_addr ∪ pm'rw_addr)))
 - =
 - pe_in_pdir_range?(s_2,
 - restrict[Address, Memory_Address_4G, boolean]
 - ((pm'ro_addr ∪ pm'rw_addr)))

C Proof scripts

Simplifying, rewriting, and recording with decision procedures,

Applying propositional simplification,

we get 2 subgoals:

linear_resolve_unchanged_pm_phy.1.1.3.3.3.1.1.4.1.1:

```

{-1}  (addresses ⊆ ((pm_phy'ro_addr ∪ pm_phy'rw_addr) \ address_block(xlat_idx(1, d'''2, addr'), expt(2, 2))))
{-2}  Mem?(xlat_idx(1, d'''2, addr')'type_of)
{-3}  xlat_idx(1, d'''2, addr')'offset < max_linear_offset
{-4}  aligned?(12)(offset(d'''2))
{-5}  ∀ (s: (pm_phy'states)):
      OK?(translate(0, d''base_addr, addr', access', segment_to_priv(d'))(s) ∧
          d''' = data(translate(0, d''base_addr, addr', access', segment_to_priv(d'))(s))
{-6}  ∀ (s: (pm_phy'states)):
      OK?(read_data(pm_phy, pdir_data_type)(PDBR)(s) ∧
          data(read_data(pm_phy, pdir_data_type)(PDBR)(s)) = d''
{-7}  ∀ (s: (pm' states)):
      OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s) ∧
          d' = data(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-8}  ∀ (cs: Segment_Reg_type, pdir: Pdir_type):
      ∀ (x1: Physical_memory):
          every[Physical_memory, [bool, Address]]
            (λ (x: Physical_memory): TRUE,
              λ (t: [bool, Address]):
                Mem?(t'2'type_of) ∧
                (0 ≤ t'2'offset ∧ t'2'offset < max_linear_offset) ∧
                IF t'1
                  THEN aligned?(22)(offset(t'2))
                  ELSE aligned?(12)(offset(t'2))
                ENDIF)
            (translate[Physical_memory, pm_phy]
              (0, pdir'base_addr, addr', access', segment_to_priv(cs))(x1))
{-9}  ∀ (cs: Segment_Reg_type, pdir: Pdir_type, leaf: bool,
      base:
        {b: Memory_Address_4G |
          IF leaf
            THEN aligned?(-1 × (pdir_lvl [Physical_memory, pm_phy] × bits_per_level [Physical
              (offset(b))
            ELSE aligned?(bits_per_level[Physical_memory, pm_phy] + pe_size)
              (offset(b))
            ENDIF}):
      ¬ leaf ⊃
      (∀ (x1: Physical_memory):
        every[Physical_memory, [bool, Address]]
          (λ (x: Physical_memory): TRUE,
            λ (t: [bool, Address]):
              Mem?(t'2'type_of) ∧ 0 ≤ t'2'offset ∧ t'2'offset < max_linear_offset)
          (translate[Physical_memory, pm_phy]
            (1, base, addr', access', segment_to_priv(cs))(x1)))
{-10} pm'mem = linear_pm
{-11} ∀ (s: (pm' states)): OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-12} ∀ (s1, s2: (pm' states)):
      data(read_data(pm_phy, segment_reg_data_type)(CS)(s1)) =
      data(read_data(pm_phy, segment_reg_data_type)(CS)(s2))
{-13} ∀ (s: (pm' states)): OK?(read_data(pm_phy, pdir_data_type)(PDBR)(s))
{-14} ∀ (s1, s2: (pm' states)):
      data(read_data(pm_phy, pdir_data_type)(PDBR)(s1)) =
      data(read_data(pm_phy, pdir_data_type)(PDBR)(s2))
{-15} ∀ (s1, s2: (pm' states)):
      pe_in_pdir_range?(s1,
        restrict[Address, Memory_Address_4G, boolean]
          ((pm'ro_addr ∪ pm'rw_addr)))
      =
      pe_in_pdir_range?(s2,
        restrict[Address, Memory_Address_4G, boolean]
          ((pm'ro_addr ∪ pm'rw_addr)))
      ∧
      pe_in_ptab_range?(s1,
        restrict[Address, Memory_Address_4G, boolean]
          ((pm'ro_addr ∪ pm'rw_addr)))

```


C Proof scripts

Using lemma `pm_address_in_pt_range_in_rw_addr`,

Case splitting on `restrict[Address, Memory_Address_4G, boolean] (union(pm!1'ro_addr, pm!1'rw_addr)) = union(restrict[Address, Memory_Address_4G, boolean](pm!1'ro_addr), restrict[Address, Memory_Address_4G, boolean](pm!1'rw_addr))`,

we get 2 subgoals:

linear_resolve_unchanged_pm_phy.1.1.3.3.3.1.1.4.1.1.1:

```

{-1} restrict[Address, Memory_Address_4G, boolean]((pm'ro_addr ∪ pm'rw_addr)) =
      (restrict[Address, Memory_Address_4G, boolean](pm'ro_addr) ∪ restrict [Address, Memory_Address_4G,
{-2} is_linear_plain_memory?(pm') ∧ pm'states(s') ⊃
      (address_in_pt_range?(s', restrict [Address, Memory_Address_4G, boolean]((pm'ro_addr ∪ pm'rw_addr)))
{-3} (addresses ⊆ ((pm_phy'ro_addr ∪ pm_phy'rw_addr) \ address_block(xlat_idx(1, d'''2, addr'), expt(2, 2))))
{-4} Mem?(xlat_idx(1, d'''2, addr')'type_of)
{-5} xlat_idx(1, d'''2, addr')'offset < max_linear_offset
{-6} aligned?(12)(offset(d'''2))
{-7} ∀ (s: (pm_phy'states)):
      OK?(translate(0, d''base_addr, addr', access', segment_to_priv(d'))(s)) ∧
      d''' = data(translate(0, d''base_addr, addr', access', segment_to_priv(d'))(s))
{-8} ∀ (s: (pm_phy'states)):
      OK?(read_data(pm_phy, pdb_data_type)(PDBR)(s)) ∧
      data(read_data(pm_phy, pdb_data_type)(PDBR)(s)) = d''
{-9} ∀ (s: (pm'states)):
      OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s)) ∧
      d' = data(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-10} ∀ (cs: Segment_Reg_type, pdb: Pdb_type):
      ∀ (x1: Physical_memory):
        every[Physical_memory, [bool, Address]]
          (λ (x: Physical_memory): TRUE,
           λ (t: [bool, Address]):
             Mem?(t'2'type_of) ∧
             (0 ≤ t'2'offset ∧ t'2'offset < max_linear_offset) ∧
             IF t'1
               THEN aligned?(22)(offset(t'2))
             ELSE aligned?(12)(offset(t'2))
             ENDIF)
          (translate[Physical_memory, pm_phy]
           (0, pdb'base_addr, addr', access', segment_to_priv(cs))(x1))
{-11} ∀ (cs: Segment_Reg_type, pdb: Pdb_type, leaf: bool,
      base:
        {b: Memory_Address_4G |
          IF leaf
            THEN aligned?(-1 × (pdir_lvl [Physical_memory, pm_phy] × bits_per_level [Physical
              (offset(b))
            ELSE aligned?(bits_per_level[Physical_memory, pm_phy] + pe_size)
              (offset(b))
            ENDIF}):
      ¬ leaf ⊃
      (∀ (x1: Physical_memory):
        every[Physical_memory, [bool, Address]]
          (λ (x: Physical_memory): TRUE,
           λ (t: [bool, Address]):
             Mem?(t'2'type_of) ∧ 0 ≤ t'2'offset ∧ t'2'offset < max_linear_offset)
          (translate[Physical_memory, pm_phy]
           (1, base, addr', access', segment_to_priv(cs))(x1)))
{-12} pm'mem = linear_pm
{-13} ∀ (s: (pm'states)): OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-14} ∀ (s1, s2: (pm'states)):
      data(read_data(pm_phy, segment_reg_data_type)(CS)(s1)) =
      data(read_data(pm_phy, segment_reg_data_type)(CS)(s2))
{-15} ∀ (s: (pm'states)): OK?(read_data(pm_phy, pdb_data_type)(PDBR)(s))
{-16} ∀ (s1, s2: (pm'states)):
      data(read_data(pm_phy, pdb_data_type)(PDBR)(s1)) =
      data(read_data(pm_phy, pdb_data_type)(PDBR)(s2))
{-17} ∀ (s1, s2: (pm'states)):
      pe_in_pdir_range?(s1,
        restrict[Address, Memory_Address_4G, boolean]
          ((pm'ro_addr ∪ pm'rw_addr)))
      =
      pe_in_pdir_range?(s2,
        restrict[Address, Memory_Address_4G, boolean]
          ((pm'ro_addr ∪ pm'rw_addr)))

```

C Proof scripts

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `linear_resolve_unchanged_pm_phy.1.1.3.3.3.1.1.4.1.1.1.`

linear_resolve_unchanged_pm_phy.1.1.3.3.3.1.1.4.1.1.2:

- {-1} is_linear_plain_memory?(pm') \wedge pm' states(s') \supset
 (address_in_pt_range?(s', restrict [Address, Memory_Address_4G, boolean]((pm'ro_addr \cup pm'rw_addr)))
- {-2} (addresses \subseteq ((pm_phy'ro_addr \cup pm_phy'rw_addr) \ address_block(xlat_idx(1, d'''2, addr'), expt(2, 2))))
- {-3} Mem?(xlat_idx(1, d'''2, addr') type_of)
- {-4} xlat_idx(1, d'''2, addr') offset < max_linear_offset
- {-5} aligned?(12)(offset(d'''2))
- {-6} \forall (s: (pm_phy' states)):
 OK?(translate(0, d''base_addr, addr', access', segment_to_priv(d'))(s)) \wedge
 d''' = data(translate(0, d''base_addr, addr', access', segment_to_priv(d'))(s))
- {-7} \forall (s: (pm_phy' states)):
 OK?(read_data(pm_phy, pdb_data_type)(PDBR)(s)) \wedge
 data(read_data(pm_phy, pdb_data_type)(PDBR)(s)) = d''
- {-8} \forall (s: (pm' states)):
 OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s)) \wedge
 d' = data(read_data(pm_phy, segment_reg_data_type)(CS)(s))
- {-9} \forall (cs: Segment_Reg_type, pdb: Pdb_type):
 \forall (x₁: Physical_memory):
 every [Physical_memory, [bool, Address]]
 (λ (x: Physical_memory): TRUE,
 λ (t: [bool, Address]):
 Mem?(t'2 type_of) \wedge
 (0 \leq t'2 offset \wedge t'2 offset < max_linear_offset) \wedge
 IF t'1
 THEN aligned?(22)(offset(t'2))
 ELSE aligned?(12)(offset(t'2))
 ENDIF)
 (translate [Physical_memory, pm_phy]
 (0, pdb'base_addr, addr', access', segment_to_priv(cs))(x₁))
- {-10} \forall (cs: Segment_Reg_type, pdb: Pdb_type, leaf: bool,
 base:
 {b: Memory_Address_4G |
 IF leaf
 THEN aligned?(-1 \times (pdir_lvl [Physical_memory, pm_phy] \times bits_per_level [Physical
 (offset(b))
 ELSE aligned?(bits_per_level [Physical_memory, pm_phy] + pe_size)
 (offset(b))
 ENDIF}):
 \neg leaf \supset
 (\forall (x₁: Physical_memory):
 every [Physical_memory, [bool, Address]]
 (λ (x: Physical_memory): TRUE,
 λ (t: [bool, Address]):
 Mem?(t'2 type_of) \wedge 0 \leq t'2 offset \wedge t'2 offset < max_linear_offset)
 (translate [Physical_memory, pm_phy]
 (1, base, addr', access', segment_to_priv(cs))(x₁)))
- {-11} pm' mem = linear_pm
- {-12} \forall (s: (pm' states)): OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s))
- {-13} \forall (s₁, s₂: (pm' states)):
 data(read_data(pm_phy, segment_reg_data_type)(CS)(s₁)) =
 data(read_data(pm_phy, segment_reg_data_type)(CS)(s₂))
- {-14} \forall (s: (pm' states)): OK?(read_data(pm_phy, pdb_data_type)(PDBR)(s))
- {-15} \forall (s₁, s₂: (pm' states)):
 data(read_data(pm_phy, pdb_data_type)(PDBR)(s₁)) =
 data(read_data(pm_phy, pdb_data_type)(PDBR)(s₂))
- {-16} \forall (s₁, s₂: (pm' states)):
 pe_in_pdir_range?(s₁,
 restrict [Address, Memory_Address_4G, boolean]
 ((pm'ro_addr \cup pm'rw_addr)))
 =
 pe_in_pdir_range?(s₂,
 restrict [Address, Memory_Address_4G, boolean]
 ((pm'ro_addr \cup pm'rw_addr)))
 \wedge

C Proof scripts

Keeping (1) and hiding *,

Applying decompose-equality,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `linear_resolve_unchanged_pm_phy.1.1.3.3.3.1.1.4.1.1.2.`

linear_resolve_unchanged_pm_phy.1.1.3.3.3.1.1.4.1.2:

```

{-1}  (addresses ⊆ ((pm_phy'ro_addr ∪ pm_phy'rw_addr) \ address_block(xlat_idx(1, d'''2, addr'), expt(2, 2))))
{-2}  Mem?(xlat_idx(1, d'''2, addr')'type_of)
{-3}  xlat_idx(1, d'''2, addr')'offset < max_linear_offset
{-4}  aligned?(12)(offset(d'''2))
{-5}  ∀ (s: (pm_phy'states)):
      OK?(translate(0, d'''base_addr, addr', access', segment_to_priv(d'))(s) ∧
          d''' = data(translate(0, d'''base_addr, addr', access', segment_to_priv(d'))(s))
{-6}  ∀ (s: (pm_phy'states)):
      OK?(read_data(pm_phy, pdir_data_type)(PDIR)(s) ∧
          data(read_data(pm_phy, pdir_data_type)(PDIR)(s)) = d''
{-7}  ∀ (s: (pm'states)):
      OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s) ∧
          d' = data(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-8}  ∀ (cs: Segment_Reg_type, pdir: Pdir_type):
      ∀ (x1: Physical_memory):
          every[Physical_memory, [bool, Address]]
            (λ (x: Physical_memory): TRUE,
              λ (t: [bool, Address]):
                Mem?(t'2'type_of) ∧
                (0 ≤ t'2'offset ∧ t'2'offset < max_linear_offset) ∧
                IF t'1
                  THEN aligned?(22)(offset(t'2))
                  ELSE aligned?(12)(offset(t'2))
                ENDIF)
            (translate[Physical_memory, pm_phy]
              (0, pdir'base_addr, addr', access', segment_to_priv(cs))(x1))
{-9}  ∀ (cs: Segment_Reg_type, pdir: Pdir_type, leaf: bool,
      base:
        {b: Memory_Address_4G |
          IF leaf
            THEN aligned?(-1 × (pdir_lvl [Physical_memory, pm_phy] × bits_per_level [Physical
              (offset(b))
            ELSE aligned?(bits_per_level[Physical_memory, pm_phy] + pe_size)
              (offset(b))
            ENDIF}):
        ¬ leaf ⊃
      (∀ (x1: Physical_memory):
        every[Physical_memory, [bool, Address]]
          (λ (x: Physical_memory): TRUE,
            λ (t: [bool, Address]):
              Mem?(t'2'type_of) ∧ 0 ≤ t'2'offset ∧ t'2'offset < max_linear_offset)
          (translate[Physical_memory, pm_phy]
            (1, base, addr', access', segment_to_priv(cs))(x1)))
{-10} pm'mem = linear_pm
{-11} ∀ (s: (pm'states)): OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-12} ∀ (s1, s2: (pm'states)):
      data(read_data(pm_phy, segment_reg_data_type)(CS)(s1)) =
      data(read_data(pm_phy, segment_reg_data_type)(CS)(s2))
{-13} ∀ (s: (pm'states)): OK?(read_data(pm_phy, pdir_data_type)(PDIR)(s))
{-14} ∀ (s1, s2: (pm'states)):
      data(read_data(pm_phy, pdir_data_type)(PDIR)(s1)) =
      data(read_data(pm_phy, pdir_data_type)(PDIR)(s2))
{-15} ∀ (s1, s2: (pm'states)):
      pe_in_pdir_range?(s1,
        restrict[Address, Memory_Address_4G, boolean]
          ((pm'ro_addr ∪ pm'rw_addr)))
      =
      pe_in_pdir_range?(s2,
        restrict[Address, Memory_Address_4G, boolean]
          ((pm'ro_addr ∪ pm'rw_addr)))
      ∧
      pe_in_ptab_range?(s1,
        restrict[Address, Memory_Address_4G, boolean]
          ((pm'ro_addr ∪ pm'rw_addr)))

```

C Proof scripts

Using lemma `pe_in_pt_address_in_pt`,

we get 2 subgoals:

linear_resolve_unchanged_pm_phy.1.1.3.3.3.1.1.4.1.2.1:

```

{-1}  pe_in_pt_range?(s',
      restrict[Address, Memory_Address_4G, bool]
        ((pm'ro_addr ∪ pm'rw_addr),
         1)
      (xlat_idx(1, d'''2, addr'))
    ⊃
    (address_block(xlat_idx(1, d'''2, addr'), expt(2, pe_size)) ⊆ extend [Address, Memory_Address_4G, bo
{-2}  (addresses ⊆ ((pm_phy'ro_addr ∪ pm_phy'rw_addr) \ address_block(xlat_idx(1, d'''2, addr'), expt(2, 2))))
{-3}  Mem?(xlat_idx(1, d'''2, addr')'type_of)
{-4}  xlat_idx(1, d'''2, addr')'offset < max_linear_offset
{-5}  aligned?(12)(offset(d'''2))
{-6}  ∀ (s: (pm_phy'states)):
      OK?(translate(0, d'''base_addr, addr', access', segment_to_priv(d'))(s)) ∧
      d''' = data(translate(0, d'''base_addr, addr', access', segment_to_priv(d'))(s))
{-7}  ∀ (s: (pm_phy'states)):
      OK?(read_data(pm_phy, pabr_data_type)(PDBR)(s)) ∧
      data(read_data(pm_phy, pabr_data_type)(PDBR)(s)) = d''
{-8}  ∀ (s: (pm'states)):
      OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s)) ∧
      d' = data(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-9}  ∀ (cs: Segment_Reg_type, pabr: Pabr_type):
      ∀ (x1: Physical_memory):
        every[Physical_memory, [bool, Address]]
          (λ (x: Physical_memory): TRUE,
           λ (t: [bool, Address]):
             Mem?(t'2'type_of) ∧
             (0 ≤ t'2'offset ∧ t'2'offset < max_linear_offset) ∧
             IF t'1
               THEN aligned?(22)(offset(t'2))
               ELSE aligned?(12)(offset(t'2))
             ENDIF)
          (translate[Physical_memory, pm_phy]
            (0, pabr'base_addr, addr', access', segment_to_priv(cs))(x1))
{-10} ∀ (cs: Segment_Reg_type, pabr: Pabr_type, leaf: bool,
      base:
        {b: Memory_Address_4G |
          IF leaf
            THEN aligned?(-1 × (pdir_lvl [Physical_memory, pm_phy] × bits_per_level [Physical
              (offset(b))
            ELSE aligned?(bits_per_level[Physical_memory, pm_phy] + pe_size)
              (offset(b))
            ENDIF}):
        ¬ leaf ⊃
        (∀ (x1: Physical_memory):
          every[Physical_memory, [bool, Address]]
            (λ (x: Physical_memory): TRUE,
             λ (t: [bool, Address]):
               Mem?(t'2'type_of) ∧ 0 ≤ t'2'offset ∧ t'2'offset < max_linear_offset)
            (translate[Physical_memory, pm_phy]
              (1, base, addr', access', segment_to_priv(cs))(x1)))
{-11} pm'mem = linear_pm
{-12} ∀ (s: (pm'states)): OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-13} ∀ (s1, s2: (pm'states)):
      data(read_data(pm_phy, segment_reg_data_type)(CS)(s1)) =
      data(read_data(pm_phy, segment_reg_data_type)(CS)(s2))
{-14} ∀ (s: (pm'states)): OK?(read_data(pm_phy, pabr_data_type)(PDBR)(s)) 1863
{-15} ∀ (s1, s2: (pm'states)):
      data(read_data(pm_phy, pabr_data_type)(PDBR)(s1)) =
      data(read_data(pm_phy, pabr_data_type)(PDBR)(s2))
{-16} ∀ (s1, s2: (pm'states)):
      pe_in_pdir_range?(s1,
        restrict[Address, Memory_Address_4G, boolean]
          ((pm'ro_addr ∪ pm'rw_addr)))

```


C Proof scripts

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Using lemma `xlat_pe_in_pt_range`,

Instantiating quantified variables,

Case splitting on `pm!1'states(state(read_data(pm_phy, pdbr_data_type)(PDBR)(s!1)))`,

we get 3 subgoals:

linear_resolve_unchanged_pm_phy.1.1.3.3.3.1.1.4.1.2.1.1:

```

{-1} pm' states(state(read_data(pm_phy, pdir_data_type)(PDBR)(s')))
{-2} union(pm' ro_addr, pm' rw_addr)(addr') ∧
      is_linear_plain_memory?(pm') ∧
      pm' states(s') ∧
      pm' states(s') ∧
      OK?(translate(pdir_lvl, data(read_data(pm_phy, pdir_data_type)(PDBR)(s'))'base_addr,
                    addr', access', segment_to_priv(d'))
            (state(read_data(pm_phy, pdir_data_type)(PDBR)(s'))))
      ∧
      ¬ data(translate(pdir_lvl,
                     data(read_data(pm_phy, pdir_data_type)(PDBR)(s'))'base_addr,
                     addr', access', segment_to_priv(d'))
            (state(read_data(pm_phy, pdir_data_type)(PDBR)(s'))))'1
      ⊃
      pe_in_pt_range?(s',
                    restrict[Address, Memory_Address_4G, boolean]
                      ((pm' ro_addr ∪ pm' rw_addr),
                      ptab_lvl)
                    (xlat_idx(ptab_lvl,
                              data(translate(pdir_lvl,
                                             data(read_data(pm_phy, pdir_data_type)
                                                         (PDBR)
                                                         (s'))'base_addr,
                                             addr', access', seg-
ment_to_priv(d'))
                              (state(read_data(pm_phy, pdir_data_type)
                                         (PDBR)
                                         (s'))))'2,
                              addr'))
                    (addresses ⊆ ((pm_phy'ro_addr ∪ pm_phy'rw_addr) \ address_block(xlat_idx(1, d'''2, addr'), expt(2, 2))))
{-3} Mem?(xlat_idx(1, d'''2, addr'))'type_of)
{-4} xlat_idx(1, d'''2, addr'))'offset < max_linear_offset
{-5} aligned?(12)(offset(d'''2))
{-6} ∀ (s: (pm_phy'states)):
      OK?(translate(0, d'''base_addr, addr', access', segment_to_priv(d'))(s)) ∧
      d''' = data(translate(0, d'''base_addr, addr', access', segment_to_priv(d'))(s))
{-7} OK?(read_data(pm_phy, pdir_data_type)(PDBR)(s')) ∧
      data(read_data(pm_phy, pdir_data_type)(PDBR)(s')) = d''
{-8} ∀ (s: (pm' states)):
      OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s)) ∧
      d' = data(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-9} ∀ (cs: Segment_Reg_type, pdir: Pdir_type):
      ∀ (x1: Physical_memory):
        every[Physical_memory, [bool, Address]]
          (λ (x: Physical_memory): TRUE,
           λ (t: [bool, Address]):
             Mem?(t'2'type_of) ∧
             (0 ≤ t'2'offset ∧ t'2'offset < max_linear_offset) ∧
             IF t'1
               THEN aligned?(22)(offset(t'2))
               ELSE aligned?(12)(offset(t'2))
             ENDIF)
          (translate[Physical_memory, pm_phy]
           (0, pdir'base_addr, addr', access', segment_to_priv(cs))(x1))
{-10} ∀ (cs: Segment_Reg_type, pdir: Pdir_type, leaf: bool,
        base:
          {b: Memory_Address_4G |
            IF leaf
              THEN aligned?(-1 × (pdir_lvl [Physical_memory, pm_phy] × bits_per_level [Physical
                (offset(b))
              ELSE aligned?(bits_per_level[Physical_memory, pm_phy] + pe_size)
                (offset(b))
            ENDIF}):

```

C Proof scripts

Instantiating quantified variables,

we get 3 subgoals:

linear_resolve_unchanged_pm_phy.1.1.3.3.3.1.1.4.1.2.1.1.1:

```

{-1} pm' states(state(read_data(pm_phy, pdir_data_type)(PDBR)(s')))
{-2} union(pm' ro_addr, pm' rw_addr)(addr') ∧
      is_linear_plain_memory?(pm') ∧
      pm' states(s') ∧
      pm' states(s') ∧
      OK?(translate(pdir_lvl, data(read_data(pm_phy, pdir_data_type)(PDBR)(s'))'base_addr,
                    addr', access', segment_to_priv(d'))
            (state(read_data(pm_phy, pdir_data_type)(PDBR)(s'))))
      ∧
      ¬ data(translate(pdir_lvl,
                      data(read_data(pm_phy, pdir_data_type)(PDBR)(s'))'base_addr,
                      addr', access', segment_to_priv(d'))
            (state(read_data(pm_phy, pdir_data_type)(PDBR)(s'))))'1
      ⊃
      pe_in_pt_range?(s',
                      restrict[Address, Memory_Address_4G, boolean]
                        ((pm' ro_addr ∪ pm' rw_addr),
                         ptab_lvl)
                      (xlat_idx(ptab_lvl,
                                data(translate(pdir_lvl,
                                                data(read_data(pm_phy, pdir_data_type)
                                                              (PDBR)
                                                              (s'))'base_addr,
                                                addr', access', seg-
ment_to_priv(d'))
                                (state(read_data(pm_phy, pdir_data_type)
                                              (PDBR)
                                              (s'))))'2,
                                addr'))
                      (addresses ⊆ ((pm_phy'ro_addr ∪ pm_phy'rw_addr) \ address_block(xlat_idx(1, d'''2, addr'), expt(2, 2))))
{-3} Mem?(xlat_idx(1, d'''2, addr')'type_of)
{-4} xlat_idx(1, d'''2, addr')'offset < max_linear_offset
{-5} aligned?(12)(offset(d'''2))
{-6} OK?(translate(0, d''base_addr, addr', access', segment_to_priv(d'))
              (state(read_data(pm_phy, pdir_data_type)(PDBR)(s'))))
      ∧
      d''' =
      data(translate(0, d''base_addr, addr', access', segment_to_priv(d'))
            (state(read_data(pm_phy, pdir_data_type)(PDBR)(s'))))
{-8} OK?(read_data(pm_phy, pdir_data_type)(PDBR)(s')) ∧
      data(read_data(pm_phy, pdir_data_type)(PDBR)(s')) = d''
{-9} ∀ (s: (pm' states)):
      OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s)) ∧
      d' = data(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-10} ∀ (cs: Segment_Reg_type, pdir: Pdir_type):
      ∀ (x1: Physical_memory):
      every[Physical_memory, [bool, Address]]
        (λ (x: Physical_memory): TRUE,
         λ (t: [bool, Address]):
         Mem?(t'2'type_of) ∧
         (0 ≤ t'2'offset ∧ t'2'offset < max_linear_offset) ∧
         IF t'1
           THEN aligned?(22)(offset(t'2))
           ELSE aligned?(12)(offset(t'2))
         ENDIF
        (translate[Physical_memory, pm_phy]
          (0, pdir'base_addr, addr', access', segment_to_priv(cs))(x1))
      1867
{-11} ∀ (cs: Segment_Reg_type, pdir: Pdir_type, leaf: bool,
      base:
      {b: Memory_Address_4G |
      IF leaf
      THEN aligned?(-1 × (pdir_lvl [Physical_memory, pm_phy] × bits_per_level [Physical
      (offset(b))

```

C Proof scripts

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `linear_resolve_unchanged_pm_phy.1.1.3.3.3.1.1.4.1.2.1.1.1.`

linear_resolve_unchanged_pm_phy.1.1.3.3.3.1.1.4.1.2.1.1.2:

```

{-1} pm' states(state(read_data(pm_phy, pdir_data_type)(PDBR)(s')))
{-2} union(pm' ro_addr, pm' rw_addr)(addr') ∧
      is_linear_plain_memory?(pm') ∧
      pm' states(s') ∧
      pm' states(s') ∧
      OK?(translate(pdir_lvl, data(read_data(pm_phy, pdir_data_type)(PDBR)(s'))'base_addr,
                    addr', access', segment_to_priv(d'))
            (state(read_data(pm_phy, pdir_data_type)(PDBR)(s'))))
      ∧
      ¬ data(translate(pdir_lvl,
                      data(read_data(pm_phy, pdir_data_type)(PDBR)(s'))'base_addr,
                      addr', access', segment_to_priv(d'))
              (state(read_data(pm_phy, pdir_data_type)(PDBR)(s'))))'1
      ⊃
      pe_in_pt_range?(s',
                      restrict[Address, Memory_Address_4G, boolean]
                        ((pm' ro_addr ∪ pm' rw_addr),
                         ptab_lvl)
                      (xlat_idx(ptab_lvl,
                                data(translate(pdir_lvl,
                                                data(read_data(pm_phy, pdir_data_type)
                                                              (PDBR)
                                                              (s'))'base_addr,
                                                addr', access', seg-
ment_to_priv(d'))
                                (state(read_data(pm_phy, pdir_data_type)
                                          (PDBR)
                                          (s'))))'2,
                                addr'))
                      (addresses ⊆ ((pm_phy ro_addr ∪ pm_phy rw_addr) \ address_block(xlat_idx(1, d'''2, addr'), expt(2, 2))))
{-3} Mem?(xlat_idx(1, d'''2, addr'))'type_of)
{-4} xlat_idx(1, d'''2, addr'))'offset < max_linear_offset
{-5} aligned?(12)(offset(d'''2))
{-6} OK?(read_data(pm_phy, pdir_data_type)(PDBR)(s')) ∧
      data(read_data(pm_phy, pdir_data_type)(PDBR)(s')) = d''
{-7} ∀ (s: (pm' states)):
      OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s)) ∧
      d' = data(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-8} ∀ (cs: Segment_Reg_type, pdir: Pdir_type):
      ∀ (x1: Physical_memory):
      every[Physical_memory, [bool, Address]]
        (λ (x: Physical_memory): TRUE,
         λ (t: [bool, Address]):
           Mem?(t'2'type_of) ∧
           (0 ≤ t'2'offset ∧ t'2'offset < max_linear_offset) ∧
           IF t'1
             THEN aligned?(22)(offset(t'2))
             ELSE aligned?(12)(offset(t'2))
           ENDIF)
        (translate[Physical_memory, pm_phy]
          (0, pdir'base_addr, addr', access', segment_to_priv(cs))(x1))
{-9} ∀ (cs: Segment_Reg_type, pdir: Pdir_type, leaf: bool,
      base:
        {b: Memory_Address_4G |
          IF leaf
            THEN aligned?(-1 × (pdir_lvl [Physical_memory, pm_phy] 869 bits_per_level [Physical
              (offset(b))
            ELSE aligned?(bits_per_level[Physical_memory, pm_phy] + pe_size)
              (offset(b))
            ENDIF}):
      ¬ leaf ⊃
      (∀ (x1: Physical_memory):
        every[Physical_memory, [bool, Address]]

```

C Proof scripts

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `linear_resolve_unchanged_pm_phy.1.1.3.3.3.1.1.4.1.2.1.1.2.`

linear_resolve_unchanged_pm_phy.1.1.3.3.3.1.1.4.1.2.1.1.3:

```

{-1} pm' states(state(read_data(pm_phy, pdir_data_type)(PDBR)(s')))
{-2} union(pm' ro_addr, pm' rw_addr)(addr') ^
      is_linear_plain_memory?(pm') ^
      pm' states(s') ^
      pm' states(s') ^
      OK?(translate(pdir_lvl, data(read_data(pm_phy, pdir_data_type)(PDBR)(s'))'base_addr,
                    addr', access', segment_to_priv(d'))
            (state(read_data(pm_phy, pdir_data_type)(PDBR)(s'))))
      ^
      ¬ data(translate(pdir_lvl,
                      data(read_data(pm_phy, pdir_data_type)(PDBR)(s'))'base_addr,
                      addr', access', segment_to_priv(d'))
              (state(read_data(pm_phy, pdir_data_type)(PDBR)(s'))))'1
      ⊃
      pe_in_pt_range?(s',
                      restrict[Address, Memory_Address_4G, boolean]
                        ((pm' ro_addr ∪ pm' rw_addr),
                         ptab_lvl)
                      (xlat_idx(ptab_lvl,
                                data(translate(pdir_lvl,
                                                data(read_data(pm_phy, pdir_data_type)
                                                            (PDBR)
                                                            (s'))'base_addr,
                                                addr', access', seg-
ment_to_priv(d'))
                                (state(read_data(pm_phy, pdir_data_type)
                                            (PDBR)
                                            (s'))))'2,
                                addr'))
                      (addresses ⊆ ((pm_phy'ro_addr ∪ pm_phy'rw_addr) \ address_block(xlat_idx(1, d'''2, addr'), expt(2, 2))))
{-3} Mem?(xlat_idx(1, d'''2, addr'))'type_of)
{-4} xlat_idx(1, d'''2, addr'))'offset < max_linear_offset
{-5} aligned?(12)(offset(d'''2))
{-6} ∀ (s: (pm_phy'states)):
      OK?(translate(0, d'''base_addr, addr', access', segment_to_priv(d'))(s) ^
            d''' = data(translate(0, d'''base_addr, addr', access', segment_to_priv(d'))(s))
{-7} OK?(read_data(pm_phy, pdir_data_type)(PDBR)(s')) ^
      data(read_data(pm_phy, pdir_data_type)(PDBR)(s')) = d''
{-8} ∀ (s: (pm' states)):
      OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s) ^
            d' = data(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-9} ∀ (cs: Segment_Reg_type, pdir: Pdir_type):
      ∀ (x1: Physical_memory):
        every[Physical_memory, [bool, Address]]
          (λ (x: Physical_memory): TRUE,
           λ (t: [bool, Address]):
             Mem?(t'2'type_of) ^
             (0 ≤ t'2'offset ^ t'2'offset < max_linear_offset) ^
             IF t'1
               THEN aligned?(22)(offset(t'2))
               ELSE aligned?(12)(offset(t'2))
             ENDIF)
          (translate[Physical_memory, pm_phy]
            (0, pdir'base_addr, addr', access', segment_to_priv(cs))(x1))
{-10} ∀ (cs: Segment_Reg_type, pdir: Pdir_type, leaf: bool,
        base:
          {b: Memory_Address_4G |
            IF leaf
              THEN aligned?(−1 × (pdir_lvl [Physical_memory, pm_phy] × bits_per_level [Physical
                (offset(b))
              ELSE aligned?(bits_per_level[Physical_memory, pm_phy] + pe_size)
                (offset(b))
            ENDIF}):

```


C Proof scripts

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `linear_resolve_unchanged_pm_phy.1.1.3.3.3.1.1.4.1.2.1.1.3`.

linear_resolve_unchanged_pm_phy.1.1.3.3.3.1.1.4.1.2.1.2:

```

{-1} union(pm'ro_addr, pm'rw_addr)(addr') ∧
      is_linear_plain_memory?(pm') ∧
      pm'states(s') ∧
      pm'states(s') ∧
      OK?(translate(pdir_lvl, data(read_data(pm_phy, pabr_data_type)(PDBR)(s'))'base_addr,
                    addr', access', segment_to_priv(d'))
              (state(read_data(pm_phy, pabr_data_type)(PDBR)(s'))))
      ∧
      ¬ data(translate(pdir_lvl,
                      data(read_data(pm_phy, pabr_data_type)(PDBR)(s'))'base_addr,
                      addr', access', segment_to_priv(d'))
              (state(read_data(pm_phy, pabr_data_type)(PDBR)(s'))))'1
      ⊃
      pe_in_pt_range?(s',
                      restrict[Address, Memory_Address_4G, boolean]
                        ((pm'ro_addr ∪ pm'rw_addr),
                         ptab_lvl)
                      (xlat_idx(ptab_lvl,
                                data(translate(pdir_lvl,
                                                data(read_data(pm_phy, pabr_data_type)
                                                              (PDBR)
                                                              (s'))'base_addr,
                                                              addr', access', seg-
ment_to_priv(d'))
                                (state(read_data(pm_phy, pabr_data_type)
                                              (PDBR)
                                              (s'))))'2,
                                addr'))
{-2} (addresses ⊆ ((pm_phy'ro_addr ∪ pm_phy'rw_addr) \ address_block(xlat_idx(1, d'''2, addr'), expt(2, 2))))
{-3} Mem?(xlat_idx(1, d'''2, addr')'type_of)
{-4} xlat_idx(1, d'''2, addr')'offset < max_linear_offset
{-5} aligned?(12)(offset(d'''2))
{-6} ∀ (s: (pm_phy'states)):
      OK?(translate(0, d'''base_addr, addr', access', segment_to_priv(d'))(s)) ∧
      d''' = data(translate(0, d'''base_addr, addr', access', segment_to_priv(d'))(s))
{-7} OK?(read_data(pm_phy, pabr_data_type)(PDBR)(s')) ∧
      data(read_data(pm_phy, pabr_data_type)(PDBR)(s')) = d''
{-8} ∀ (s: (pm'states)):
      OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s)) ∧
      d' = data(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-9} ∀ (cs: Segment_Reg_type, pabr: Pabr_type):
      ∀ (x1: Physical_memory):
        every[Physical_memory, [bool, Address]]
          (λ (x: Physical_memory): TRUE,
           λ (t: [bool, Address]):
             Mem?(t'2'type_of) ∧
             (0 ≤ t'2'offset ∧ t'2'offset < max_linear_offset) ∧
             IF t'1
               THEN aligned?(22)(offset(t'2))
               ELSE aligned?(12)(offset(t'2))
             ENDIF)
          (translate[Physical_memory, pm_phy]
            (0, pabr'base_addr, addr', access', segment_to_priv(cs))(x1))
{-10} ∀ (cs: Segment_Reg_type, pabr: Pabr_type, leaf: bool,
        base:
          {b: Memory_Address_4G |
            IF leaf
              THEN aligned?(−1 × (pdir_lvl [Physical_memory, pm_phy] × bits_per_level [Physical
                (offset(b))
              ELSE aligned?(bits_per_level[Physical_memory, pm_phy] + pe_size)
                (offset(b))
            ENDIF}):
      ¬ leaf ⊃

```

C Proof scripts

Hiding formulas: -1,

Using lemma `pm_unchanged_singleton_linear_resolve_reg_transformers`,

Expanding the definition of `linear_resolve_register_transformers`,

Using lemma `unchanged_memory_invariant_invariant`,

Using lemma `pm_states`,

Using lemma `expr_transformer_invariant_next_ok`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `linear_resolve_unchanged_pm_phy.1.1.3.3.3.1.1.4.1.2.1.2`.

linear_resolve_unchanged_pm_phy.1.1.3.3.3.1.1.4.1.2.1.3:

```

{-1} union(pm'ro_addr, pm'rw_addr)(addr') ∧
      is_linear_plain_memory?(pm') ∧
      pm'states(s') ∧
      pm'states(s') ∧
      OK?(translate(pdir_lvl, data(read_data(pm_phy, pabr_data_type)(PDBR)(s'))'base_addr,
                    addr', access', segment_to_priv(d'))
              (state(read_data(pm_phy, pabr_data_type)(PDBR)(s'))))
      ∧
      ¬ data(translate(pdir_lvl,
                      data(read_data(pm_phy, pabr_data_type)(PDBR)(s'))'base_addr,
                      addr', access', segment_to_priv(d'))
              (state(read_data(pm_phy, pabr_data_type)(PDBR)(s'))))'1
      ⊃
      pe_in_pt_range?(s',
                      restrict[Address, Memory_Address_4G, boolean]
                        ((pm'ro_addr ∪ pm'rw_addr),
                         ptab_lvl)
                      (xlat_idx(ptab_lvl,
                                data(translate(pdir_lvl,
                                                data(read_data(pm_phy, pabr_data_type)
                                                              (PDBR)
                                                              (s'))'base_addr,
                                                              addr', access', seg-
ment_to_priv(d'))
                                (state(read_data(pm_phy, pabr_data_type)
                                              (PDBR)
                                              (s'))))'2,
                                addr'))
{-2} (addresses ⊆ ((pm_phy'ro_addr ∪ pm_phy'rw_addr) \ address_block(xlat_idx(1, d'''2, addr'), expt(2, 2))))
{-3} Mem?(xlat_idx(1, d'''2, addr')'type_of)
{-4} xlat_idx(1, d'''2, addr')'offset < max_linear_offset
{-5} aligned?(12)(offset(d'''2))
{-6} ∀ (s: (pm_phy'states)):
      OK?(translate(0, d'''base_addr, addr', access', segment_to_priv(d'))(s)) ∧
      d''' = data(translate(0, d'''base_addr, addr', access', segment_to_priv(d'))(s))
{-7} OK?(read_data(pm_phy, pabr_data_type)(PDBR)(s')) ∧
      data(read_data(pm_phy, pabr_data_type)(PDBR)(s')) = d''
{-8} ∀ (s: (pm'states)):
      OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s)) ∧
      d' = data(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-9} ∀ (cs: Segment_Reg_type, pabr: Pabr_type):
      ∀ (x1: Physical_memory):
        every[Physical_memory, [bool, Address]]
          (λ (x: Physical_memory): TRUE,
           λ (t: [bool, Address]):
             Mem?(t'2'type_of) ∧
             (0 ≤ t'2'offset ∧ t'2'offset < max_linear_offset) ∧
             IF t'1
               THEN aligned?(22)(offset(t'2))
               ELSE aligned?(12)(offset(t'2))
             ENDIF)
          (translate[Physical_memory, pm_phy]
            (0, pabr'base_addr, addr', access', segment_to_priv(cs))(x1))
{-10} ∀ (cs: Segment_Reg_type, pabr: Pabr_type, leaf: bool,
        base:
          {b: Memory_Address_4G |
            IF leaf
              THEN aligned?(−1 × (pdir_lvl [Physical_memory, pm_phy] × bits_per_level [Physical
                (offset(b))
              ELSE aligned?(bits_per_level[Physical_memory, pm_phy] + pe_size)
                (offset(b))
            ENDIF}):
      ¬ leaf ⊃

```

C Proof scripts

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `linear_resolve_unchanged_pm_phy.1.1.3.3.3.1.1.4.1.2.1.3`.

linear_resolve_unchanged_pm_phy.1.1.3.3.3.1.1.4.1.2.2:

```

{-1}  (addresses ⊆ ((pm_phy'ro_addr ∪ pm_phy'rw_addr) \ address_block(xlat_idx(1, d'''2, addr'), expt(2, 2))))
{-2}  Mem?(xlat_idx(1, d'''2, addr')'type_of)
{-3}  xlat_idx(1, d'''2, addr')'offset < max_linear_offset
{-4}  aligned?(12)(offset(d'''2))
{-5}  ∀ (s: (pm_phy'states)):
      OK?(translate(0, d'''base_addr, addr', access', segment_to_priv(d'))(s) ∧
          d''' = data(translate(0, d'''base_addr, addr', access', segment_to_priv(d'))(s))
{-6}  ∀ (s: (pm_phy'states)):
      OK?(read_data(pm_phy, pdir_data_type)(PDIR)(s) ∧
          data(read_data(pm_phy, pdir_data_type)(PDIR)(s)) = d''
{-7}  ∀ (s: (pm'states)):
      OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s) ∧
          d' = data(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-8}  ∀ (cs: Segment_Reg_type, pdir: Pdir_type):
      ∀ (x1: Physical_memory):
          every[Physical_memory, [bool, Address]]
            (λ (x: Physical_memory): TRUE,
              λ (t: [bool, Address]):
                Mem?(t'2'type_of) ∧
                (0 ≤ t'2'offset ∧ t'2'offset < max_linear_offset) ∧
                IF t'1
                  THEN aligned?(22)(offset(t'2))
                  ELSE aligned?(12)(offset(t'2))
                ENDIF)
            (translate[Physical_memory, pm_phy]
              (0, pdir'base_addr, addr', access', segment_to_priv(cs))(x1))
{-9}  ∀ (cs: Segment_Reg_type, pdir: Pdir_type, leaf: bool,
      base:
        {b: Memory_Address_4G |
          IF leaf
            THEN aligned?(-1 × (pdir_lvl [Physical_memory, pm_phy] × bits_per_level [Physical
              (offset(b))
            ELSE aligned?(bits_per_level[Physical_memory, pm_phy] + pe_size)
              (offset(b))
            ENDIF}):
        ¬ leaf ⊃
      (∀ (x1: Physical_memory):
        every[Physical_memory, [bool, Address]]
          (λ (x: Physical_memory): TRUE,
            λ (t: [bool, Address]):
              Mem?(t'2'type_of) ∧ 0 ≤ t'2'offset ∧ t'2'offset < max_linear_offset)
          (translate[Physical_memory, pm_phy]
            (1, base, addr', access', segment_to_priv(cs))(x1)))
{-10} pm'mem = linear_pm
{-11} ∀ (s: (pm'states)): OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-12} ∀ (s1, s2: (pm'states)):
      data(read_data(pm_phy, segment_reg_data_type)(CS)(s1)) =
        data(read_data(pm_phy, segment_reg_data_type)(CS)(s2))
{-13} ∀ (s: (pm'states)): OK?(read_data(pm_phy, pdir_data_type)(PDIR)(s))
{-14} ∀ (s1, s2: (pm'states)):
      data(read_data(pm_phy, pdir_data_type)(PDIR)(s1)) =
        data(read_data(pm_phy, pdir_data_type)(PDIR)(s2))
{-15} ∀ (s1, s2: (pm'states)):
      pe_in_pdir_range?(s1,
        restrict[Address, Memory_Address_4G, boolean]
          ((pm'ro_addr ∪ pm'rw_addr)))
      =
      pe_in_pdir_range?(s2,
        restrict[Address, Memory_Address_4G, boolean]
          ((pm'ro_addr ∪ pm'rw_addr)))
      ∧
      pe_in_ptab_range?(s1,
        restrict[Address, Memory_Address_4G, boolean]
          ((pm'ro_addr ∪ pm'rw_addr)))

```

C Proof scripts

Using lemma `xlat_idx_memory_address2`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `linear_resolve_unchanged_pm_phy.1.1.3.3.3.1.1.4.1.2.2`.

linear_resolve_unchanged_pm_phy.1.1.3.3.3.1.1.4.2:

```

{-1} (addresses ⊆ ((pm_phy'ro_addr ∪ pm_phy'rw_addr) \ address_block(xlat_idx(1, d'''2, addr'), expt(2, 2))))
{-2} Mem?(xlat_idx(1, d'''2, addr')'type_of)
{-3} xlat_idx(1, d'''2, addr')'offset < max_linear_offset
{-4} aligned?(12)(offset(d'''2))
{-5} ∀ (s: (pm_phy'states)):
      OK?(translate(0, d''base_addr, addr', access', segment_to_priv(d'))(s) ∧
            d''' = data(translate(0, d''base_addr, addr', access', segment_to_priv(d'))(s))
{-6} ∀ (s: (pm_phy'states)):
      OK?(read_data(pm_phy, pdir_data_type)(PDBR)(s) ∧
            data(read_data(pm_phy, pdir_data_type)(PDBR)(s)) = d''
{-7} ∀ (s: (pm' states)):
      OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s) ∧
            d' = data(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-8} ∀ (cs: Segment_Reg_type, pdir: Pdir_type):
      ∀ (x1: Physical_memory):
        every[Physical_memory, [bool, Address]]
          (λ (x: Physical_memory): TRUE,
            λ (t: [bool, Address]):
              Mem?(t'2'type_of) ∧
              (0 ≤ t'2'offset ∧ t'2'offset < max_linear_offset) ∧
              IF t'1
                THEN aligned?(22)(offset(t'2))
                ELSE aligned?(12)(offset(t'2))
              ENDIF)
          (translate[Physical_memory, pm_phy]
            (0, pdir'base_addr, addr', access', segment_to_priv(cs))(x1))
{-9} ∀ (cs: Segment_Reg_type, pdir: Pdir_type, leaf: bool,
      base:
        {b: Memory_Address_4G |
          IF leaf
            THEN aligned?(-1 × (pdir_lvl [Physical_memory, pm_phy] × bits_per_level [Physical
              (offset(b))
            ELSE aligned?(bits_per_level[Physical_memory, pm_phy] + pe_size)
              (offset(b))
            ENDIF}):
        ¬ leaf ⊃
      (∀ (x1: Physical_memory):
        every[Physical_memory, [bool, Address]]
          (λ (x: Physical_memory): TRUE,
            λ (t: [bool, Address]):
              Mem?(t'2'type_of) ∧ 0 ≤ t'2'offset ∧ t'2'offset < max_linear_offset)
          (translate[Physical_memory, pm_phy]
            (1, base, addr', access', segment_to_priv(cs))(x1)))
{-10} pm'mem = linear_pm
{-11} ∀ (s: (pm' states)): OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-12} ∀ (s1, s2: (pm' states)):
      data(read_data(pm_phy, segment_reg_data_type)(CS)(s1)) =
      data(read_data(pm_phy, segment_reg_data_type)(CS)(s2))
{-13} ∀ (s: (pm' states)): OK?(read_data(pm_phy, pdir_data_type)(PDBR)(s))
{-14} ∀ (s1, s2: (pm' states)):
      data(read_data(pm_phy, pdir_data_type)(PDBR)(s1)) =
      data(read_data(pm_phy, pdir_data_type)(PDBR)(s2))
{-15} ∀ (s1, s2: (pm' states)):
      pe_in_pdir_range?(s1,
        restrict[Address, Memory_Address_4G, boolean]
          ((pm'ro_addr ∪ pm'rw_addr)))
      =
      pe_in_pdir_range?(s2,
        restrict[Address, Memory_Address_4G, boolean]
          ((pm'ro_addr ∪ pm'rw_addr)))
      ∧
      pe_in_ptab_range?(s1,
        restrict[Address, Memory_Address_4G, boolean]
          ((pm'ro_addr ∪ pm'rw_addr)))

```


C Proof scripts

Using lemma `xlat_idx_memory_address2`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `linear_resolve_unchanged_pm_phy.1.1.3.3.3.1.1.4.2`.

linear_resolve_unchanged_pm_phy.1.1.3.3.3.1.2:

```

{-1} ((address_block(xlat_idx(1, d'''2, addr'), expt(2, 2)) ⊆ pm_phy'rw_addr) ∧
      (∀ (s: (pm' states)):
        OK?(read_data(pm_phy, paging_data_type(1))(xlat_idx(1, d'''2, addr'))(s)))
      ∧
      (∀ (s1, s2: (pm' states)):
        set_reference(data(read_data(pm_phy, paging_data_type(1))
                              (xlat_idx(1, d'''2, addr'))(s1)),
                      Write)
          =
          set_reference(data(read_data(pm_phy, paging_data_type(1))
                              (xlat_idx(1, d'''2, addr'))(s2)),
                      Write)))
      ⊃
      unchanged_memory_invariant?(pm_phy' mem, pm_phy' states,
                                   singleton(expr_2_super(translate(1,
                                                                    d'''2,
                                                                    addr',
                                                                    access',
                                                                    seg-
                                                                    ment_to_priv(d')))),
                                   ((pm_phy'ro_addr ∪ pm_phy'rw_addr) \ address_block(xlat_idx(1, d'''2,
{-2} xlat_idx(1, d'''2, addr')'type_of = Mem_ ∧
      xlat_idx(1, d'''2, addr')'offset < max_linear_offset
{-3} aligned?(12)(offset(d'''2))
{-4} ∀ (s: (pm_phy' states)):
      OK?(translate(0, d'''base_addr, addr', access', segment_to_priv(d'))(s)) ∧
      d''' = data(translate(0, d'''base_addr, addr', access', segment_to_priv(d'))(s))
{-5} ∀ (s: (pm_phy' states)):
      OK?(read_data(pm_phy, pddb_data_type)(PDBR)(s)) ∧
      data(read_data(pm_phy, pddb_data_type)(PDBR)(s)) = d''
{-6} ∀ (s: (pm' states)):
      OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s)) ∧
      d' = data(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-7} ∀ (cs: Segment_Reg_type, pddb: Pddb_type):
      ∀ (x1: Physical_memory):
        every[Physical_memory, [bool, Address]]
          (λ (x: Physical_memory): TRUE,
           λ (t: [bool, Address]):
             Mem?(t'2'type_of) ∧
             (0 ≤ t'2'offset ∧ t'2'offset < max_linear_offset) ∧
             IF t'1
               THEN aligned?(22)(offset(t'2))
               ELSE aligned?(12)(offset(t'2))
             ENDIF)
          (translate[Physical_memory, pm_phy]
           (0, pddb'base_addr, addr', access', segment_to_priv(cs))(x1))
{-8} ∀ (cs: Segment_Reg_type, pddb: Pddb_type, leaf: bool,
      base:
        {b: Memory_Address_4G |
          IF leaf
            THEN aligned?(-1 × (pdir_lvl [Physical_memory, pm_phy] × bits_per_level [Physical
            (offset(b))
            ELSE aligned?(bits_per_level[Physical_memory, pm_phy] + pe_size)
            (offset(b))
          ENDIF}):
      ¬ leaf ⊃
      (∀ (x1: Physical_memory):
        every[Physical_memory, [bool, Address]]
          (λ (x: Physical_memory): TRUE,
           λ (t: [bool, Address]):
             Mem?(t'2'type_of) ∧ 0 ≤ t'2'offset ∧ t'2'offset < max_linear_offset)
          (translate[Physical_memory, pm_phy]
           (1, base, addr', access', segment_to_priv(cs))(x1)))

```

C Proof scripts

Hiding formulas: (2 3),

Replacing using formula -24,

Rewriting using subset_of_differences, matching in *,

Using lemma pe_in_pt_address_in_pt,

we get 2 subgoals:

linear_resolve_unchanged_pm_phy.1.1.3.3.3.1.2.1:

```

{-1} pe_in_pt_range?(s',
      restrict[Address, Memory_Address_4G, boolean]
        ((pm'ro_addr ∪ pm'rw_addr)),
      1)
      (xlat_idx(1, d'''2, addr'))
  ⊃
  (address_block(xlat_idx(1, d'''2, addr'), expt(2, pe_size)) ⊆ extend [Address, Memory_Address_4G, bo
{-2} ((address_block(xlat_idx(1, d'''2, addr'), expt(2, 2)) ⊆ pm_phy'rw_addr) ∧
      (∀ (s: (pm'states)):
        OK?(read_data(pm_phy, paging_data_type(1))(xlat_idx(1, d'''2, addr'))(s)))
      ∧
      (∀ (s1, s2: (pm'states)):
        set_reference(data(read_data(pm_phy, paging_data_type(1))
                          (xlat_idx(1, d'''2, addr'))(s1)),
                      Write)
          =
          set_reference(data(read_data(pm_phy, paging_data_type(1))
                          (xlat_idx(1, d'''2, addr'))(s2)),
                      Write)))
  ⊃
  unchanged_memory_invariant?(pm_phy'mem, pm_phy'states,
                              singleton(expr_2_super(translate(1,
                                                                d'''2,
                                                                addr',
                                                                access',
                                                                seg-
                                                                ment_to_priv(d')))),
                              ((pm_phy'ro_addr ∪ pm_phy'rw_addr) \ address_block(xlat_idx(1, d'''2,
{-3} xlat_idx(1, d'''2, addr')'type_of = Mem_ ∧
      xlat_idx(1, d'''2, addr')'offset < max_linear_offset
{-4} aligned?(12)(offset(d'''2))
{-5} ∀ (s: (pm_phy'states)):
      OK?(translate(0, d''base_addr, addr', access', segment_to_priv(d'))(s)) ∧
      d''' = data(translate(0, d''base_addr, addr', access', segment_to_priv(d'))(s))
{-6} ∀ (s: (pm_phy'states)):
      OK?(read_data(pm_phy, pdbr_data_type)(PDBR)(s)) ∧
      data(read_data(pm_phy, pdbr_data_type)(PDBR)(s)) = d''
{-7} ∀ (s: (pm'states)):
      OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s)) ∧
      d' = data(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-8} ∀ (cs: Segment_Reg_type, pdbr: Pdbr_type):
      ∀ (x1: Physical_memory):
        every[Physical_memory, [bool, Address]]
          (λ (x: Physical_memory): TRUE,
           λ (t: [bool, Address]):
             Mem?(t'2'type_of) ∧
             (0 ≤ t'2'offset ∧ t'2'offset < max_linear_offset) ∧
             IF t'1
               THEN aligned?(22)(offset(t'2))
               ELSE aligned?(12)(offset(t'2))
             ENDIF)
          (translate[Physical_memory, pm_phy]
           (0, pdbr'base_addr, addr', access', segment_to_priv(cs))(x1))
{-9} ∀ (cs: Segment_Reg_type, pdbr: Pdbr_type, leaf: bool,
      base:
        {b: Memory_Address_4G |
          IF leaf
            THEN aligned?(-1 × (pdir_lvl [Physical_memory, pm_phy] × bits_per_level [Physical
              (offset(b))
            ELSE aligned?(bits_per_level[Physical_memory, pm_phy] + pe_size)
              (offset(b))
            ENDIF}):
      ¬ leaf ⊃

```

C Proof scripts

Hiding formulas: -2,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Using lemma `xlat_pe_in_pt_range`,

Instantiating quantified variables,

Case splitting on `pm!1'states(state(read_data(pm_phy, pdbr_data_type)(PDBR)(s!1)))`,

we get 3 subgoals:

linear_resolve_unchanged_pm_phy.1.1.3.3.3.1.2.1.1:

```

{-1} pm' states(state(read_data(pm_phy, pdir_data_type)(PDBR)(s')))
{-2} union(pm' ro_addr, pm' rw_addr)(addr') ∧
      is_linear_plain_memory?(pm') ∧
      pm' states(s') ∧
      pm' states(s') ∧
      OK?(translate(pdir_lvl, data(read_data(pm_phy, pdir_data_type)(PDBR)(s'))'base_addr,
                    addr', access', segment_to_priv(d'))
              (state(read_data(pm_phy, pdir_data_type)(PDBR)(s'))))
      ∧
      ¬ data(translate(pdir_lvl,
                      data(read_data(pm_phy, pdir_data_type)(PDBR)(s'))'base_addr,
                      addr', access', segment_to_priv(d'))
              (state(read_data(pm_phy, pdir_data_type)(PDBR)(s'))))'1
      ⊃
      pe_in_pt_range?(s',
                      restrict[Address, Memory_Address_4G, boolean]
                        ((pm' ro_addr ∪ pm' rw_addr),
                         ptab_lvl)
                      (xlat_idx(ptab_lvl,
                                data(translate(pdir_lvl,
                                                data(read_data(pm_phy, pdir_data_type)
                                                              (PDBR)
                                                              (s'))'base_addr,
                                                addr', access', seg-
ment_to_priv(d'))
                                (state(read_data(pm_phy, pdir_data_type)
                                              (PDBR)
                                              (s'))))'2,
                                addr'))
{-3} Mem?(xlat_idx(1, d''' '2, addr')'type_of)
{-4} xlat_idx(1, d''' '2, addr')'offset < max_linear_offset
{-5} aligned?(12)(offset(d''' '2))
{-6} ∀ (s: (pm_phy' states)):
      OK?(translate(0, d''' 'base_addr, addr', access', segment_to_priv(d'))(s)) ∧
      d''' = data(translate(0, d''' 'base_addr, addr', access', segment_to_priv(d'))(s))
{-7} OK?(read_data(pm_phy, pdir_data_type)(PDBR)(s')) ∧
      data(read_data(pm_phy, pdir_data_type)(PDBR)(s')) = d''
{-8} ∀ (s: (pm' states)):
      OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s)) ∧
      d' = data(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-9} ∀ (cs: Segment_Reg_type, pdir: Pdir_type):
      ∀ (x1: Physical_memory):
        every[Physical_memory, [bool, Address]]
          (λ (x: Physical_memory): TRUE,
           λ (t: [bool, Address]):
             Mem?(t'2'type_of) ∧
             (0 ≤ t'2'offset ∧ t'2'offset < max_linear_offset) ∧
             IF t'1
               THEN aligned?(22)(offset(t'2))
               ELSE aligned?(12)(offset(t'2))
             ENDIF)
          (translate[Physical_memory, pm_phy]
            (0, pdir'base_addr, addr', access', segment_to_priv(cs))(x1))
{-10} ∀ (cs: Segment_Reg_type, pdir: Pdir_type, leaf: bool,
        base:
          {b: Memory_Address_4G |
            IF leaf
              THEN aligned?(−1 × (pdir_lvl [Physical_memory, pm_phy] × bits_per_level [Physical
                (offset(b))
              ELSE aligned?(bits_per_level[Physical_memory, pm_phy] + pe_size)
                (offset(b))
            ENDIF}):
      ¬ leaf ⊃

```

C Proof scripts

Instantiating quantified variables,

we get 3 subgoals:

linear_resolve_unchanged_pm_phy.1.1.3.3.3.1.2.1.1.1:

```

{-1} pm' states(state(read_data(pm_phy, pdir_data_type)(PDBR)(s')))
{-2} union(pm' ro_addr, pm' rw_addr)(addr') ∧
      is_linear_plain_memory?(pm') ∧
      pm' states(s') ∧
      pm' states(s') ∧
      OK?(translate(pdir_lvl, data(read_data(pm_phy, pdir_data_type)(PDBR)(s'))'base_addr,
                    addr', access', segment_to_priv(d'))
            (state(read_data(pm_phy, pdir_data_type)(PDBR)(s'))))
      ∧
      ¬ data(translate(pdir_lvl,
                    data(read_data(pm_phy, pdir_data_type)(PDBR)(s'))'base_addr,
                    addr', access', segment_to_priv(d'))
            (state(read_data(pm_phy, pdir_data_type)(PDBR)(s'))))'1
      ⊃
      pe_in_pt_range?(s',
                    restrict[Address, Memory_Address_4G, boolean]
                      ((pm' ro_addr ∪ pm' rw_addr)),
                    ptab_lvl)
                    (xlat_idx(ptab_lvl,
                              data(translate(pdir_lvl,
                                              data(read_data(pm_phy, pdir_data_type)
                                                    (PDBR)
                                                    (s'))'base_addr,
                                              addr', access', seg-
ment_to_priv(d'))
                              (state(read_data(pm_phy, pdir_data_type)
                                        (PDBR)
                                        (s'))))'2,
                              addr'))
{-3} Mem?(xlat_idx(1, d''' '2, addr')'type_of)
{-4} xlat_idx(1, d''' '2, addr)'offset < max_linear_offset
{-5} aligned?(12)(offset(d''' '2))
{-6} OK?(translate(0, d''' 'base_addr, addr', access', segment_to_priv(d'))
            (state(read_data(pm_phy, pdir_data_type)(PDBR)(s'))))
      ∧
      d''' =
      data(translate(0, d''' 'base_addr, addr', access', segment_to_priv(d'))
            (state(read_data(pm_phy, pdir_data_type)(PDBR)(s'))))
{-7} OK?(read_data(pm_phy, pdir_data_type)(PDBR)(s')) ∧
      data(read_data(pm_phy, pdir_data_type)(PDBR)(s')) = d''
{-8} ∀ (s: (pm' states)):
      OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s)) ∧
      d' = data(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-9} ∀ (cs: Segment_Reg_type, pdir: Pdir_type):
      ∀ (x1: Physical_memory):
      every[Physical_memory, [bool, Address]]
        (λ (x: Physical_memory): TRUE,
         λ (t: [bool, Address]):
         Mem?(t'2'type_of) ∧
         (0 ≤ t'2'offset ∧ t'2'offset < max_linear_offset) ∧
         IF t'1
           THEN aligned?(22)(offset(t'2))
           ELSE aligned?(12)(offset(t'2))
         ENDIF)
        (translate[Physical_memory, pm_phy]
          (0, pdir'base_addr, addr', access', segment_to_priv(cs))(x1))1887
{-10} ∀ (cs: Segment_Reg_type, pdir: Pdir_type, leaf: bool,
        base:
          {b: Memory_Address_4G |
            IF leaf
              THEN aligned?(-1 × (pdir_lvl [Physical_memory, pm_phy] × bits_per_level [Physical
              (offset(b))
            ELSE aligned?(bits_per_level[Physical_memory, pm_phy] + pe_size)

```


C Proof scripts

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `linear_resolve_unchanged_pm_phy.1.1.3.3.3.1.2.1.1.1.`

linear_resolve_unchanged_pm_phy.1.1.3.3.3.1.2.1.1.2:

```

{-1} pm' states(state(read_data(pm_phy, pdir_data_type)(PDBR)(s')))
{-2} union(pm' ro_addr, pm' rw_addr)(addr') ∧
      is_linear_plain_memory?(pm') ∧
      pm' states(s') ∧
      pm' states(s') ∧
      OK?(translate(pdir_lvl, data(read_data(pm_phy, pdir_data_type)(PDBR)(s'))'base_addr,
                    addr', access', segment_to_priv(d'))
            (state(read_data(pm_phy, pdir_data_type)(PDBR)(s'))))
      ∧
      ¬ data(translate(pdir_lvl,
                      data(read_data(pm_phy, pdir_data_type)(PDBR)(s'))'base_addr,
                      addr', access', segment_to_priv(d'))
              (state(read_data(pm_phy, pdir_data_type)(PDBR)(s'))))'1
      ⊃
      pe_in_pt_range?(s',
                      restrict[Address, Memory_Address_4G, boolean]
                        ((pm' ro_addr ∪ pm' rw_addr),
                         ptab_lvl)
                      (xlat_idx(ptab_lvl,
                                data(translate(pdir_lvl,
                                                data(read_data(pm_phy, pdir_data_type)
                                                              (PDBR)
                                                              (s'))'base_addr,
                                                              addr', access', seg-
ment_to_priv(d'))
                                (state(read_data(pm_phy, pdir_data_type)
                                              (PDBR)
                                              (s'))))'2,
                                addr'))
                      Mem?(xlat_idx(1, d'''2, addr')'type_of)
{-3} xlat_idx(1, d'''2, addr')'offset < max_linear_offset
{-4} aligned?(12)(offset(d'''2))
{-5} OK?(read_data(pm_phy, pdir_data_type)(PDBR)(s') ∧
        data(read_data(pm_phy, pdir_data_type)(PDBR)(s')) = d''
{-6} ∀ (s: (pm' states)):
      OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s) ∧
          d' = data(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-7} ∀ (cs: Segment_Reg_type, pdir: Pdir_type):
      ∀ (x1: Physical_memory):
        every[Physical_memory, [bool, Address]]
          (λ (x: Physical_memory): TRUE,
           λ (t: [bool, Address]):
             Mem?(t'2'type_of) ∧
             (0 ≤ t'2'offset ∧ t'2'offset < max_linear_offset) ∧
             IF t'1
               THEN aligned?(22)(offset(t'2))
               ELSE aligned?(12)(offset(t'2))
             ENDIF)
          (translate[Physical_memory, pm_phy]
            (0, pdir'base_addr, addr', access', segment_to_priv(cs))(x1))
{-8} ∀ (cs: Segment_Reg_type, pdir: Pdir_type, leaf: bool,
      base:
        {b: Memory_Address_4G |
          IF leaf
            THEN aligned?(-1 × (pdir_lvl [Physical_memory, pm_phy] × bits_per_level [Physical
            (offset(b))
            1889
            ELSE aligned?(bits_per_level[Physical_memory, pm_phy] + pe_size)
            (offset(b))
          ENDIF}):
      ¬ leaf ⊃
      (∀ (x1: Physical_memory):
        every[Physical_memory, [bool, Address]]
          (λ (x: Physical_memory): TRUE,

```

C Proof scripts

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `linear_resolve_unchanged_pm_phy.1.1.3.3.3.1.2.1.1.2.`

linear_resolve_unchanged_pm_phy.1.1.3.3.3.1.2.1.1.3:

```

{-1} pm' states(state(read_data(pm_phy, pdir_data_type)(PDBR)(s')))
{-2} union(pm' ro_addr, pm' rw_addr)(addr') ∧
      is_linear_plain_memory?(pm') ∧
      pm' states(s') ∧
      pm' states(s') ∧
      OK?(translate(pdir_lvl, data(read_data(pm_phy, pdir_data_type)(PDBR)(s'))'base_addr,
                    addr', access', segment_to_priv(d'))
              (state(read_data(pm_phy, pdir_data_type)(PDBR)(s'))))
      ∧
      ¬ data(translate(pdir_lvl,
                      data(read_data(pm_phy, pdir_data_type)(PDBR)(s'))'base_addr,
                      addr', access', segment_to_priv(d'))
              (state(read_data(pm_phy, pdir_data_type)(PDBR)(s'))))'1
      ⊃
      pe_in_pt_range?(s',
                      restrict[Address, Memory_Address_4G, boolean]
                        ((pm' ro_addr ∪ pm' rw_addr),
                         ptab_lvl)
                      (xlat_idx(ptab_lvl,
                                data(translate(pdir_lvl,
                                                data(read_data(pm_phy, pdir_data_type)
                                                            (PDBR)
                                                            (s'))'base_addr,
                                                addr', access', seg-
ment_to_priv(d'))
                                (state(read_data(pm_phy, pdir_data_type)
                                            (PDBR)
                                            (s'))))'2,
                                addr'))
{-3} Mem?(xlat_idx(1, d''' '2, addr')'type_of)
{-4} xlat_idx(1, d''' '2, addr')'offset < max_linear_offset
{-5} aligned?(12)(offset(d''' '2))
{-6} ∀ (s: (pm_phy' states)):
      OK?(translate(0, d''' 'base_addr, addr', access', segment_to_priv(d'))(s)) ∧
      d''' = data(translate(0, d''' 'base_addr, addr', access', segment_to_priv(d'))(s))
{-7} OK?(read_data(pm_phy, pdir_data_type)(PDBR)(s')) ∧
      data(read_data(pm_phy, pdir_data_type)(PDBR)(s')) = d''
{-8} ∀ (s: (pm' states)):
      OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s)) ∧
      d' = data(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-9} ∀ (cs: Segment_Reg_type, pdir: Pdir_type):
      ∀ (x1: Physical_memory):
        every[Physical_memory, [bool, Address]]
          (λ (x: Physical_memory): TRUE,
           λ (t: [bool, Address]):
             Mem?(t'2'type_of) ∧
             (0 ≤ t'2'offset ∧ t'2'offset < max_linear_offset) ∧
             IF t'1
               THEN aligned?(22)(offset(t'2))
               ELSE aligned?(12)(offset(t'2))
             ENDIF)
          (translate[Physical_memory, pm_phy]
            (0, pdir'base_addr, addr', access', segment_to_priv(cs))(x1))
{-10} ∀ (cs: Segment_Reg_type, pdir: Pdir_type, leaf: bool,
        base:
          {b: Memory_Address_4G |
            IF leaf
              THEN aligned?(−1 × (pdir_lvl [Physical_memory, pm_phy] × bits_per_level [Physical
                (offset(b))
              ELSE aligned?(bits_per_level[Physical_memory, pm_phy] + pe_size)
                (offset(b))
            ENDIF}):
      ¬ leaf ⊃

```

C Proof scripts

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `linear_resolve_unchanged_pm_phy.1.1.3.3.3.1.2.1.1.3.`

linear_resolve_unchanged_pm_phy.1.1.3.3.3.1.2.1.2:

```

{-1} union(pm'ro_addr, pm'rw_addr)(addr') ∧
      is_linear_plain_memory?(pm') ∧
      pm'states(s') ∧
      pm'states(s') ∧
      OK?(translate(pdir_lvl, data(read_data(pm_phy, pabr_data_type)(PDBR)(s'))'base_addr,
                    addr', access', segment_to_priv(d'))
              (state(read_data(pm_phy, pabr_data_type)(PDBR)(s'))))
      ∧
      ¬ data(translate(pdir_lvl,
                      data(read_data(pm_phy, pabr_data_type)(PDBR)(s'))'base_addr,
                      addr', access', segment_to_priv(d'))
              (state(read_data(pm_phy, pabr_data_type)(PDBR)(s'))))'1
      ⊃
      pe_in_pt_range?(s',
                      restrict[Address, Memory_Address_4G, boolean]
                        ((pm'ro_addr ∪ pm'rw_addr),
                         ptab_lvl)
                      (xlat_idx(ptab_lvl,
                                data(translate(pdir_lvl,
                                                data(read_data(pm_phy, pabr_data_type)
                                                            (PDBR)
                                                            (s'))'base_addr,
                                                addr', access', seg-
ment_to_priv(d'))
                                (state(read_data(pm_phy, pabr_data_type)
                                                (PDBR)
                                                (s'))))'2,
                                addr'))
{-2} Mem?(xlat_idx(1, d'''2, addr')'type_of)
{-3} xlat_idx(1, d'''2, addr')'offset < max_linear_offset
{-4} aligned?(12)(offset(d'''2))
{-5} ∀ (s: (pm_phy'states)):
      OK?(translate(0, d'''base_addr, addr', access', segment_to_priv(d'))(s)) ∧
      d''' = data(translate(0, d'''base_addr, addr', access', segment_to_priv(d'))(s))
{-6} OK?(read_data(pm_phy, pabr_data_type)(PDBR)(s')) ∧
      data(read_data(pm_phy, pabr_data_type)(PDBR)(s')) = d''
{-7} ∀ (s: (pm'states)):
      OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s)) ∧
      d' = data(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-8} ∀ (cs: Segment_Reg_type, pabr: Pabr_type):
      ∀ (x1: Physical_memory):
        every[Physical_memory, [bool, Address]]
          (λ (x: Physical_memory): TRUE,
           λ (t: [bool, Address]):
             Mem?(t'2'type_of) ∧
             (0 ≤ t'2'offset ∧ t'2'offset < max_linear_offset) ∧
             IF t'1
               THEN aligned?(22)(offset(t'2))
               ELSE aligned?(12)(offset(t'2))
             ENDIF)
          (translate[Physical_memory, pm_phy]
            (0, pabr'base_addr, addr', access', segment_to_priv(cs))(x1))
{-9} ∀ (cs: Segment_Reg_type, pabr: Pabr_type, leaf: bool,
      base:
        {b: Memory_Address_4G |
          IF leaf
            THEN aligned?(-1 × (pdir_lvl [Physical_memory, pm_phy] × bits_per_level [Physical
              (offset(b))
            ELSE aligned?(bits_per_level[Physical_memory, pm_phy] + pe_size)
              (offset(b))
            ENDIF}):
      ¬ leaf ⊃
      (∀ (x1: Physical_memory):

```

C Proof scripts

Hiding formulas: -1,

Using lemma `pm_unchanged_singleton_linear_resolve_reg_transformers`,

Expanding the definition of `linear_resolve_register_transformers`,

Using lemma `unchanged_memory_invariant_invariant`,

Using lemma `expr_transformer_invariant_next_ok`,

Using lemma `pm_states`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `linear_resolve_unchanged_pm_phy.1.1.3.3.3.1.2.1.2`.

linear_resolve_unchanged_pm_phy.1.1.3.3.3.1.2.1.3:

```

{-1}  union(pm'ro_addr, pm'rw_addr)(addr') ∧
      is_linear_plain_memory?(pm') ∧
      pm'states(s') ∧
      pm'states(s') ∧
      OK?(translate(pdir_lvl, data(read_data(pm_phy, pabr_data_type)(PDBR)(s'))'base_addr,
                    addr', access', segment_to_priv(d'))
              (state(read_data(pm_phy, pabr_data_type)(PDBR)(s'))))
      ∧
      ¬ data(translate(pdir_lvl,
                      data(read_data(pm_phy, pabr_data_type)(PDBR)(s'))'base_addr,
                      addr', access', segment_to_priv(d'))
              (state(read_data(pm_phy, pabr_data_type)(PDBR)(s'))))'1
      ⊃
      pe_in_pt_range?(s',
                      restrict[Address, Memory_Address_4G, boolean]
                        ((pm'ro_addr ∪ pm'rw_addr),
                         ptab_lvl)
                      (xlat_idx(ptab_lvl,
                                data(translate(pdir_lvl,
                                              data(read_data(pm_phy, pabr_data_type)
                                                            (PDBR)
                                                            (s'))'base_addr,
                                              addr', access', seg-
ment_to_priv(d'))
                                (state(read_data(pm_phy, pabr_data_type)
                                              (PDBR)
                                              (s'))))'2,
                                addr'))
{-2}  Mem?(xlat_idx(1, d'''2, addr')'type_of)
{-3}  xlat_idx(1, d'''2, addr')'offset < max_linear_offset
{-4}  aligned?(12)(offset(d'''2))
{-5}  ∀ (s: (pm_phy'states)):
      OK?(translate(0, d'''base_addr, addr', access', segment_to_priv(d'))(s)) ∧
      d''' = data(translate(0, d'''base_addr, addr', access', segment_to_priv(d'))(s))
{-6}  OK?(read_data(pm_phy, pabr_data_type)(PDBR)(s')) ∧
      data(read_data(pm_phy, pabr_data_type)(PDBR)(s')) = d''
{-7}  ∀ (s: (pm'states)):
      OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s)) ∧
      d' = data(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-8}  ∀ (cs: Segment_Reg_type, pabr: Pabr_type):
      ∀ (x1: Physical_memory):
      every[Physical_memory, [bool, Address]]
        (λ (x: Physical_memory): TRUE,
         λ (t: [bool, Address]):
           Mem?(t'2'type_of) ∧
           (0 ≤ t'2'offset ∧ t'2'offset < max_linear_offset) ∧
           IF t'1
             THEN aligned?(22)(offset(t'2))
             ELSE aligned?(12)(offset(t'2))
           ENDIF)
        (translate[Physical_memory, pm_phy]
         (0, pabr'base_addr, addr', access', segment_to_priv(cs))(x1))
{-9}  ∀ (cs: Segment_Reg_type, pabr: Pabr_type, leaf: bool,
      base:
        {b: Memory_Address_4G |
          IF leaf
            THEN aligned?(-1 × (pdir_lvl [Physical_memory, pm_phy] × bits_per_level [Physical
              (offset(b))
            ELSE aligned?(bits_per_level[Physical_memory, pm_phy] + pe_size)
              (offset(b))
          ENDIF}):
      ¬ leaf ⊃
      (∀ (x1: Physical_memory):

```


C Proof scripts

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `linear_resolve_unchanged_pm_phy.1.1.3.3.3.1.2.1.3`.

linear_resolve_unchanged_pm_phy.1.1.3.3.3.1.2.2:

```

{-1} ((address_block(xlat_idx(1, d'''2, addr'), expt(2, 2)) ⊆ pm_phy'rw_addr) ∧
      (∀ (s: (pm' states)):
        OK?(read_data(pm_phy, paging_data_type(1))(xlat_idx(1, d'''2, addr'))(s)))
      ∧
      (∀ (s1, s2: (pm' states)):
        set_reference(data(read_data(pm_phy, paging_data_type(1))
                              (xlat_idx(1, d'''2, addr'))(s1)),
                      Write)
          =
        set_reference(data(read_data(pm_phy, paging_data_type(1))
                              (xlat_idx(1, d'''2, addr'))(s2)),
                      Write)))
      ⊃
      unchanged_memory_invariant?(pm_phy' mem, pm_phy' states,
                                   singleton(expr_2_super(translate(1,
                                                                    d'''2,
                                                                    addr',
                                                                    access',
                                                                    seg-
                                                                    ment_to_priv(d')))),
                                   ((pm_phy'ro_addr ∪ pm_phy'rw_addr) \ address_block(xlat_idx(1, d'''2,
{-2} xlat_idx(1, d'''2, addr')'type_of = Mem_ ∧
      xlat_idx(1, d'''2, addr')'offset < max_linear_offset
{-3} aligned?(12)(offset(d'''2))
{-4} ∀ (s: (pm_phy' states)):
      OK?(translate(0, d'''base_addr, addr', access', segment_to_priv(d'))(s)) ∧
      d''' = data(translate(0, d'''base_addr, addr', access', segment_to_priv(d'))(s))
{-5} ∀ (s: (pm_phy' states)):
      OK?(read_data(pm_phy, pddb_data_type)(PDBR)(s)) ∧
      data(read_data(pm_phy, pddb_data_type)(PDBR)(s)) = d''
{-6} ∀ (s: (pm' states)):
      OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s)) ∧
      d' = data(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-7} ∀ (cs: Segment_Reg_type, pddb: Pddb_type):
      ∀ (x1: Physical_memory):
        every[Physical_memory, [bool, Address]]
          (λ (x: Physical_memory): TRUE,
           λ (t: [bool, Address]):
             Mem?(t'2'type_of) ∧
             (0 ≤ t'2'offset ∧ t'2'offset < max_linear_offset) ∧
             IF t'1
               THEN aligned?(22)(offset(t'2))
               ELSE aligned?(12)(offset(t'2))
             ENDIF)
          (translate[Physical_memory, pm_phy]
            (0, pddb'base_addr, addr', access', segment_to_priv(cs))(x1))
{-8} ∀ (cs: Segment_Reg_type, pddb: Pddb_type, leaf: bool,
      base:
        {b: Memory_Address_4G |
          IF leaf
            THEN aligned?(-1 × (pdir_lvl [Physical_memory, pm_phy] × bits_per_level [Physical
              (offset(b))
            ELSE aligned?(bits_per_level[Physical_memory, pm_phy] + pe_size)
              (offset(b))
            ENDIF}):
      ¬ leaf ⊃
      (∀ (x1: Physical_memory):
        every[Physical_memory, [bool, Address]]
          (λ (x: Physical_memory): TRUE,
           λ (t: [bool, Address]):
             Mem?(t'2'type_of) ∧ 0 ≤ t'2'offset ∧ t'2'offset < max_linear_offset)
          (translate[Physical_memory, pm_phy]
            (1, base, addr', access', segment_to_priv(cs))(x1)))

```

C Proof scripts

Using lemma `xlat_idx_memory_address`,

Using lemma `xlat_idx_memory_address2`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `linear_resolve_unchanged_pm_phy.1.1.3.3.3.1.2.2`.

linear_resolve_unchanged_pm_phy.1.1.3.3.3.2:

```

{-1}  ∀ (s: (pm_phy' states)):
      OK?(translate(0, d''base_addr, addr', access', segment_to_priv(d'))(s)) ∧
      d''' = data(translate(0, d''base_addr, addr', access', segment_to_priv(d'))(s))
{-2}  ∀ (s: (pm_phy' states)):
      OK?(read_data(pm_phy, pdbr_data_type)(PDBR)(s)) ∧
      data(read_data(pm_phy, pdbr_data_type)(PDBR)(s)) = d''
{-3}  ∀ (s: (pm' states)):
      OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s)) ∧
      d' = data(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-4}  ∀ (cs: Segment_Reg_type, pdbr: Pdbr_type):
      ∀ (x1: Physical_memory):
        every[Physical_memory, [bool, Address]]
          (λ (x: Physical_memory): TRUE,
           λ (t: [bool, Address]):
             Mem?(t'2'type_of) ∧
             (0 ≤ t'2'offset ∧ t'2'offset < max_linear_offset) ∧
             IF t'1
               THEN aligned?(22)(offset(t'2))
               ELSE aligned?(12)(offset(t'2))
             ENDIF)
          (translate[Physical_memory, pm_phy]
           (0, pdbr'base_addr, addr', access', segment_to_priv(cs))(x1))
{-5}  ∀ (cs: Segment_Reg_type, pdbr: Pdbr_type, leaf: bool,
      base:
        {b: Memory_Address_4G |
          IF leaf
            THEN aligned?(-1 × (pdir_lvl [Physical_memory, pm_phy] × bits_per_level [Physical
              (offset(b))
            ELSE aligned?(bits_per_level[Physical_memory, pm_phy] + pe_size)
              (offset(b))
            ENDIF}):
      ¬ leaf ⊃
      (∀ (x1: Physical_memory):
        every[Physical_memory, [bool, Address]]
          (λ (x: Physical_memory): TRUE,
           λ (t: [bool, Address]):
             Mem?(t'2'type_of) ∧ 0 ≤ t'2'offset ∧ t'2'offset < max_linear_offset)
          (translate[Physical_memory, pm_phy]
           (1, base, addr', access', segment_to_priv(cs))(x1)))
{-6}  pm'mem = linear_pm
{-7}  ∀ (s: (pm' states)): OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-8}  ∀ (s1, s2: (pm' states)):
      data(read_data(pm_phy, segment_reg_data_type)(CS)(s1)) =
      data(read_data(pm_phy, segment_reg_data_type)(CS)(s2))
{-9}  ∀ (s: (pm' states)): OK?(read_data(pm_phy, pdbr_data_type)(PDBR)(s))
{-10} ∀ (s1, s2: (pm' states)):
      data(read_data(pm_phy, pdbr_data_type)(PDBR)(s1)) =
      data(read_data(pm_phy, pdbr_data_type)(PDBR)(s2))
{-11} ∀ (s1, s2: (pm' states)):
      pe_in_pdir_range?(s1,
        restrict[Address, Memory_Address_4G, boolean]
          ((pm''ro_addr ∪ pm''rw_addr)))
      =
      pe_in_pdir_range?(s2,
        restrict[Address, Memory_Address_4G, boolean]
          ((pm''ro_addr ∪ pm''rw_addr)))
      ∧
      pe_in_ptab_range?(s1,
        restrict[Address, Memory_Address_4G, boolean]
          ((pm''ro_addr ∪ pm''rw_addr)))
      =
      pe_in_ptab_range?(s2,
        restrict[Address, Memory_Address_4G, boolean]
          ((pm''ro_addr ∪ pm''rw_addr)))

```

C Proof scripts

Instantiating quantified variables,

Using lemma `translate_result_no_leaf`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `linear_resolve_unchanged_pm_phy.1.1.3.3.3.2`.

linear_resolve_unchanged_pm_phy.1.1.3.3.4:

```

{-1}  ∀ (s: (pm_phy' states)):
      OK?(read_data(pm_phy, pdbr_data_type)(PDBR)(s)) ∧
      data(read_data(pm_phy, pdbr_data_type)(PDBR)(s)) = d''
{-2}  ∀ (s: (pm' states)):
      OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s)) ∧
      d' = data(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-3}  ∀ (cs: Segment_Reg_type, pdbr: Pdbr_type):
      ∀ (x1: Physical_memory):
        every[Physical_memory, [bool, Address]]
          (λ (x: Physical_memory): TRUE,
           λ (t: [bool, Address]):
             Mem?(t'2'type_of) ∧
             (0 ≤ t'2'offset ∧ t'2'offset < max_linear_offset) ∧
             IF t'1
               THEN aligned?(22)(offset(t'2))
               ELSE aligned?(12)(offset(t'2))
             ENDIF)
          (translate[Physical_memory, pm_phy]
           (0, pdbr'base_addr, addr', access', segment_to_priv(cs))(x1))
{-4}  ∀ (cs: Segment_Reg_type, pdbr: Pdbr_type, leaf: bool,
      base:
        {b: Memory_Address_4G |
          IF leaf
            THEN aligned?(-1 × (pdir_lvl [Physical_memory, pm_phy] × bits_per_level [Physical
              (offset(b))
            ELSE aligned?(bits_per_level[Physical_memory, pm_phy] + pe_size)
              (offset(b))
            ENDIF}):
      ¬ leaf ⊃
      (∀ (x1: Physical_memory):
        every[Physical_memory, [bool, Address]]
          (λ (x: Physical_memory): TRUE,
           λ (t: [bool, Address]):
             Mem?(t'2'type_of) ∧ 0 ≤ t'2'offset ∧ t'2'offset < max_linear_offset)
          (translate[Physical_memory, pm_phy]
           (1, base, addr', access', segment_to_priv(cs))(x1)))
{-5}  pm'mem = linear_pm
{-6}  ∀ (s: (pm' states)): OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-7}  ∀ (s1, s2: (pm' states)):
      data(read_data(pm_phy, segment_reg_data_type)(CS)(s1)) =
      data(read_data(pm_phy, segment_reg_data_type)(CS)(s2))
{-8}  ∀ (s: (pm' states)): OK?(read_data(pm_phy, pdbr_data_type)(PDBR)(s))
{-9}  ∀ (s1, s2: (pm' states)):
      data(read_data(pm_phy, pdbr_data_type)(PDBR)(s1)) =
      data(read_data(pm_phy, pdbr_data_type)(PDBR)(s2))
{-10} ∀ (s1, s2: (pm' states)):
      pe_in_pdir_range?(s1,
        restrict[Address, Memory_Address_4G, boolean]
          ((pm'ro_addr ∪ pm'rw_addr)))
      =
      pe_in_pdir_range?(s2,
        restrict[Address, Memory_Address_4G, boolean]
          ((pm'ro_addr ∪ pm'rw_addr)))
      ∧
      pe_in_ptab_range?(s1,
        restrict[Address, Memory_Address_4G, boolean]
          ((pm'ro_addr ∪ pm'rw_addr)))
      =
      pe_in_ptab_range?(s2,
        restrict[Address, Memory_Address_4G, boolean]
          ((pm'ro_addr ∪ pm'rw_addr)))
{-11} ∀ (s: (pm' states), lvl: Level,
      a:

```

C Proof scripts

Repeatedly Skolemizing and flattening,

Instantiating quantified variables,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Instantiating the top quantifier in -8 with the terms: d!2,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Instantiating quantified variables,

This completes the proof of `linear_resolve_unchanged_pm_phy.1.1.3.3.4`.

linear_resolve_unchanged_pm_phy.1.1.3.4:

```

{-1}  ∀ (s: (pm' states)):
      OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s)) ∧
      d' = data(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-2}  ∀ (cs: Segment_Reg_type, pdir: Pdir_type):
      ∀ (x1: Physical_memory):
        every[Physical_memory, [bool, Address]]
          (λ (x: Physical_memory): TRUE,
           λ (t: [bool, Address]):
             Mem?(t'2'type_of) ∧
             (0 ≤ t'2'offset ∧ t'2'offset < max_linear_offset) ∧
             IF t'1
               THEN aligned?(22)(offset(t'2))
               ELSE aligned?(12)(offset(t'2))
             ENDIF)
          (translate[Physical_memory, pm_phy]
           (0, pdir'base_addr, addr', access', segment_to_priv(cs))(x1))
{-3}  ∀ (cs: Segment_Reg_type, pdir: Pdir_type, leaf: bool,
      base:
        {b: Memory_Address_4G |
          IF leaf
            THEN aligned?(-1 × (pdir_lvl [Physical_memory, pm_phy] × bits_per_level [Physical
              (offset(b))
            ELSE aligned?(bits_per_level[Physical_memory, pm_phy] + pe_size)
              (offset(b))
            ENDIF}):
      ¬ leaf ⊃
      (∀ (x1: Physical_memory):
        every[Physical_memory, [bool, Address]]
          (λ (x: Physical_memory): TRUE,
           λ (t: [bool, Address]):
             Mem?(t'2'type_of) ∧ 0 ≤ t'2'offset ∧ t'2'offset < max_linear_offset)
          (translate[Physical_memory, pm_phy]
           (1, base, addr', access', segment_to_priv(cs))(x1)))
{-4}  pm'mem = linear_pm
{-5}  ∀ (s: (pm' states)): OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-6}  ∀ (s1, s2: (pm' states)):
      data(read_data(pm_phy, segment_reg_data_type)(CS)(s1)) =
      data(read_data(pm_phy, segment_reg_data_type)(CS)(s2))
{-7}  ∀ (s: (pm' states)): OK?(read_data(pm_phy, pdir_data_type)(PDIR)(s))
{-8}  ∀ (s1, s2: (pm' states)):
      data(read_data(pm_phy, pdir_data_type)(PDIR)(s1)) =
      data(read_data(pm_phy, pdir_data_type)(PDIR)(s2))
{-9}  ∀ (s1, s2: (pm' states)):
      pe_in_pdir_range?(s1,
        restrict[Address, Memory_Address_4G, boolean]
          ((pm'ro_addr ∪ pm'rw_addr)))
      =
      pe_in_pdir_range?(s2,
        restrict[Address, Memory_Address_4G, boolean]
          ((pm'ro_addr ∪ pm'rw_addr)))
      ∧
      pe_in_ptab_range?(s1,
        restrict[Address, Memory_Address_4G, boolean]
          ((pm'ro_addr ∪ pm'rw_addr)))
      =
      pe_in_ptab_range?(s2,
        restrict[Address, Memory_Address_4G, boolean]
          ((pm'ro_addr ∪ pm'rw_addr)))
{-10} ∀ (s: (pm' states), lvl: Level,
      a:
        (pe_in_pt_range?(s,
          restrict[Address, Memory_Address_4G, boolean]
            ((pm'ro_addr ∪ pm'rw_addr)),

```


C Proof scripts

Keeping $(-3\ 1)$ and hiding $*$,

Instantiating quantified variables,

This completes the proof of `linear_resolve_unchanged_pm_phy.1.1.3.4`.

linear_resolve_unchanged_pm_phy.1.1.3.5:

```

{-1}  ∀ (s: (pm' states)):
      OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s)) ∧
      d' = data(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-2}  ∀ (cs: Segment_Reg_type, pdir: Pdir_type):
      ∀ (x1: Physical_memory):
        every[Physical_memory, [bool, Address]]
          (λ (x: Physical_memory): TRUE,
           λ (t: [bool, Address]):
             Mem?(t'2'type_of) ∧
             (0 ≤ t'2'offset ∧ t'2'offset < max_linear_offset) ∧
             IF t'1
               THEN aligned?(22)(offset(t'2))
               ELSE aligned?(12)(offset(t'2))
             ENDIF)
          (translate[Physical_memory, pm_phy]
           (0, pdir'base_addr, addr', access', segment_to_priv(cs))(x1))
{-3}  ∀ (cs: Segment_Reg_type, pdir: Pdir_type, leaf: bool,
      base:
        {b: Memory_Address_4G |
          IF leaf
            THEN aligned?(-1 × (pdir_lvl [Physical_memory, pm_phy] × bits_per_level [Physical
              (offset(b))
            ELSE aligned?(bits_per_level[Physical_memory, pm_phy] + pe_size)
              (offset(b))
            ENDIF}):
      ¬ leaf ⊃
      (∀ (x1: Physical_memory):
        every[Physical_memory, [bool, Address]]
          (λ (x: Physical_memory): TRUE,
           λ (t: [bool, Address]):
             Mem?(t'2'type_of) ∧ 0 ≤ t'2'offset ∧ t'2'offset < max_linear_offset)
          (translate[Physical_memory, pm_phy]
           (1, base, addr', access', segment_to_priv(cs))(x1)))
{-4}  pm'mem = linear_pm
{-5}  ∀ (s: (pm' states)): OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-6}  ∀ (s1, s2: (pm' states)):
      data(read_data(pm_phy, segment_reg_data_type)(CS)(s1)) =
      data(read_data(pm_phy, segment_reg_data_type)(CS)(s2))
{-7}  ∀ (s: (pm' states)): OK?(read_data(pm_phy, pdir_data_type)(PDIR)(s))
{-8}  ∀ (s1, s2: (pm' states)):
      data(read_data(pm_phy, pdir_data_type)(PDIR)(s1)) =
      data(read_data(pm_phy, pdir_data_type)(PDIR)(s2))
{-9}  ∀ (s1, s2: (pm' states)):
      pe_in_pdir_range?(s1,
        restrict[Address, Memory_Address_4G, boolean]
          ((pm'ro_addr ∪ pm'rw_addr)))
      =
      pe_in_pdir_range?(s2,
        restrict[Address, Memory_Address_4G, boolean]
          ((pm'ro_addr ∪ pm'rw_addr)))
      ∧
      pe_in_ptab_range?(s1,
        restrict[Address, Memory_Address_4G, boolean]
          ((pm'ro_addr ∪ pm'rw_addr)))
      =
      pe_in_ptab_range?(s2,
        restrict[Address, Memory_Address_4G, boolean]
          ((pm'ro_addr ∪ pm'rw_addr)))
{-10} ∀ (s: (pm' states), lvl: Level,
      a:
        (pe_in_ptab_range?(s,
          restrict[Address, Memory_Address_4G, boolean]
            ((pm'ro_addr ∪ pm'rw_addr)),

```

C Proof scripts

Keeping $(-2\ 1)$ and hiding $*$,

Instantiating quantified variables,

This completes the proof of `linear_resolve_unchanged_pm_phy.1.1.3.5`.

linear_resolve_unchanged_pm_phy.1.2:

```

{-1}  ∀ (cs: Segment_Reg_type, pdbr: Pdbr_type, leaf: bool,
      base:
        {b: Memory_Address_4G |
          IF leaf
            THEN aligned?(-1 × (pdir_lvl [Physical_memory, pm_phy] × bits_per_level [Physical_memory, pm_phy]
              (offset(b))
            ELSE aligned?(bits_per_level[Physical_memory, pm_phy] + pe_size)
              (offset(b))
          ENDIF}):
      ¬ leaf ⊃
      (∀ (x1: Physical_memory):
        every[Physical_memory, [bool, Address]]
          (λ (x: Physical_memory): TRUE,
            λ (t: [bool, Address]):
              Mem?(t'2'type_of) ∧ 0 ≤ t'2'offset ∧ t'2'offset < max_linear_offset)
            (translate[Physical_memory, pm_phy]
              (1, base, addr', access', segment_to_priv(cs))(x1)))
{-2}  pm' mem = linear_pm
{-3}  ∀ (s: (pm' states)): OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-4}  ∀ (s1, s2: (pm' states)):
      data(read_data(pm_phy, segment_reg_data_type)(CS)(s1)) =
        data(read_data(pm_phy, segment_reg_data_type)(CS)(s2))
{-5}  ∀ (s: (pm' states)): OK?(read_data(pm_phy, pdbr_data_type)(PDBR)(s))
{-6}  ∀ (s1, s2: (pm' states)):
      data(read_data(pm_phy, pdbr_data_type)(PDBR)(s1)) =
        data(read_data(pm_phy, pdbr_data_type)(PDBR)(s2))
{-7}  ∀ (s1, s2: (pm' states)):
      pe_in_pdir_range?(s1,
        restrict[Address, Memory_Address_4G, boolean]
          ((pm'ro_addr ∪ pm'rw_addr)))
      =
      pe_in_pdir_range?(s2,
        restrict[Address, Memory_Address_4G, boolean]
          ((pm'ro_addr ∪ pm'rw_addr)))
      ∧
      pe_in_ptab_range?(s1,
        restrict[Address, Memory_Address_4G, boolean]
          ((pm'ro_addr ∪ pm'rw_addr)))
      =
      pe_in_ptab_range?(s2,
        restrict[Address, Memory_Address_4G, boolean]
          ((pm'ro_addr ∪ pm'rw_addr)))
{-8}  ∀ (s: (pm' states), lvl: Level,
      a:
        (pe_in_pt_range?(s,
          restrict[Address, Memory_Address_4G, boolean]
            ((pm'ro_addr ∪ pm'rw_addr)),
          lvl))) :
      OK?(read_data(pm_phy, paging_data_type(lvl))(a)(s))
{-9}  ∀ (s1, s2: (pm' states), lvl: Level,
      a:
        (pe_in_pt_range?(s1,
          restrict[Address, Memory_Address_4G, boolean]
            ((pm'ro_addr ∪ pm'rw_addr)),
          lvl))) :
      set_reference(data(read_data(pm_phy, paging_data_type(lvl))(a)(s1)), Write) 1907
      set_reference(data(read_data(pm_phy, paging_data_type(lvl))(a)(s2)), Write)
{-10} (pm'other_actions ⊆ pm_phy'other_actions)
{-11} (linear_resolve_register_transformers ⊆ pm_phy'other_actions)
{-12} ∀ (a: ((pm'ro_addr ∪ pm'rw_addr))) :
      Mem?(type_of(a)) ∧ 0 ≤ offset(a) ∧ offset(a) < max_linear_offset
{-13} ∀ (s: (pm' states)):
      linear_blessed?(s, restrict[Address, Memory_Address_4G, boolean](pm'ro_addr,

```

C Proof scripts

Hiding formulas: 2,

Repeatedly Skolemizing and flattening,

Expanding the definition of every,

Using lemma `translate_memory_address`,

Using lemma `translate_result_leaf`,

Using lemma `translate_result_no_leaf`,

Using lemma `translate_memory_address2`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `linear_resolve_unchanged_pm_phy.1.2`.

linear_resolve_unchanged_pm_phy.1.3:

```

{-1} pm'`mem = linear_pm
{-2}  $\forall (s: (pm'`states)): OK?(read\_data(pm\_phy, segment\_reg\_data\_type)(CS)(s))$ 
{-3}  $\forall (s_1, s_2: (pm'`states)):$ 
      data(read_data(pm_phy, segment_reg_data_type)(CS)(s1)) =
      data(read_data(pm_phy, segment_reg_data_type)(CS)(s2))
{-4}  $\forall (s: (pm'`states)): OK?(read\_data(pm\_phy, pdir\_data\_type)(PDBR)(s))$ 
{-5}  $\forall (s_1, s_2: (pm'`states)):$ 
      data(read_data(pm_phy, pdir_data_type)(PDBR)(s1)) =
      data(read_data(pm_phy, pdir_data_type)(PDBR)(s2))
{-6}  $\forall (s_1, s_2: (pm'`states)):$ 
      pe_in_pdir_range?(s1,
        restrict[Address, Memory_Address_4G, boolean]
          ((pm'`ro_addr  $\cup$  pm'`rw_addr)))
      =
      pe_in_pdir_range?(s2,
        restrict[Address, Memory_Address_4G, boolean]
          ((pm'`ro_addr  $\cup$  pm'`rw_addr)))
       $\wedge$ 
      pe_in_ptab_range?(s1,
        restrict[Address, Memory_Address_4G, boolean]
          ((pm'`ro_addr  $\cup$  pm'`rw_addr)))
      =
      pe_in_ptab_range?(s2,
        restrict[Address, Memory_Address_4G, boolean]
          ((pm'`ro_addr  $\cup$  pm'`rw_addr)))
{-7}  $\forall (s: (pm'`states), lvl: Level,$ 
      a:
      (pe_in_pt_range?(s,
        restrict[Address, Memory_Address_4G, boolean]
          ((pm'`ro_addr  $\cup$  pm'`rw_addr)),
        lvl))) :
      OK?(read_data(pm_phy, paging_data_type(lvl))(a)(s))
{-8}  $\forall (s_1, s_2: (pm'`states), lvl: Level,$ 
      a:
      (pe_in_pt_range?(s1,
        restrict[Address, Memory_Address_4G, boolean]
          ((pm'`ro_addr  $\cup$  pm'`rw_addr)),
        lvl))) :
      set_reference(data(read_data(pm_phy, paging_data_type(lvl))(a)(s1)), Write) =
      set_reference(data(read_data(pm_phy, paging_data_type(lvl))(a)(s2)), Write)
{-9} (pm'`other_actions  $\subseteq$  pm_phy'`other_actions)
{-10} (linear_resolve_register_transformers  $\subseteq$  pm_phy'`other_actions)
{-11}  $\forall (a: ((pm'`ro\_addr \cup pm'`rw\_addr))) :$ 
      Mem?(type_of(a))  $\wedge$  0  $\leq$  offset(a)  $\wedge$  offset(a) < max_linear_offset
{-12}  $\forall (s: (pm'`states)):$ 
      linear_blessed?(s, restrict[Address, Memory_Address_4G, boolean](pm'`ro_addr),
        restrict[Address, Memory_Address_4G, boolean](pm'`rw_addr))
{-13} pm'`states = pm_phy'`states
{-14} plain_memory?(pm_phy)
{-15} (addresses  $\subseteq$  (pm_phy'`ro_addr  $\cup$  pm_phy'`rw_addr))
{-16} ((pm_phy'`ro_addr  $\cup$  pm_phy'`rw_addr) \ extend [Address, Memory_Address_4G, bool, FALSE](address_in_ptab))
      = addresses
{-17} Mem?(addr'`type_of)
{-18} 0  $\leq$  addr'`offset
{-19} addr'`offset < max_linear_offset
{-20} union(pm'`ro_addr, pm'`rw_addr)(addr')
{-21} is_linear_plain_memory?(pm')
{-22} pm_phy'`states(s')

```

```

{1}  $\forall (cs: Segment_Reg\_type, pdir: Pdir\_type, leaf: bool,$ 
      base:
      {b: Memory_Address_4G |
        IF leaf
          THEN aligned?(-1  $\times$  (pdir_lvl [Physical_memory, pm_phy]  $\times$  bits_per_level [Physical_memory, pm_phy])
            (offset(b)))

```

C Proof scripts

Hiding formulas: 2,

Repeatedly Skolemizing and flattening,

Expanding the definition of every,

Using lemma `translate_memory_address`,

Using lemma `translate_result_leaf`,

Using lemma `translate_result_no_leaf`,

Using lemma `translate_memory_address2`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `linear_resolve_unchanged_pm_phy.1.3`.

linear_resolve_unchanged_pm_phy.1.4:

```

{-1} pm'`mem = linear_pm
{-2}  $\forall (s: (pm'`states)): OK?(read\_data(pm\_phy, segment\_reg\_data\_type)(CS)(s))$ 
{-3}  $\forall (s_1, s_2: (pm'`states)):$ 
    data(read\_data(pm\_phy, segment\_reg\_data\_type)(CS)(s1)) =
    data(read\_data(pm\_phy, segment\_reg\_data\_type)(CS)(s2))
{-4}  $\forall (s: (pm'`states)): OK?(read\_data(pm\_phy, pdir\_data\_type)(PDBR)(s))$ 
{-5}  $\forall (s_1, s_2: (pm'`states)):$ 
    data(read\_data(pm\_phy, pdir\_data\_type)(PDBR)(s1)) =
    data(read\_data(pm\_phy, pdir\_data\_type)(PDBR)(s2))
{-6}  $\forall (s_1, s_2: (pm'`states)):$ 
    pe_in_pdir_range?(s1,
        restrict[Address, Memory_Address_4G, boolean]
            ((pm'`ro\_addr  $\cup$  pm'`rw\_addr)))
    =
    pe_in_pdir_range?(s2,
        restrict[Address, Memory_Address_4G, boolean]
            ((pm'`ro\_addr  $\cup$  pm'`rw\_addr)))
     $\wedge$ 
    pe_in_ptab_range?(s1,
        restrict[Address, Memory_Address_4G, boolean]
            ((pm'`ro\_addr  $\cup$  pm'`rw\_addr)))
    =
    pe_in_ptab_range?(s2,
        restrict[Address, Memory_Address_4G, boolean]
            ((pm'`ro\_addr  $\cup$  pm'`rw\_addr)))
{-7}  $\forall (s: (pm'`states), lvl: Level,$ 
    a:
    (pe_in_pt_range?(s,
        restrict[Address, Memory_Address_4G, boolean]
            ((pm'`ro\_addr  $\cup$  pm'`rw\_addr)),
        lvl))) :
    OK?(read\_data(pm\_phy, paging\_data\_type(lvl))(a)(s))
{-8}  $\forall (s_1, s_2: (pm'`states), lvl: Level,$ 
    a:
    (pe_in_pt_range?(s1,
        restrict[Address, Memory_Address_4G, boolean]
            ((pm'`ro\_addr  $\cup$  pm'`rw\_addr)),
        lvl))) :
    set\_reference(data(read\_data(pm\_phy, paging\_data\_type(lvl))(a)(s1)), Write) =
    set\_reference(data(read\_data(pm\_phy, paging\_data\_type(lvl))(a)(s2)), Write)
{-9} (pm'`other\_actions  $\subseteq$  pm\_phy'`other\_actions)
{-10} (linear\_resolve\_register\_transformers  $\subseteq$  pm\_phy'`other\_actions)
{-11}  $\forall (a: ((pm'`ro\_addr  $\cup$  pm'`rw\_addr))) :$ 
    Mem?(type\_of(a))  $\wedge$  0  $\leq$  offset(a)  $\wedge$  offset(a) < max\_linear\_offset
{-12}  $\forall (s: (pm'`states)):$ 
    linear\_blessed?(s, restrict[Address, Memory_Address_4G, boolean](pm'`ro\_addr),
        restrict[Address, Memory_Address_4G, boolean](pm'`rw\_addr))
{-13} pm'`states = pm\_phy'`states
{-14} plain\_memory?(pm\_phy)
{-15} (addresses  $\subseteq$  (pm\_phy'`ro\_addr  $\cup$  pm\_phy'`rw\_addr))
{-16} ((pm\_phy'`ro\_addr  $\cup$  pm\_phy'`rw\_addr)  $\setminus$  extend [Address, Memory_Address_4G, bool, FALSE](address\_in\_pt
    = addresses
{-17} Mem?(addr'`type\_of)
{-18} 0  $\leq$  addr'`offset
{-19} addr'`offset < max\_linear\_offset
{-20} union(pm'`ro\_addr, pm'`rw\_addr)(addr')
{-21} is\_linear\_plain\_memory?(pm')
{-22} pm\_phy'`states(s')
```

```

{1}  $\forall (cs: Segment\_Reg\_type, pdir: Pdir\_type, x_1: Physical\_memory, t: [bool, Ad-$ 
    dress]):
     $\neg t'1 \wedge 0 \leq t'2'offset \wedge t'2'offset < max\_linear\_offset \wedge Mem?(t'2'type\_of) \supset$ 
    offset(t'2)  $\geq$  0
{2} unchanged\_memory\_invariant?[Physical\_memory]
    (pm\_phy'`mem, pm\_phy'`states
```


C Proof scripts

Repeatedly Skolemizing and flattening,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `linear_resolve_unchanged_pm_phy.1.4`.

linear_resolve_unchanged_pm_phy.1.5:

```

{-1} pm' mem = linear_pm
{-2}  $\forall (s: (pm' states)): OK?(read\_data(pm\_phy, segment\_reg\_data\_type)(CS)(s))$ 
{-3}  $\forall (s_1, s_2: (pm' states)):$ 
      data(read_data(pm_phy, segment_reg_data_type)(CS)(s1)) =
      data(read_data(pm_phy, segment_reg_data_type)(CS)(s2))
{-4}  $\forall (s: (pm' states)): OK?(read\_data(pm\_phy, pdir\_data\_type)(PDBR)(s))$ 
{-5}  $\forall (s_1, s_2: (pm' states)):$ 
      data(read_data(pm_phy, pdir_data_type)(PDBR)(s1)) =
      data(read_data(pm_phy, pdir_data_type)(PDBR)(s2))
{-6}  $\forall (s_1, s_2: (pm' states)):$ 
      pe_in_pdir_range?(s1,
        restrict[Address, Memory_Address_4G, boolean]
          ((pm'ro_addr  $\cup$  pm'rw_addr)))
      =
      pe_in_pdir_range?(s2,
        restrict[Address, Memory_Address_4G, boolean]
          ((pm'ro_addr  $\cup$  pm'rw_addr)))
       $\wedge$ 
      pe_in_ptab_range?(s1,
        restrict[Address, Memory_Address_4G, boolean]
          ((pm'ro_addr  $\cup$  pm'rw_addr)))
      =
      pe_in_ptab_range?(s2,
        restrict[Address, Memory_Address_4G, boolean]
          ((pm'ro_addr  $\cup$  pm'rw_addr)))
{-7}  $\forall (s: (pm' states), lvl: Level,$ 
      a:
      (pe_in_pt_range?(s,
        restrict[Address, Memory_Address_4G, boolean]
          ((pm'ro_addr  $\cup$  pm'rw_addr)),
        lvl))) :
      OK?(read_data(pm_phy, paging_data_type(lvl))(a)(s))
{-8}  $\forall (s_1, s_2: (pm' states), lvl: Level,$ 
      a:
      (pe_in_pt_range?(s1,
        restrict[Address, Memory_Address_4G, boolean]
          ((pm'ro_addr  $\cup$  pm'rw_addr)),
        lvl))) :
      set_reference(data(read_data(pm_phy, paging_data_type(lvl))(a)(s1)), Write) =
      set_reference(data(read_data(pm_phy, paging_data_type(lvl))(a)(s2)), Write)
{-9} (pm'other_actions  $\subseteq$  pm_phy'other_actions)
{-10} (linear_resolve_register_transformers  $\subseteq$  pm_phy'other_actions)
{-11}  $\forall (a: ((pm'ro\_addr \cup pm'rw\_addr))) :$ 
      Mem?(type_of(a))  $\wedge$  0  $\leq$  offset(a)  $\wedge$  offset(a)  $<$  max_linear_offset
{-12}  $\forall (s: (pm' states)):$ 
      linear_blessed?(s, restrict[Address, Memory_Address_4G, boolean](pm'ro_addr),
        restrict[Address, Memory_Address_4G, boolean](pm'rw_addr))
{-13} pm' states = pm_phy' states
{-14} plain_memory?(pm_phy)
{-15} (addresses  $\subseteq$  (pm_phy'ro_addr  $\cup$  pm_phy'rw_addr))
{-16} ((pm_phy'ro_addr  $\cup$  pm_phy'rw_addr)  $\setminus$  extend [Address, Memory_Address_4G, bool, FALSE](address_in_ptab))
      = addresses
{-17} Mem?(addr' type_of)
{-18} 0  $\leq$  addr' offset
{-19} addr' offset  $<$  max_linear_offset
{-20} union(pm'ro_addr, pm'rw_addr)(addr')
{-21} is_linear_plain_memory?(pm')
{-22} pm_phy' states(s')

```

```

{1}  $\forall (cs: Segment\_Reg\_type, pdir: Pdir\_type, x_1: Physical\_memory, t: [bool, Ad-$ 
      dress]):
      t'1  $\wedge$  0  $\leq$  t'2'offset  $\wedge$  t'2'offset  $<$  max_linear_offset  $\wedge$  Mem?(t'2' type_of)  $\supset$ 
      offset(t'2)  $\geq$  0
{2} unchanged_memory_invariant?[Physical_memory]
      (pm_phy' mem, pm_phy' states)

```

C Proof scripts

Repeatedly Skolemizing and flattening,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `linear_resolve_unchanged_pm_phy.1.5`.
`linear_resolve_unchanged_pm_phy.2`:

{-1}	$((\text{pm_phy}'\text{ro_addr} \cup \text{pm_phy}'\text{rw_addr}) \setminus \text{extend} [\text{Address}, \text{Memory_Address_4G}, \text{bool}, \text{FALSE}] (\text{addresses} = \text{addresses}))$
{-2}	$\text{Mem?}(\text{addr}'\text{, type_of})$
{-3}	$0 \leq \text{addr}'\text{'offset}$
{-4}	$\text{addr}'\text{'offset} < \text{max_linear_offset}$
{-5}	$\text{union}(\text{pm}'\text{'ro_addr}, \text{pm}'\text{'rw_addr})(\text{addr}'\text{'})$
{-6}	$\text{is_linear_plain_memory?}(\text{pm}'\text{'})$
{-7}	$\text{pm}'\text{'states}(s')$
{1}	$(\text{addresses} \subseteq (\text{pm_phy}'\text{ro_addr} \cup \text{pm_phy}'\text{rw_addr}))$
{2}	$\text{unchanged_memory_invariant?}[\text{Physical_memory}]$ $(\text{pm_phy}'\text{mem}, \text{pm}'\text{'states},$ $\text{singleton}(\text{expr_2_super}[\text{Physical_memory}, \text{Address}](\text{linear_resolve}(\text{addr}'\text{'}, \text{access}'\text{' })),$ $\text{addresses})$

Hiding formulas: 2,
Replacing using formula -1,
Rewriting using `difference_subset`, matching in `*`,
This completes the proof of `linear_resolve_unchanged_pm_phy.2`.
Q.E.D.

C.117.50

Linear_Memory_Properties.linear_resolve_read_transformers_ok

Terse proof for `linear_resolve_read_transformers_ok`.

`linear_resolve_read_transformers_ok`:

{1}	$\forall (\text{pm}: \text{Plain_Memory}[\text{Linear_memory}], \text{addr}: \text{Memory_Address_4G}):$ $\text{is_linear_plain_memory?}(\text{pm}) \wedge \text{union}(\text{pm}'\text{'ro_addr}, \text{pm}'\text{'rw_addr})(\text{addr}) \supset$ $\text{transformers_ok?}(\text{pm}'\text{'states}, \text{singleton}(\text{expr_2_super}(\text{linear_resolve}(\text{addr}, \text{Read}))))$
-----	--

Installing automatic rewrites from: `(singleton ok_expr_2_super)`
Expanding the definition of `transformers_ok?`,
Repeatedly Skolemizing and flattening,
Expanding the definition of `singleton`,
Replacing using formula -7,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Using lemma `pm_linear_resolve_read_ok`,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `linear_resolve_read_transformers_ok`.
Q.E.D.

C.117.51

Linear_Memory_Properties.linear_resolve_write_transformers_ok

Terse proof for `linear_resolve_write_transformers_ok`.

linear_resolve_write_transformers_ok:

$\{1\} \quad \forall (pm: \text{Plain_Memory}[\text{Linear_memory}], \text{addr}: \text{Memory_Address_4G}):$ $\text{is_linear_plain_memory?}(pm) \wedge pm'rw_addr(\text{addr}) \supset$ $\text{transformers_ok?}(pm'states, \text{singleton}(\text{expr_2_super}(\text{linear_resolve}(\text{addr}, \text{Write}))))$

Installing automatic rewrites from: ok_expr_2_super

Expanding the definition of transformers_ok?,

Repeatedly Skolemizing and flattening,

Expanding the definition of singleton,

Replacing using formula -7,

Using lemma pm_linear_resolve_write_ok,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of linear_resolve_write_transformers_ok.

Q.E.D.

C.117.52

Linear_Memory_Properties.linear_resolve_transformer_invariant

Terse proof for linear_resolve_transformer_invariant.

linear_resolve_transformer_invariant:

$\{1\} \quad \forall (pm: \text{Plain_Memory}[\text{Linear_memory}], \text{addr}: \text{Memory_Address_4G}, \text{access}: \text{Mem-}$ $\text{ory_access}):$ $\text{is_linear_plain_memory?}(pm) \wedge \text{union}(pm'ro_addr, pm'rw_addr)(\text{addr}) \supset$ $\text{transformer_invariant?}(pm'states, \text{singleton}(\text{expr_2_super}(\text{linear_resolve}(\text{addr}, \text{ac-}$ $\text{cess}))))$

Expanding the definition of transformer_invariant?,

Repeatedly Skolemizing and flattening,

Using lemma linear_resolve_unchanged_pm_phy,

Using lemma unchanged_memory_invariant_invariant[Physical_memory],

Using lemma pm_states,

Expanding the definition of transformer_invariant?,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Instantiating quantified variables,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Keeping (-9 1) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

Applying decompose-equality,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of linear_resolve_transformer_invariant.

Q.E.D.

C.117.53 Linear_Memory_Properties.linear_resolve_states_TCC1

Terse proof for linear_resolve_states_TCC1.

`linear_resolve_states_TCC1:`

$\{1\} \quad \forall (pm: \text{Plain_Memory}[\text{Linear_memory}], s: \text{Linear_memory}[\text{Physical_memory}, pm_phy],$ $\quad \text{addr: Memory_Address_4G, access: Memory_access}):$ $\text{union}(pm'ro_addr, pm'rw_addr)(\text{addr}) \wedge$ $\text{is_linear_plain_memory?}(pm) \wedge pm'states(s) \wedge \text{OK?}(\text{linear_resolve}(\text{addr}, \text{access})(s))$ \supset $\text{OK?}[\text{Physical_memory}, \text{Address}]$ $\quad (\text{linear_resolve}[\text{Physical_memory}, pm_phy](\text{addr}, \text{access})(s))$ \vee $\text{Exception?}[\text{Physical_memory}, \text{Address}]$ $\quad (\text{linear_resolve}[\text{Physical_memory}, pm_phy](\text{addr}, \text{access})(s))$
--

Repeatedly Skolemizing and flattening,

This completes the proof of `linear_resolve_states_TCC1`.

Q.E.D.

C.117.54 Linear_Memory_Properties.linear_resolve_states

Terse proof for `linear_resolve_states`.

`linear_resolve_states:`

$\{1\} \quad \forall (pm: \text{Plain_Memory}[\text{Linear_memory}], s: \text{Linear_memory}[\text{Physical_memory}, pm_phy],$ $\quad \text{addr: Memory_Address_4G, access: Memory_access}):$ $\text{union}(pm'ro_addr, pm'rw_addr)(\text{addr}) \wedge$ $\text{is_linear_plain_memory?}(pm) \wedge pm'states(s) \wedge \text{OK?}(\text{linear_resolve}(\text{addr}, \text{access})(s))$ $\supset pm_phy'states(\text{state}(\text{linear_resolve}(\text{addr}, \text{access})(s)))$
--

Installing automatic rewrites from: `has_next_state`

Repeatedly Skolemizing and flattening,

Using lemma `linear_resolve_unchanged_pm_phy`,

Using lemma `pm_states`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Replacing using formula -2,

Using lemma `unchanged_memory_invariant_invariant`,

Using lemma `super_transformer_invariant_next_ok`,

Installing automatic rewrites from: `(expr_2_super! expr_2_super_res!)`

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Expanding the definition of singleton,

which is trivially true.

This completes the proof of `linear_resolve_states`.

Q.E.D.

C.117.55 Linear_Memory_Properties.linear_resolve_same_result

Terse proof for `linear_resolve_same_result`.

linear_resolve_same_result:

<pre> {1} ∀ (pm: Plain_Memory[Linear_memory], addr: Memory_Address_4G, ac1, ac2: Mem- ory_access, s₁, s₂: Linear_memory): union(pm'ro_addr, pm'rw_addr)(addr) ∧ is_linear_plain_memory?(pm) ∧ pm'states(s₁) ∧ pm'states(s₂) ∧ OK?(linear_resolve(addr, ac1)(s₁)) ∧ OK?(linear_resolve(addr, ac2)(s₂)) ⊃ data(linear_resolve(addr, ac1)(s₁)) = data(linear_resolve(addr, ac2)(s₂)) </pre>

Repeatedly Skolemizing and flattening,

Installing automatic rewrites from: has_next_state ## ok_result singleton max_level bus_width bits_per_level Mem min_linear Address_Helpers.<= ptab_lvl pdir_lvl max_linear < pe_size

Case splitting on interpreted_data_type?(pdbr_data_type), interpreted_data_type?(segment_reg_data_type),

we get 3 subgoals:

linear_resolve_same_result.1:

<pre> {-1} interpreted_data_type?(segment_reg_data_type) {-2} interpreted_data_type?(pdbr_data_type) {-3} Mem?(addr' type_of) {-4} 0 ≤ addr' offset {-5} addr' offset < max_linear_offset {-6} union(pm'ro_addr, pm'rw_addr)(addr') {-7} is_linear_plain_memory?(pm') {-8} pm'states(s'₁) {-9} pm'states(s'₂) {-10} OK?(linear_resolve(addr', ac1')(s'₁)) {-11} OK?(linear_resolve(addr', ac2')(s'₂)) </pre> <hr/> <pre> {1} data(linear_resolve(addr', ac1')(s'₁)) = data(linear_resolve(addr', ac2')(s'₂)) </pre>
--

Case splitting on OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s1!1)) AND OK?(read_data(pm_phy, pdbr_data_type)(PDBR) (state(read_data(pm_phy, segment_reg_data_type)(CS) (s1!1)))) AND OK?(translate(0, data(read_data(pm_phy, pdbr_data_type)(PDBR) (state(read_data (pm_phy, segment_reg_data_type) (CS) (s1!1))))'base_addr, addr!1, ac1!1, segment_to_priv(data (read_data (pm_phy, segment_reg_data_type) (CS) (s1!1)))) (state(read_data(pm_phy, pdbr_data_type)(PDBR) (state (read_data (pm_phy, segment_reg_data_type) (CS) (s1!1))))))),

we get 5 subgoals:

linear_resolve_same_result.1.1:

{-1}	$\begin{aligned} & \text{OK?}(\text{read_data}(\text{pm_phy}, \text{segment_reg_data_type})(\text{CS})(s'_1)) \wedge \\ & \text{OK?}(\text{read_data}(\text{pm_phy}, \text{pdbr_data_type})(\text{PDBR}) \\ & \quad (\text{state}(\text{read_data}(\text{pm_phy}, \text{segment_reg_data_type})(\text{CS})(s'_1)))) \\ & \wedge \\ & \text{OK?}(\text{translate}(0, \\ & \quad \text{data}(\text{read_data}(\text{pm_phy}, \text{pdbr_data_type})(\text{PDBR}) \\ & \quad \quad (\text{state}(\text{read_data}(\text{pm_phy}, \text{segment_reg_data_type})(\text{CS}) \\ & \quad \quad \quad (s'_1)))) \text{'base_addr}, \\ & \quad \text{addr}', \text{ac1}', \\ & \quad \text{segment_to_priv}(\text{data}(\text{read_data}(\text{pm_phy}, \text{segment_reg_data_type})(\text{CS}) \\ & \quad \quad \quad (s'_1)))) \\ & \quad (\text{state}(\text{read_data}(\text{pm_phy}, \text{pdbr_data_type})(\text{PDBR}) \\ & \quad \quad (\text{state}(\text{read_data}(\text{pm_phy}, \text{segment_reg_data_type})(\text{CS}) \\ & \quad \quad \quad (s'_1))))) \end{aligned}$
{-2}	interpreted_data_type?(segment_reg_data_type)
{-3}	interpreted_data_type?(pdbr_data_type)
{-4}	Mem?(addr' 'type_of)
{-5}	$0 \leq \text{addr}' \text{'offset}$
{-6}	$\text{addr}' \text{'offset} < \text{max_linear_offset}$
{-7}	union(pm' 'ro_addr, pm' 'rw_addr)(addr')
{-8}	is_linear_plain_memory?(pm')
{-9}	pm' 'states(s'_1)
{-10}	pm' 'states(s'_2)
{-11}	OK?(linear_resolve(addr', ac1')(s'_1))
{-12}	OK?(linear_resolve(addr', ac2')(s'_2))
{1}	data(linear_resolve(addr', ac1')(s'_1)) = data(linear_resolve(addr', ac2')(s'_2))

Case splitting on $\text{OK?}(\text{read_data}(\text{pm_phy}, \text{segment_reg_data_type})(\text{CS})(s2!1)) \text{ AND } \text{OK?}(\text{read_data}(\text{pm_phy}, \text{pdbr_data_type})(\text{PDBR}) (\text{state}(\text{read_data}(\text{pm_phy}, \text{segment_reg_data_type})(\text{CS}) (s2!1)))) \text{ AND } \text{OK?}(\text{translate}(0, \text{data}(\text{read_data}(\text{pm_phy}, \text{pdbr_data_type})(\text{PDBR}) (\text{state}(\text{read_data}(\text{pm_phy}, \text{segment_reg_data_type})(\text{CS}) (s2!1)))) \text{'base_addr}, \text{addr}'1, \text{ac2}'1, \text{segment_to_priv}(\text{data}(\text{read_data}(\text{pm_phy}, \text{segment_reg_data_type})(\text{CS}) (s2!1)))) (\text{state}(\text{read_data}(\text{pm_phy}, \text{pdbr_data_type})(\text{PDBR}) (\text{state}(\text{read_data}(\text{pm_phy}, \text{segment_reg_data_type})(\text{CS}) (s2!1))))))$,

we get 5 subgoals:

linear_resolve_same_result.1.1.1.1:

{-1}	$\begin{aligned} & \text{OK?}(\text{read_data}(\text{pm_phy}, \text{segment_reg_data_type})(\text{CS})(s'_2)) \wedge \\ & \text{OK?}(\text{read_data}(\text{pm_phy}, \text{pdbr_data_type})(\text{PDBR}) \\ & \quad (\text{state}(\text{read_data}(\text{pm_phy}, \text{segment_reg_data_type})(\text{CS})(s'_2)))) \\ & \wedge \\ & \text{OK?}(\text{translate}(0, \\ & \quad \text{data}(\text{read_data}(\text{pm_phy}, \text{pdbr_data_type})(\text{PDBR}) \\ & \quad \quad (\text{state}(\text{read_data}(\text{pm_phy}, \text{segment_reg_data_type})(\text{CS}) \\ & \quad \quad \quad (s'_2)))) \text{'base_addr}, \\ & \quad \text{addr}', \text{ac2}', \\ & \quad \text{segment_to_priv}(\text{data}(\text{read_data}(\text{pm_phy}, \text{segment_reg_data_type})(\text{CS}) \\ & \quad \quad (s'_2)))) \\ & \quad (\text{state}(\text{read_data}(\text{pm_phy}, \text{pdbr_data_type})(\text{PDBR}) \\ & \quad \quad (\text{state}(\text{read_data}(\text{pm_phy}, \text{segment_reg_data_type})(\text{CS}) \\ & \quad \quad \quad (s'_2)))))) \end{aligned}$
{-2}	$\begin{aligned} & \text{OK?}(\text{read_data}(\text{pm_phy}, \text{segment_reg_data_type})(\text{CS})(s'_1)) \wedge \\ & \text{OK?}(\text{read_data}(\text{pm_phy}, \text{pdbr_data_type})(\text{PDBR}) \\ & \quad (\text{state}(\text{read_data}(\text{pm_phy}, \text{segment_reg_data_type})(\text{CS})(s'_1)))) \\ & \wedge \\ & \text{OK?}(\text{translate}(0, \\ & \quad \text{data}(\text{read_data}(\text{pm_phy}, \text{pdbr_data_type})(\text{PDBR}) \\ & \quad \quad (\text{state}(\text{read_data}(\text{pm_phy}, \text{segment_reg_data_type})(\text{CS}) \\ & \quad \quad \quad (s'_1)))) \text{'base_addr}, \\ & \quad \text{addr}', \text{ac1}', \\ & \quad \text{segment_to_priv}(\text{data}(\text{read_data}(\text{pm_phy}, \text{segment_reg_data_type})(\text{CS}) \\ & \quad \quad (s'_1)))) \\ & \quad (\text{state}(\text{read_data}(\text{pm_phy}, \text{pdbr_data_type})(\text{PDBR}) \\ & \quad \quad (\text{state}(\text{read_data}(\text{pm_phy}, \text{segment_reg_data_type})(\text{CS}) \\ & \quad \quad \quad (s'_1)))))) \end{aligned}$
{-3}	$\text{interpreted_data_type?}(\text{segment_reg_data_type})$
{-4}	$\text{interpreted_data_type?}(\text{pdbr_data_type})$
{-5}	$\text{Mem?}(\text{addr}' \text{'type_of})$
{-6}	$0 \leq \text{addr}' \text{'offset}$
{-7}	$\text{addr}' \text{'offset} < \text{max_linear_offset}$
{-8}	$\text{union}(\text{pm}' \text{'ro_addr}, \text{pm}' \text{'rw_addr})(\text{addr}')$
{-9}	$\text{is_linear_plain_memory?}(\text{pm}')$
{-10}	$\text{pm}' \text{'states}(s'_1)$
{-11}	$\text{pm}' \text{'states}(s'_2)$
{-12}	$\text{OK?}(\text{linear_resolve}(\text{addr}', \text{ac1}')(s'_1))$
{-13}	$\text{OK?}(\text{linear_resolve}(\text{addr}', \text{ac2}')(s'_2))$
{1}	$\text{data}(\text{linear_resolve}(\text{addr}', \text{ac1}')(s'_1)) = \text{data}(\text{linear_resolve}(\text{addr}', \text{ac2}')(s'_2))$

Applying disjunctive simplification to flatten sequent,

Hiding formulas: (-16 -17),

Using lemma translate_memory_address,

Using lemma translate_memory_address,

Case splitting on $\text{data}(\text{read_data}(\text{pm_phy}, \text{segment_reg_data_type})(\text{CS})(s1!1)) = \text{data}(\text{read_data}(\text{pm_phy},$

C Proof scripts

```
segment_reg_data_type)(CS)(s2!1)), pm!1'states(state(read_data(pm_phy, segment_reg_data_type)(CS)(s1!1))),  
pm!1'states(state(read_data(pm_phy, segment_reg_data_type)(CS)(s2!1))),
```

we get 4 subgoals:

linear_resolve_same_result.1.1.1.1:

```

{-1} pm' states(state(read_data(pm_phy, segment_reg_data_type)(CS)(s'_2)))
{-2} pm' states(state(read_data(pm_phy, segment_reg_data_type)(CS)(s'_1)))
{-3} data(read_data(pm_phy, segment_reg_data_type)(CS)(s'_1)) =
      data(read_data(pm_phy, segment_reg_data_type)(CS)(s'_2))
{-4} aligned?(bits_per_level + pe_size)
      (offset
        (data(read_data(pm_phy, pdbr_data_type)(PDBR)
              (state(read_data(pm_phy, segment_reg_data_type)(CS)
                        (s'_1))))'base_addr))

      ^
      OK?(translate(0,
                    data(read_data(pm_phy, pdbr_data_type)(PDBR)
                          (state(read_data(pm_phy, seg-
segment_reg_data_type)(CS)
                                (s'_1))))'base_addr,
                    addr', ac1',
                    segment_to_priv(data(read_data(pm_phy, seg-
segment_reg_data_type)(CS)
                                        (s'_1))))
                    (state(read_data(pm_phy, pdbr_data_type)(PDBR)
                          (state(read_data(pm_phy, seg-
segment_reg_data_type)(CS)
                                (s'_1))))))

      ⊃
      data(translate(0,
                    data(read_data(pm_phy, pdbr_data_type)(PDBR)
                          (state(read_data(pm_phy, seg-
segment_reg_data_type)(CS)
                                (s'_1))))'base_addr,
                    addr', ac1',
                    segment_to_priv(data(read_data(pm_phy, seg-
segment_reg_data_type)(CS)
                                        (s'_1))))
                    (state(read_data(pm_phy, pdbr_data_type)(PDBR)
                          (state(read_data(pm_phy, seg-
segment_reg_data_type)(CS)
                                (s'_1))))))'2

      < max_linear
{-5} aligned?(bits_per_level + pe_size)
      (offset
        (data(read_data(pm_phy, pdbr_data_type)(PDBR)
              (state(read_data(pm_phy, segment_reg_data_type)(CS)
                        (s'_2))))'base_addr))

      ^
      OK?(translate(0,
                    data(read_data(pm_phy, pdbr_data_type)(PDBR)
                          (state(read_data(pm_phy, seg-
segment_reg_data_type)(CS)
                                (s'_2))))'base_addr,
                    addr', ac2',
                    segment_to_priv(data(read_data(pm_phy, seg-
segment_reg_data_type)(CS)
                                        (s'_2))))
                    (state(read_data(pm_phy, pdbr_data_type)(PDBR)
                          (state(read_data(pm_phy, seg-
segment_reg_data_type)(CS)
                                (s'_2))))))

      ⊃
      data(translate(0,
                    data(read_data(pm_phy, pdbr_data_type)(PDBR)
                          (state(read_data(pm_phy, seg-
segment_reg_data_type)(CS)
                                (s'_2))))'base_addr,

```

C Proof scripts

Case splitting on $\text{data}(\text{read_data}(\text{pm_phy}, \text{pdbl_data_type})(\text{PDBR}) (\text{state}(\text{read_data}(\text{pm_phy}, \text{segment_reg_data_type})(\text{CS}) (\text{s1!1}))))' \text{base_addr} = \text{data}(\text{read_data}(\text{pm_phy}, \text{pdbl_data_type})(\text{PDBR}) (\text{state}(\text{read_data}(\text{pm_phy}, \text{segment_reg_data_type})(\text{CS}) (\text{s2!1}))))' \text{base_addr}, \text{pm!1}' \text{states} (\text{state}(\text{read_data}(\text{pm_phy}, \text{pdbl_data_type})(\text{PDBR}) (\text{state}(\text{read_data}(\text{pm_phy}, \text{segment_reg_data_type})(\text{CS}) (\text{s1!1}))))), \text{pm!1}' \text{states} (\text{state}(\text{read_data}(\text{pm_phy}, \text{pdbl_data_type})(\text{PDBR}) (\text{state}(\text{read_data}(\text{pm_phy}, \text{segment_reg_data_type})(\text{CS}) (\text{s2!1}))))),$

we get 4 subgoals:

linear_resolve_same_result.1.1.1.1.1:

```

{-1} pm' 'states
      (state(read_data(pm_phy, pdbrr_data_type)(PDBR)
              (state(read_data(pm_phy, segment_reg_data_type)(CS)(s'_2))))))
{-2} pm' 'states
      (state(read_data(pm_phy, pdbrr_data_type)(PDBR)
              (state(read_data(pm_phy, segment_reg_data_type)(CS)(s'_1))))))
{-3} data(read_data(pm_phy, pdbrr_data_type)(PDBR)
          (state(read_data(pm_phy, segment_reg_data_type)(CS)(s'_1)))) 'base_addr
      =
      data(read_data(pm_phy, pdbrr_data_type)(PDBR)
          (state(read_data(pm_phy, segment_reg_data_type)(CS)(s'_2)))) 'base_addr
{-4} pm' 'states(state(read_data(pm_phy, segment_reg_data_type)(CS)(s'_2)))
{-5} pm' 'states(state(read_data(pm_phy, segment_reg_data_type)(CS)(s'_1)))
{-6} data(read_data(pm_phy, segment_reg_data_type)(CS)(s'_1)) =
      data(read_data(pm_phy, segment_reg_data_type)(CS)(s'_2))
{-7} aligned?(bits_per_level + pe_size)
      (offset
        (data(read_data(pm_phy, pdbrr_data_type)(PDBR)
              (state(read_data(pm_phy, segment_reg_data_type)(CS)
                        (s'_1)))) 'base_addr))

      ^
      OK?(translate(0,
                    data(read_data(pm_phy, pdbrr_data_type)(PDBR)
                          (state(read_data(pm_phy, seg-
segment_reg_data_type)(CS)
                                (s'_1)))) 'base_addr,
                    addr', ac1',
                    segment_to_priv(data(read_data(pm_phy, seg-
segment_reg_data_type)(CS)
                                (s'_1))))
                    (state(read_data(pm_phy, pdbrr_data_type)(PDBR)
                          (state(read_data(pm_phy, seg-
segment_reg_data_type)(CS)
                                (s'_1))))))

      ⊃
      data(translate(0,
                    data(read_data(pm_phy, pdbrr_data_type)(PDBR)
                          (state(read_data(pm_phy, seg-
segment_reg_data_type)(CS)
                                (s'_1)))) 'base_addr,
                    addr', ac1',
                    segment_to_priv(data(read_data(pm_phy, seg-
segment_reg_data_type)(CS)
                                (s'_1))))
                    (state(read_data(pm_phy, pdbrr_data_type)(PDBR)
                          (state(read_data(pm_phy, seg-
segment_reg_data_type)(CS)
                                (s'_1)))))) '2

      < max_linear
{-8} aligned?(bits_per_level + pe_size)
      (offset
        (data(read_data(pm_phy, pdbrr_data_type)(PDBR)
              (state(read_data(pm_phy, segment_reg_data_type)(CS)
                        (s'_2)))) 'base_addr))

      ^
      OK?(translate(0,
                    data(read_data(pm_phy, pdbrr_data_type)(PDBR)
                          (state(read_data(pm_phy, seg-
segment_reg_data_type)(CS)
                                (s'_2)))) 'base_addr,
                    addr', ac2',
                    segment_to_priv(data(read_data(pm_phy, seg-
segment_reg_data_type)(CS)
                                (s'_1))))

```

C Proof scripts

Case splitting on $\text{data}(\text{translate}(0, \text{data}(\text{read_data}(\text{pm_phy}, \text{pdr_data_type})(\text{PDR}) (\text{state}(\text{read_data}(\text{pm_phy}, \text{segment_reg_data_type}) (\text{CS}) (s2!1)))) \text{'base_addr, addr!1, ac2!1, segment_to_priv}(\text{data}(\text{read_data}(\text{pm_phy}, \text{segment_reg_data_type}) (\text{CS}) (s2!1)))) (\text{state}(\text{read_data}(\text{pm_phy}, \text{pdr_data_type})(\text{PDR}) (\text{state}(\text{read_data}(\text{pm_phy}, \text{segment_reg_data_type}) (\text{CS}) (s2!1)))))) = \text{data}(\text{translate}(0, \text{data}(\text{read_data}(\text{pm_phy}, \text{pdr_data_type})(\text{PDR}) (\text{state}(\text{read_data}(\text{pm_phy}, \text{segment_reg_data_type}) (\text{CS}) (s1!1)))) \text{'base_addr, addr!1, ac1!1, segment_to_priv}(\text{data}(\text{read_data}(\text{pm_phy}, \text{segment_reg_data_type}) (\text{CS}) (s1!1)))) (\text{state}(\text{read_data}(\text{pm_phy}, \text{pdr_data_type})(\text{PDR}) (\text{state}(\text{read_data}(\text{pm_phy}, \text{segment_reg_data_type}) (\text{CS}) (s1!1))))))$,

we get 4 subgoals:

linear_resolve_same_result.1.1.1.1.1.1:

```

{-1} data(translate(0,
                    data(read_data(pm_phy, pddb_data_type)(PDBR)
                        (state(read_data(pm_phy, seg-
ment_reg_data_type)(CS)
                                (s'_2)))) 'base_addr,
                    addr', ac2',
                    segment_to_priv(data(read_data(pm_phy, seg-
ment_reg_data_type)(CS)
                                (s'_2))))
                    (state(read_data(pm_phy, pddb_data_type)(PDBR)
                        (state(read_data(pm_phy, seg-
ment_reg_data_type)(CS)
                                (s'_2))))))
                    (s'_2))))))
=
data(translate(0,
                    data(read_data(pm_phy, pddb_data_type)(PDBR)
                        (state(read_data(pm_phy, seg-
ment_reg_data_type)(CS)
                                (s'_1)))) 'base_addr,
                    addr', ac1',
                    segment_to_priv(data(read_data(pm_phy, seg-
ment_reg_data_type)(CS)
                                (s'_1))))
                    (state(read_data(pm_phy, pddb_data_type)(PDBR)
                        (state(read_data(pm_phy, seg-
ment_reg_data_type)(CS)
                                (s'_1))))))
                    (s'_1))))))
{-2} pm' 'states
      (state(read_data(pm_phy, pddb_data_type)(PDBR)
              (state(read_data(pm_phy, segment_reg_data_type)(CS)(s'_2))))))
{-3} pm' 'states
      (state(read_data(pm_phy, pddb_data_type)(PDBR)
              (state(read_data(pm_phy, segment_reg_data_type)(CS)(s'_1))))))
{-4} data(read_data(pm_phy, pddb_data_type)(PDBR)
          (state(read_data(pm_phy, segment_reg_data_type)(CS)(s'_1)))) 'base_addr
=
data(read_data(pm_phy, pddb_data_type)(PDBR)
      (state(read_data(pm_phy, segment_reg_data_type)(CS)(s'_2)))) 'base_addr
{-5} pm' 'states(state(read_data(pm_phy, segment_reg_data_type)(CS)(s'_2))))
{-6} pm' 'states(state(read_data(pm_phy, segment_reg_data_type)(CS)(s'_1))))
{-7} data(read_data(pm_phy, segment_reg_data_type)(CS)(s'_1)) =
      data(read_data(pm_phy, segment_reg_data_type)(CS)(s'_2))
{-8} aligned?(bits_per_level + pe_size)
      (offset
        (data(read_data(pm_phy, pddb_data_type)(PDBR)
              (state(read_data(pm_phy, segment_reg_data_type)(CS)
                        (s'_1)))) 'base_addr))
        ^
        OK?(translate(0,
                      data(read_data(pm_phy, pddb_data_type)(PDBR)
                          (state(read_data(pm_phy, seg-
ment_reg_data_type)(CS)
                                  (s'_1)))) 'base_addr,
                      addr', ac1',
                      segment_to_priv(data(read_data(pm_phy, seg-
ment_reg_data_type)(CS)
                                  (s'_1))))
                      (state(read_data(pm_phy, pddb_data_type)(PDBR)
                          (state(read_data(pm_phy, seg-
ment_reg_data_type)(CS)
                                  (s'_1))))))
                      (s'_1))))))
      ⊃
data(translate(0,
              data(read_data(pm_phy, pddb_data_type)(PDBR)
                  (state(read_data(pm_phy, seg-
ment_reg_data_type)(CS)
                          (s'_1)))) 'base_addr,
              addr', ac1',
              segment_to_priv(data(read_data(pm_phy, seg-
ment_reg_data_type)(CS)
                          (s'_1))))
              (state(read_data(pm_phy, pddb_data_type)(PDBR)
                  (state(read_data(pm_phy, seg-
ment_reg_data_type)(CS)
                          (s'_1))))))
              (s'_1))))))

```

C Proof scripts

Case splitting on `data(translate(0, data(read_data(pm_phy, pabr_data_type)(PDBR) (state(read_data (pm_phy, segment_reg_data_type) (CS) (s1!1))))'base_addr, addr!1, ac1!1, segment_to_priv(data(read_data (pm_phy, segment_reg_data_type) (CS) (s1!1)))) (state(read_data(pm_phy, pabr_data_type)(PDBR) (state(read_data (pm_phy, segment_reg_data_type) (CS) (s1!1))))))'1,`

we get 2 subgoals:

linear_resolve_same_result.1.1.1.1.1.1.1:

```

{-1} data(translate(0,
                data(read_data(pm_phy, pddb_data_type)(PDBR)
                    (state(read_data(pm_phy, seg-
ment_reg_data_type)(CS)
                                (s'_1)))) 'base_addr,
                addr', ac1',
                segment_to_priv(data(read_data(pm_phy, seg-
ment_reg_data_type)(CS)
                                (s'_1))))
                (state(read_data(pm_phy, pddb_data_type)(PDBR)
                    (state(read_data(pm_phy, seg-
ment_reg_data_type)(CS)
                                (s'_1)))))) '1
{-2} data(translate(0,
                data(read_data(pm_phy, pddb_data_type)(PDBR)
                    (state(read_data(pm_phy, seg-
ment_reg_data_type)(CS)
                                (s'_2)))) 'base_addr,
                addr', ac2',
                segment_to_priv(data(read_data(pm_phy, seg-
ment_reg_data_type)(CS)
                                (s'_2))))
                (state(read_data(pm_phy, pddb_data_type)(PDBR)
                    (state(read_data(pm_phy, seg-
ment_reg_data_type)(CS)
                                (s'_2))))))
                (s'_2))))))
=
data(translate(0,
                data(read_data(pm_phy, pddb_data_type)(PDBR)
                    (state(read_data(pm_phy, seg-
ment_reg_data_type)(CS)
                                (s'_1)))) 'base_addr,
                addr', ac1',
                segment_to_priv(data(read_data(pm_phy, seg-
ment_reg_data_type)(CS)
                                (s'_1))))
                (state(read_data(pm_phy, pddb_data_type)(PDBR)
                    (state(read_data(pm_phy, seg-
ment_reg_data_type)(CS)
                                (s'_1))))))
                (s'_1))))))
{-3} pm' 'states
      (state(read_data(pm_phy, pddb_data_type)(PDBR)
              (state(read_data(pm_phy, segment_reg_data_type)(CS)(s'_2))))))
{-4} pm' 'states
      (state(read_data(pm_phy, pddb_data_type)(PDBR)
              (state(read_data(pm_phy, segment_reg_data_type)(CS)(s'_1))))))
{-5} data(read_data(pm_phy, pddb_data_type)(PDBR)
          (state(read_data(pm_phy, segment_reg_data_type)(CS)(s'_1)))) 'base_addr
=
data(read_data(pm_phy, pddb_data_type)(PDBR)
      (state(read_data(pm_phy, segment_reg_data_type)(CS)(s'_2)))) 'base_addr
{-6} pm' 'states(state(read_data(pm_phy, segment_reg_data_type)(CS)(s'_2)))
{-7} pm' 'states(state(read_data(pm_phy, segment_reg_data_type)(CS)(s'_1)))
{-8} data(read_data(pm_phy, segment_reg_data_type)(CS)(s'_1)) =
      data(read_data(pm_phy, segment_reg_data_type)(CS)(s'_2))
{-9} aligned?(bits_per_level + pe_size)
      (offset
        (data(read_data(pm_phy, pddb_data_type)(PDBR)
              (state(read_data(pm_phy, segment_reg_data_type)(CS)
                        (s'_1)))) 'base_addr))
      )
^
OK?(translate(0,
              data(read_data(pm_phy, pddb_data_type)(PDBR)
                  (state(read_data(pm_phy, seg-

```


C Proof scripts

Expanding the definition of `linear_resolve`,

Expanding the definition of `##`,

Simplifying, rewriting, and recording with decision procedures,

Replacing using formula -2,

Replacing using formula -1,

which is trivially true.

This completes the proof of `linear_resolve_same_result.1.1.1.1.1.1.1`.

linear_resolve_same_result.1.1.1.1.1.1.2:

```

{-1} data(translate(0,
                data(read_data(pm_phy, pddb_data_type)(PDBR)
                    (state(read_data(pm_phy, seg-
ment_reg_data_type)(CS)
                                (s'_2)))) 'base_addr,
                addr', ac2',
                segment_to_priv(data(read_data(pm_phy, seg-
ment_reg_data_type)(CS)
                                (s'_2))))
                (state(read_data(pm_phy, pddb_data_type)(PDBR)
                    (state(read_data(pm_phy, seg-
ment_reg_data_type)(CS)
                                (s'_2))))))
                (s'_2))))))
=
data(translate(0,
                data(read_data(pm_phy, pddb_data_type)(PDBR)
                    (state(read_data(pm_phy, seg-
ment_reg_data_type)(CS)
                                (s'_1)))) 'base_addr,
                addr', ac1',
                segment_to_priv(data(read_data(pm_phy, seg-
ment_reg_data_type)(CS)
                                (s'_1))))
                (state(read_data(pm_phy, pddb_data_type)(PDBR)
                    (state(read_data(pm_phy, seg-
ment_reg_data_type)(CS)
                                (s'_1))))))
                (s'_1))))))
{-2} pm' 'states
      (state(read_data(pm_phy, pddb_data_type)(PDBR)
              (state(read_data(pm_phy, segment_reg_data_type)(CS)(s'_2))))))
{-3} pm' 'states
      (state(read_data(pm_phy, pddb_data_type)(PDBR)
              (state(read_data(pm_phy, segment_reg_data_type)(CS)(s'_1))))))
{-4} data(read_data(pm_phy, pddb_data_type)(PDBR)
          (state(read_data(pm_phy, segment_reg_data_type)(CS)(s'_1)))) 'base_addr
=
data(read_data(pm_phy, pddb_data_type)(PDBR)
      (state(read_data(pm_phy, segment_reg_data_type)(CS)(s'_2)))) 'base_addr
{-5} pm' 'states(state(read_data(pm_phy, segment_reg_data_type)(CS)(s'_2)))
{-6} pm' 'states(state(read_data(pm_phy, segment_reg_data_type)(CS)(s'_1)))
{-7} data(read_data(pm_phy, segment_reg_data_type)(CS)(s'_1)) =
      data(read_data(pm_phy, segment_reg_data_type)(CS)(s'_2))
{-8} aligned?(bits_per_level + pe_size)
      (offset
        (data(read_data(pm_phy, pddb_data_type)(PDBR)
              (state(read_data(pm_phy, segment_reg_data_type)(CS)
                          (s'_1)))) 'base_addr))
      ^
      OK?(translate(0,
                    data(read_data(pm_phy, pddb_data_type)(PDBR)
                        (state(read_data(pm_phy, seg-
ment_reg_data_type)(CS)
                            (s'_1)))) 'base_addr,
                    addr', ac1',
                    segment_to_priv(data(read_data(pm_phy, seg-
ment_reg_data_type)(CS)
                            (s'_1))))
                    (state(read_data(pm_phy, pddb_data_type)(PDBR)
                        (state(read_data(pm_phy, seg-
ment_reg_data_type)(CS)
                            (s'_1))))))
                    (s'_1))))))
      ⊃
data(translate(0,

```

C Proof scripts

Expanding the definition of `linear_resolve`,

Expanding the definition of `##`,

Simplifying, rewriting, and recording with decision procedures,

Replacing using formula -1,

Replacing using formula 1,

Case splitting on $\text{OK?}(\text{translate}(1, \text{data}(\text{translate}(0, \text{data}(\text{read_data}(\text{pm_phy}, \text{pdbl_data_type})$
(PDBR) (state (read_data (pm_phy, segment_reg_data_type) (CS) (s1!1))))'base_addr, addr!1, ac1!1,
segment_to_priv(data (read_data (pm_phy, segment_reg_data_type) (CS) (s1!1)))) (state(read_data
(pm_phy, pdbr_data_type) (PDBR) (state (read_data (pm_phy, segment_reg_data_type) (CS)
(s1!1))))))'2, addr!1, ac1!1, segment_to_priv(data(read_data (pm_phy, segment_reg_data_type)
(CS) (s1!1)))) (state(translate(0, data(read_data (pm_phy, pdbr_data_type) (PDBR) (state
(read_data (pm_phy, segment_reg_data_type) (CS) (s1!1))))'base_addr, addr!1, ac1!1, seg-
ment_to_priv(data (read_data (pm_phy, segment_reg_data_type) (CS) (s1!1)))) (state(read_data
(pm_phy, pdbr_data_type) (PDBR) (state (read_data (pm_phy, segment_reg_data_type) (CS)
(s1!1))))))))) , $\text{OK?}(\text{translate}(1, \text{data}(\text{translate}(0, \text{data}(\text{read_data}(\text{pm_phy}, \text{pdbl_data_type})$
(PDBR) (state (read_data (pm_phy, segment_reg_data_type) (CS) (s1!1))))'base_addr, addr!1, ac1!1, seg-
ment_to_priv(data (read_data (pm_phy, segment_reg_data_type) (CS) (s1!1)))) (state(read_data
(pm_phy, pdbr_data_type) (PDBR) (state (read_data (pm_phy, segment_reg_data_type) (CS)
(s1!1))))))'2, addr!1, ac2!1, segment_to_priv(data(read_data (pm_phy, segment_reg_data_type) (CS)
(s2!1)))) (state(translate(0, data(read_data (pm_phy, pdbr_data_type) (PDBR) (state (read_data
(pm_phy, segment_reg_data_type) (CS) (s2!1))))'base_addr, addr!1, ac2!1, segment_to_priv(data
(read_data (pm_phy, segment_reg_data_type) (CS) (s2!1)))) (state(read_data (pm_phy, pdbr_data_type)
(PDBR) (state (read_data (pm_phy, segment_reg_data_type) (CS) (s2!1))))))))) ,

we get 3 subgoals:

C.117 Proofs for Linear_Memory_Properties (challenge-linear.pvs)

linear_resolve_same_result.1.1.1.1.1.1.2.1:

```

{-1} OK?(translate(1,
                data(translate(0,
                                data(read_data(pm_phy, pdbr_data_type)(PDBR)
                                    (state(read_data(pm_phy, seg-
ment_reg_data_type)
                                                (CS)
                                                (s'1)))) 'base_addr,
                                addr', ac1',
                                segment_to_priv(data(read_data(pm_phy, seg-
ment_reg_data_type)
                                                (CS)
                                                (s'1))))
                                (state(read_data(pm_phy, pdbr_data_type)(PDBR)
                                    (state(read_data(pm_phy, seg-
ment_reg_data_type)
                                                (CS)
                                                (s'1)))))) '2,
                                addr', ac2',
                                segment_to_priv(data(read_data(pm_phy, segment_reg_data_type)(CS)
                                                        (s'2))))
                                (state(translate(0,
                                                data(read_data(pm_phy, pdbr_data_type)(PDBR)
                                                    (state(read_data(pm_phy, seg-
ment_reg_data_type)
                                                                (CS)
                                                                (s'2)))) 'base_addr,
                                                addr', ac2',
                                                segment_to_priv(data(read_data(pm_phy, seg-
ment_reg_data_type)
                                                                (CS)
                                                                (s'2))))
                                                (state(read_data(pm_phy, pdbr_data_type)(PDBR)
                                                    (state(read_data(pm_phy,
segment_reg_data_type)
                                                                (CS)
                                                                (s'2))))))))))
{-2} OK?(translate(1,
                data(translate(0,
                                data(read_data(pm_phy, pdbr_data_type)(PDBR)
                                    (state(read_data(pm_phy, seg-
ment_reg_data_type)
                                                (CS)
                                                (s'1)))) 'base_addr,
                                addr', ac1',
                                segment_to_priv(data(read_data(pm_phy, seg-
ment_reg_data_type)
                                                (CS)
                                                (s'1))))
                                (state(read_data(pm_phy, pdbr_data_type)(PDBR)
                                    (state(read_data(pm_phy, seg-
ment_reg_data_type)
                                                (CS)
                                                (s'1)))))) '2,
                                addr', ac1',
                                segment_to_priv(data(read_data(pm_phy, segment_reg_data_type)(CS)
                                                        (s'1))))
                                (state(translate(0,
                                                data(read_data(pm_phy, pdbr_data_type)(PDBR)
                                                    (state(read_data(pm_phy, seg-
ment_reg_data_type)
                                                                (CS)
                                                                (s'1)))) 'base_addr,
                                                addr', ac1',
                                                segment_to_priv(data(read_data(pm_phy, segment_reg_data_type)(CS)
                                                        (s'1))))
                                                (state(translate(0,
                                                data(read_data(pm_phy, pdbr_data_type)(PDBR)
                                                    (state(read_data(pm_phy, seg-
ment_reg_data_type)
                                                                (CS)
                                                                (s'1)))) 'base_addr,

```

C Proof scripts

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Replacing using formula -6,

Replacing using formula -9,

```
Case splitting on pm!1'states (state(translate(0, data(read_data(pm_phy, pdbr_data_type)
(PDBR) (state (read_data (pm_phy, segment_reg_data_type) (CS) (s1!1))))'base_addr, addr!1,
ac1!1, segment_to_priv(data (read_data (pm_phy, segment_reg_data_type) (CS) (s1!1))))
(state(read_data(pm_phy, pdbr_data_type) (PDBR) (state (read_data (pm_phy, segment_reg_data_type)
(CS) (s1!1))))))), pm!1'states (state(translate(0, data(read_data(pm_phy, pdbr_data_type)
(PDBR) (state (read_data (pm_phy, segment_reg_data_type) (CS) (s1!1))))'base_addr, addr!1,
ac2!1, segment_to_priv(data (read_data (pm_phy, segment_reg_data_type) (CS) (s1!1))))
(state(read_data(pm_phy, pdbr_data_type) (PDBR) (state (read_data (pm_phy, segment_reg_data_type)
(CS) (s2!1))))))),
```

we get 3 subgoals:

C.117 Proofs for Linear_Memory_Properties (challenge-linear.pvs)

linear_resolve_same_result.1.1.1.1.1.1.2.1.1:

```

{-1} pm' 'states
      (state(translate(0,
                      data(read_data(pm_phy, pdb_r_data_type)(PDBR)
                          (state(read_data(pm_phy, seg-
ment_reg_data_type)(CS)
                              (s'_1)))))) 'base_addr,
      addr', ac2',
      segment_to_priv(data(read_data(pm_phy, seg-
ment_reg_data_type)(CS)
                          (s'_1))))
      (state(read_data(pm_phy, pdb_r_data_type)(PDBR)
              (state(read_data(pm_phy, seg-
ment_reg_data_type)(CS)
                          (s'_2)))))))
{-2} pm' 'states
      (state(translate(0,
                      data(read_data(pm_phy, pdb_r_data_type)(PDBR)
                          (state(read_data(pm_phy, seg-
ment_reg_data_type)(CS)
                              (s'_1)))))) 'base_addr,
      addr', ac1',
      segment_to_priv(data(read_data(pm_phy, seg-
ment_reg_data_type)(CS)
                          (s'_1))))
      (state(read_data(pm_phy, pdb_r_data_type)(PDBR)
              (state(read_data(pm_phy, seg-
ment_reg_data_type)(CS)
                          (s'_1)))))))
{-3} OK?(translate(1,
                  data(translate(0,
                                data(read_data(pm_phy, pdb_r_data_type)(PDBR)
                                    (state(read_data(pm_phy, seg-
ment_reg_data_type)
                                                (CS)
                                                (s'_1)))))) 'base_addr,
                                addr', ac1',
                                segment_to_priv(data(read_data(pm_phy, seg-
ment_reg_data_type)
                                                        (CS)
                                                        (s'_1))))
                                (state(read_data(pm_phy, pdb_r_data_type)(PDBR)
                                        (state(read_data(pm_phy, seg-
ment_reg_data_type)
                                                    (CS)
                                                    (s'_1)))))) '2,
                                addr', ac2',
                                segment_to_priv(data(read_data(pm_phy, segment_reg_data_type)(CS)
                                                        (s'_1))))
                                (state(translate(0,
                                                data(read_data(pm_phy, pdb_r_data_type)(PDBR)
                                                    (state(read_data(pm_phy, seg-
ment_reg_data_type)
                                                                (CS)
                                                                (s'_1)))))) 'base_addr,
                                                addr', ac2',
                                                segment_to_priv(data(read_data(pm_phy, seg-
ment_reg_data_type)
                                                                    (CS)
                                                                    (s'_1))))
                                                (state(read_data(pm_phy, pdb_r_data_type)(PDBR)
                                                        (state(read_data(pm_phy, seg-
ment_reg_data_type)
                                                                    (CS)
                                                                    (s'_1))))))

```

C Proof scripts

Using lemma `translate_same_result`,

we get 2 subgoals:

C.117 Proofs for Linear_Memory_Properties (challenge-linear.pvs)

linear_resolve_same_result.1.1.1.1.1.1.2.1.1.1:

```

{-1} union(pm'ro_addr, pm'rw_addr)(addr') ∧
      is_linear_plain_memory?(pm') ∧
      pe_in_pt_range?(state(translate(0,
                                     data(read_data(pm_phy, pabr_data_type)(PDBR)
                                               (state(read_data
                                                       (pm_phy, seg-
ment_reg_data_type)
                                               (CS)
                                               (s'1))))'base_addr,
                                     addr', ac1',
                                     segment_to_priv(data(read_data
                                                           (pm_phy, seg-
ment_reg_data_type)
                                                           (CS)
                                                           (s'1))))
                                     (state(read_data(pm_phy, pabr_data_type)(PDBR)
                                               (state(read_data
                                                       (pm_phy, seg-
ment_reg_data_type)
                                               (CS)
                                               (s'1))))),
                                     restrict[Address, Memory_Address_4G, boolean]
                                     ((pm'ro_addr ∪ pm'rw_addr)),
                                     1)
      (xlat_idx(1,
                data(translate(0,
                               data(read_data(pm_phy, pabr_data_type)
                                         (PDBR)
                                         (state
                                          (read_data
                                           (pm_phy, seg-
ment_reg_data_type)
                                         (CS)
                                         (s'1))))'base_addr,
                               addr', ac1',
                               segment_to_priv(data
                                                (read_data
                                                 (pm_phy, seg-
ment_reg_data_type)
                                                (CS)
                                                (s'1))))
                               (state(read_data(pm_phy, pabr_data_type)
                                         (PDBR)
                                         (state
                                          (read_data
                                           (pm_phy, seg-
ment_reg_data_type)
                                         (CS)
                                         (s'1))))))'2,
                addr'))
      ∧
      OK?(translate(1,
                    data(translate(0,
                                   data(read_data(pm_phy, pabr_data_type)(PDBR)
                                               (state(read_data
                                                       (pm_phy, seg-
ment_reg_data_type)
                                               (CS)
                                               (s'1))))'base_addr,
                                   addr', ac1',
                                   segment_to_priv(data(read_data(pm_phy,
segment_reg_data_type)
segment_reg_data_type)
segment_reg_data_type)

```


C Proof scripts

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 2 subgoals:

C.117 Proofs for Linear_Memory_Properties (challenge-linear.pvs)

linear_resolve_same_result.1.1.1.1.1.1.2.1.1.1.1:

```

{-1} union(pm' 'ro_addr, pm' 'rw_addr)(addr')
{-2} is_linear_plain_memory?(pm')
{-3} OK?(translate(1,
                    data(translate(0,
                                    data(read_data(pm_phy, pddb_data_type)(PDBR)
                                        (state(read_data(pm_phy, seg-
ment_reg_data_type)
                                                (CS)
                                                (s'1)))) 'base_addr,
                                        addr', ac1',
segment_to_priv(data(read_data(pm_phy, seg-
ment_reg_data_type)
                                                (CS)
                                                (s'1))))
                    (state(read_data(pm_phy, pddb_data_type)(PDBR)
                        (state(read_data(pm_phy, seg-
ment_reg_data_type)
                                (CS)
                                (s'1)))))) '2,
                    addr', ac1',
segment_to_priv(data(read_data(pm_phy, segment_reg_data_type)(CS)
                    (s'1))))
                    (state(translate(0,
                    data(read_data(pm_phy, pddb_data_type)(PDBR)
                        (state(read_data(pm_phy, seg-
ment_reg_data_type)
                                (CS)
                                (s'1)))) 'base_addr,
                    addr', ac1',
segment_to_priv(data(read_data(pm_phy, seg-
ment_reg_data_type)
                                (CS)
                                (s'1))))
                    (state(read_data(pm_phy, pddb_data_type)(PDBR)
                        (state(read_data(pm_phy, seg-
ment_reg_data_type)
                                (CS)
                                (s'1)))))))))
{-4} OK?(translate(1,
                    data(translate(0,
                                    data(read_data(pm_phy, pddb_data_type)(PDBR)
                                        (state(read_data(pm_phy, seg-
ment_reg_data_type)
                                                (CS)
                                                (s'1)))) 'base_addr,
                                        addr', ac1',
segment_to_priv(data(read_data(pm_phy, seg-
ment_reg_data_type)
                                                (CS)
                                                (s'1))))
                    (state(read_data(pm_phy, pddb_data_type)(PDBR)
                        (state(read_data(pm_phy, seg-
ment_reg_data_type)
                                (CS)
                                (s'1)))))) '2,
                    addr', ac2',
segment_to_priv(data(read_data(pm_phy, segment_reg_data_type)(CS)
                    (s'1))))
                    (state(translate(0,
                    data(read_data(pm_phy, pddb_data_type)(PDBR)
                        (state(read_data(pm_phy, seg-
ment_reg_data_type)
                                (CS)
                                (s'1))))))

```

C Proof scripts

Replacing using formula -5,

which is trivially true.

This completes the proof of `linear_resolve_same_result.1.1.1.1.1.1.2.1.1.1.1.`

C.117 Proofs for Linear_Memory_Properties (challenge-linear.pvs)

linear_resolve_same_result.1.1.1.1.1.1.2.1.1.1.2:

```

{-1} union(pm' 'ro_addr, pm' 'rw_addr)(addr')
{-2} is_linear_plain_memory?(pm')
{-3} OK?(translate(1,
                    data(translate(0,
                                   data(read_data(pm_phy, pdbr_data_type)(PDBR)
                                       (state(read_data(pm_phy, seg-
ment_reg_data_type)
                                       (CS)
                                       (s'_1)))) 'base_addr,
                                   addr', ac1',
                                   segment_to_priv(data(read_data(pm_phy, seg-
ment_reg_data_type)
                                       (CS)
                                       (s'_1))))
                                       (state(read_data(pm_phy, pdbr_data_type)(PDBR)
                                       (state(read_data(pm_phy, seg-
ment_reg_data_type)
                                       (CS)
                                       (s'_1)))))) '2,
                                   addr', ac1',
                                   segment_to_priv(data(read_data(pm_phy, segment_reg_data_type)(CS)
                                       (s'_1))))
                                       (state(translate(0,
                                   data(read_data(pm_phy, pdbr_data_type)(PDBR)
                                       (state(read_data(pm_phy, seg-
ment_reg_data_type)
                                       (CS)
                                       (s'_1)))) 'base_addr,
                                   addr', ac1',
                                   segment_to_priv(data(read_data(pm_phy, seg-
ment_reg_data_type)
                                       (CS)
                                       (s'_1))))
                                       (state(read_data(pm_phy, pdbr_data_type)(PDBR)
                                       (state(read_data(pm_phy, seg-
ment_reg_data_type)
                                       (CS)
                                       (s'_1)))))))))
{-4} OK?(translate(1,
                    data(translate(0,
                                   data(read_data(pm_phy, pdbr_data_type)(PDBR)
                                       (state(read_data(pm_phy, seg-
ment_reg_data_type)
                                       (CS)
                                       (s'_1)))) 'base_addr,
                                   addr', ac1',
                                   segment_to_priv(data(read_data(pm_phy, seg-
ment_reg_data_type)
                                       (CS)
                                       (s'_1))))
                                       (state(read_data(pm_phy, pdbr_data_type)(PDBR)
                                       (state(read_data(pm_phy, seg-
ment_reg_data_type)
                                       (CS)
                                       (s'_1))))))
                                   addr', ac2',
                                   segment_to_priv(data(read_data(pm_phy, segment_reg_data_type)(CS)
                                       (s'_1))))
                                       (state(translate(0,
                                   data(read_data(pm_phy, pdbr_data_type)(PDBR)
                                       (state(read_data(pm_phy, seg-
ment_reg_data_type)
                                       (CS)
                                       (s'_1))))

```

C Proof scripts

Hiding formulas: (-3 -4 3),

Using lemma `xlat_pe_in_pt_range`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `linear_resolve_same_result.1.1.1.1.1.1.2.1.1.1.2`.

C.117 Proofs for Linear_Memory_Properties (challenge-linear.pvs)

linear_resolve_same_result.1.1.1.1.1.1.2.1.1.2:

```

{-1}  pm' `states
      (state(translate(0,
                      data(read_data(pm_phy, pdb_r_data_type)(PDBR)
                          (state(read_data(pm_phy, seg-
ment_reg_data_type)(CS)
                              (s'_1)))))) `base_addr,
                      addr', ac2',
                      segment_to_priv(data(read_data(pm_phy, seg-
ment_reg_data_type)(CS)
                              (s'_1))))
      (state(read_data(pm_phy, pdb_r_data_type)(PDBR)
          (state(read_data(pm_phy, seg-
ment_reg_data_type)(CS)
                              (s'_2))))))))

{-2}  pm' `states
      (state(translate(0,
                      data(read_data(pm_phy, pdb_r_data_type)(PDBR)
                          (state(read_data(pm_phy, seg-
ment_reg_data_type)(CS)
                              (s'_1)))))) `base_addr,
                      addr', ac1',
                      segment_to_priv(data(read_data(pm_phy, seg-
ment_reg_data_type)(CS)
                              (s'_1))))
      (state(read_data(pm_phy, pdb_r_data_type)(PDBR)
          (state(read_data(pm_phy, seg-
ment_reg_data_type)(CS)
                              (s'_1))))))))

{-3}  OK?(translate(1,
                    data(translate(0,
                                    data(read_data(pm_phy, pdb_r_data_type)(PDBR)
                                        (state(read_data(pm_phy, seg-
ment_reg_data_type)
                                                    (CS)
                                                    (s'_1)))))) `base_addr,
                                    addr', ac1',
                                    segment_to_priv(data(read_data(pm_phy, seg-
ment_reg_data_type)
                                                            (CS)
                                                            (s'_1))))
                                        (state(read_data(pm_phy, pdb_r_data_type)(PDBR)
                                            (state(read_data(pm_phy, seg-
ment_reg_data_type)
                                                            (CS)
                                                            (s'_1)))))) `2,
                                    addr', ac2',
                                    segment_to_priv(data(read_data(pm_phy, segment_reg_data_type)(CS)
                                                            (s'_1))))
                    (state(translate(0,
                                    data(read_data(pm_phy, pdb_r_data_type)(PDBR)
                                        (state(read_data(pm_phy, seg-
ment_reg_data_type)
                                                    (CS)
                                                    (s'_1)))))) `base_addr,
                                    addr', ac2',
                                    segment_to_priv(data(read_data(pm_phy, seg-
ment_reg_data_type)
                                                            (CS)
                                                            (s'_1))))
                                        (state(read_data(pm_phy, pdb_r_data_type)(PDBR)
                                            (state(read_data(pm_phy, seg-
ment_reg_data_type)
                                                            (CS)
                                                            (s'_1))))))))

```

C Proof scripts

Hiding formulas: (-1 -2 3),

Using lemma `translate_result_no_leaf`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `linear_resolve_same_result.1.1.1.1.1.1.2.1.1.2`.

C Proof scripts

Hiding formulas: (-1 -2 -3 -4 -18 3),

Using lemma `translate_transformer_invariant`,

Using lemma `expr_transformer_invariant_next_ok`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Using lemma `pm_linear_blessed`,

Expanding the definition of `linear_blessed?`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Hiding formulas: (-1 -2 -3 -4 -6 -7),

Using lemma `subset_transitive`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 2 subgoals:

linear_resolve_same_result.1.1.1.1.1.1.2.1.2.1:

```

{-1} (address_in_pt_range?(state(read_data(pm_phy, segment_reg_data_type)(CS)(s'_1)), (restrict [Address, Mem
{-2} pm' 'states
      (state(read_data(pm_phy, pdbr_data_type)(PDBR)
              (state(read_data(pm_phy, segment_reg_data_type)(CS)(s'_2))))))
{-3} union(pm' 'ro_addr, pm' 'rw_addr)(addr')
{-4} is_linear_plain_memory?(pm')
{-5} pm' 'states
      (state(read_data(pm_phy, pdbr_data_type)(PDBR)
              (state(read_data(pm_phy, segment_reg_data_type)(CS)(s'_1))))))
{-6} data(read_data(pm_phy, pdbr_data_type)(PDBR)
          (state(read_data(pm_phy, segment_reg_data_type)(CS)(s'_1)))) 'base_addr
      =
      data(read_data(pm_phy, pdbr_data_type)(PDBR)
          (state(read_data(pm_phy, segment_reg_data_type)(CS)(s'_2)))) 'base_addr
{-7} pm' 'states(state(read_data(pm_phy, segment_reg_data_type)(CS)(s'_2)))
{-8} pm' 'states(state(read_data(pm_phy, segment_reg_data_type)(CS)(s'_1)))
{-9} data(read_data(pm_phy, segment_reg_data_type)(CS)(s'_1)) =
      data(read_data(pm_phy, segment_reg_data_type)(CS)(s'_2))
{-10} Mem?(data(translate(0,
                      data(read_data(pm_phy, pdbr_data_type)(PDBR)
                          (state(read_data(pm_phy, seg-
segment_reg_data_type)(CS)
                                      (s'_1)))) 'base_addr,
                      addr', ac1',
                      segment_to_priv(data(read_data(pm_phy, seg-
segment_reg_data_type)(CS)
                                      (s'_1))))
                      (state(read_data(pm_phy, pdbr_data_type)(PDBR)
                          (state(read_data(pm_phy, seg-
segment_reg_data_type)(CS)
                                      (s'_1)))))) '2 'type_of)
{-11} data(translate(0,
                    data(read_data(pm_phy, pdbr_data_type)(PDBR)
                        (state(read_data(pm_phy, seg-
segment_reg_data_type)(CS)
                                (s'_1)))) 'base_addr,
                    addr', ac1',
                    segment_to_priv(data(read_data(pm_phy, seg-
segment_reg_data_type)(CS)
                                (s'_1))))
                    (state(read_data(pm_phy, pdbr_data_type)(PDBR)
                        (state(read_data(pm_phy, seg-
segment_reg_data_type)(CS)
                                (s'_1)))))) '2 'offset
                    < max_linear_offset
{-12} OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s'_2))
{-13} OK?(read_data(pm_phy, pdbr_data_type)(PDBR)
        (state(read_data(pm_phy, segment_reg_data_type)(CS)(s'_2))))
{-14} OK?(translate(0,
                    data(read_data(pm_phy, pdbr_data_type)(PDBR)
                        (state(read_data(pm_phy, seg-
segment_reg_data_type)(CS)
                                (s'_1)))) 'base_addr,
                    addr', ac2',
                    segment_to_priv(data(read_data(pm_phy, segment_reg_data_type)(CS)
                                (s'_1))))
                    (state(read_data(pm_phy, pdbr_data_type)(PDBR)
                        (state(read_data(pm_phy, seg-
segment_reg_data_type)(CS)
                                (s'_2))))))
{-15} OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s'_1))
{-16} OK?(read_data(pm_phy, pdbr_data_type)(PDBR)
        (state(read_data(pm_phy, segment_reg_data_type)(CS)(s'_1))))
{-17} interpreted_data_type?(segment_reg_data_type)

```

C Proof scripts

Keeping (-1 1) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

Case splitting on `restrict[Address, Memory_Address_4G, boolean] (union(pm!1'ro_addr, pm!1'rw_addr)) = union(restrict[Address, Memory_Address_4G, boolean](pm!1'ro_addr), restrict[Address, Memory_Address_4G, boolean](pm!1'rw_addr))`,

we get 2 subgoals:

`linear_resolve_same_result.1.1.1.1.1.1.2.1.2.1.1:`

<pre>{-1} restrict[Address, Memory_Address_4G, boolean]((pm'ro_addr ∪ pm'rw_addr)) = (restrict[Address, Memory_Address_4G, boolean](pm'ro_addr) ∪ restrict [Address, Memory_A {-2} Mem?(x' type_of) {-3} 0 ≤ x' offset {-4} x' offset < expt(2, 32) {-5} address_in_pt_range?(state(read_data(pm_phy, segment_reg_data_type) ((#type_of := CS_, offset := 0#))(s'_1)), restrict[Address, Memory_Address_4G, boolean] ((pm'ro_addr ∪ pm'rw_addr))) (x')</pre>	<pre> </pre>
<pre>{1} pm_phy'rw_addr(x') {2} address_in_pt_range?(state(read_data(pm_phy, segment_reg_data_type) ((#type_of := CS_, offset := 0#))(s'_1)), (restrict[Address, Memory_Address_4G, boolean](pm'ro_addr) ∪ restri (x')</pre>	<pre> </pre>

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `linear_resolve_same_result.1.1.1.1.1.1.2.1.2.1.1`.

`linear_resolve_same_result.1.1.1.1.1.1.2.1.2.1.2:`

<pre>{-1} Mem?(x' type_of) {-2} 0 ≤ x' offset {-3} x' offset < expt(2, 32) {-4} address_in_pt_range?(state(read_data(pm_phy, segment_reg_data_type) ((#type_of := CS_, offset := 0#))(s'_1)), restrict[Address, Memory_Address_4G, boolean] ((pm'ro_addr ∪ pm'rw_addr))) (x')</pre>	<pre> </pre>
<pre>{1} restrict[Address, Memory_Address_4G, boolean]((pm'ro_addr ∪ pm'rw_addr)) = (restrict[Address, Memory_Address_4G, boolean](pm'ro_addr) ∪ restrict [Address, Memory_A {2} pm_phy'rw_addr(x') {3} address_in_pt_range?(state(read_data(pm_phy, segment_reg_data_type) ((#type_of := CS_, offset := 0#))(s'_1)), (restrict[Address, Memory_Address_4G, boolean](pm'ro_addr) ∪ restri (x')</pre>	<pre> </pre>

Keeping (1) and hiding *,

Applying decompose-equality,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 2 subgoals:

`linear_resolve_same_result.1.1.1.1.1.1.2.1.2.1.2.1:`

<pre>{-1} union(pm'ro_addr, pm'rw_addr)(x'')</pre>	<pre> </pre>
<pre>{1} union(restrict[Address, Memory_Address_4G, boolean](pm'ro_addr), restrict[Address, Memory_Address_4G, boolean](pm'rw_addr)) (x'')</pre>	<pre> </pre>

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `linear_resolve_same_result.1.1.1.1.1.1.2.1.2.1.2.1`.

`linear_resolve_same_result.1.1.1.1.1.1.2.1.2.1.2.2`:

$$\begin{array}{|l}
 \{-1\} \quad \text{union}(\text{restrict}[\text{Address}, \text{Memory_Address_4G}, \text{boolean}](\text{pm}'\text{'ro_addr}), \\
 \qquad \qquad \qquad \text{restrict}[\text{Address}, \text{Memory_Address_4G}, \text{boolean}](\text{pm}'\text{'rw_addr})) \\
 \qquad \qquad \qquad (x'') \\
 \hline
 \{1\} \quad \text{union}(\text{pm}'\text{'ro_addr}, \text{pm}'\text{'rw_addr})(x'')
 \end{array}$$

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `linear_resolve_same_result.1.1.1.1.1.1.2.1.2.1.2.2`.

linear_resolve_same_result.1.1.1.1.1.1.2.1.2.2:

```

{-1} (address_in_pt_range?(state(read_data(pm_phy, segment_reg_data_type)(CS)(s'_1)), (restrict [Adc
{-2} pm' 'states
      (state(read_data(pm_phy, pdbrr_data_type)(PDBR)
              (state(read_data(pm_phy, segment_reg_data_type)(CS)(s'_2))))))
{-3} union(pm' 'ro_addr, pm' 'rw_addr)(addr')
{-4} is_linear_plain_memory?(pm')
{-5} pm' 'states
      (state(read_data(pm_phy, pdbrr_data_type)(PDBR)
              (state(read_data(pm_phy, segment_reg_data_type)(CS)(s'_1))))))
{-6} data(read_data(pm_phy, pdbrr_data_type)(PDBR)
          (state(read_data(pm_phy, segment_reg_data_type)(CS)(s'_1)))) 'base_addr
      =
      data(read_data(pm_phy, pdbrr_data_type)(PDBR)
            (state(read_data(pm_phy, segment_reg_data_type)(CS)(s'_2)))) 'base_addr
{-7} pm' 'states(state(read_data(pm_phy, segment_reg_data_type)(CS)(s'_2)))
{-8} pm' 'states(state(read_data(pm_phy, segment_reg_data_type)(CS)(s'_1)))
{-9} data(read_data(pm_phy, segment_reg_data_type)(CS)(s'_1)) =
      data(read_data(pm_phy, segment_reg_data_type)(CS)(s'_2))
{-10} Mem?(data(translate(0,
                        data(read_data(pm_phy, pdbrr_data_type)(PDBR)
                              (state(read_data(pm_phy, seg-
segment_reg_data_type)(CS)
                                (s'_1)))) 'base_addr,
                        addr', ac1',
                        segment_to_priv(data(read_data(pm_phy, seg-
segment_reg_data_type)(CS)
                                (s'_1))))
                        (state(read_data(pm_phy, pdbrr_data_type)(PDBR)
                                (state(read_data(pm_phy, seg-
segment_reg_data_type)(CS)
                                (s'_1)))))) '2 'type_of)
{-11} data(translate(0,
                    data(read_data(pm_phy, pdbrr_data_type)(PDBR)
                          (state(read_data(pm_phy, seg-
segment_reg_data_type)(CS)
                                (s'_1)))) 'base_addr,
                    addr', ac1',
                    segment_to_priv(data(read_data(pm_phy, seg-
segment_reg_data_type)(CS)
                                (s'_1))))
                    (state(read_data(pm_phy, pdbrr_data_type)(PDBR)
                          (state(read_data(pm_phy, seg-
segment_reg_data_type)(CS)
                                (s'_1)))))) '2 'offset
      < max_linear_offset
{-12} OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s'_2))
{-13} OK?(read_data(pm_phy, pdbrr_data_type)(PDBR)
      (state(read_data(pm_phy, segment_reg_data_type)(CS)(s'_2))))
{-14} OK?(translate(0,
                    data(read_data(pm_phy, pdbrr_data_type)(PDBR)
                          (state(read_data(pm_phy, seg-
segment_reg_data_type)(CS)
                                (s'_1)))) 'base_addr,
                    addr', ac2',
                    segment_to_priv(data(read_data(pm_phy, segment_reg_data_type)(CS)
                                (s'_1))))
                    (state(read_data(pm_phy, pdbrr_data_type)(PDBR)
                          (state(read_data(pm_phy, seg-
segment_reg_data_type)(CS)
                                (s'_2))))))
{-15} OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s'_1))
{-16} OK?(read_data(pm_phy, pdbrr_data_type)(PDBR)
      (state(read_data(pm_phy, segment_reg_data_type)(CS)(s'_1))))
{-17} interpreted_data_type?(segment_reg_data_type)

```

C.117 Proofs for Linear_Memory_Properties (challenge-linear.pvs)

Using lemma `pe_in_pt_address_in_pt`,

we get 2 subgoals:

C.117 Proofs for Linear_Memory_Properties (challenge-linear.pvs)

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Using lemma `xlat_pe_in_pdir_range`,

Expanding the definition of `pe_in_pt_range?`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `linear_resolve_same_result.1.1.1.1.1.1.2.1.2.2.1`.

linear_resolve_same_result.1.1.1.1.1.2.1.2.2.2:

```

{-1} (address_in_pt_range?(state(read_data(pm_phy, segment_reg_data_type)(CS)(s'_1)), (restrict [Adc
{-2} pm' 'states
      (state(read_data(pm_phy, pdb_r_data_type)(PDBR)
              (state(read_data(pm_phy, segment_reg_data_type)(CS)(s'_2))))))
{-3} union(pm' 'ro_addr, pm' 'rw_addr)(addr')
{-4} is_linear_plain_memory?(pm')
{-5} pm' 'states
      (state(read_data(pm_phy, pdb_r_data_type)(PDBR)
              (state(read_data(pm_phy, segment_reg_data_type)(CS)(s'_1))))))
{-6} data(read_data(pm_phy, pdb_r_data_type)(PDBR)
          (state(read_data(pm_phy, segment_reg_data_type)(CS)(s'_1)))) 'base_addr
      =
      data(read_data(pm_phy, pdb_r_data_type)(PDBR)
          (state(read_data(pm_phy, segment_reg_data_type)(CS)(s'_2)))) 'base_addr
{-7} pm' 'states(state(read_data(pm_phy, segment_reg_data_type)(CS)(s'_2)))
{-8} pm' 'states(state(read_data(pm_phy, segment_reg_data_type)(CS)(s'_1)))
{-9} data(read_data(pm_phy, segment_reg_data_type)(CS)(s'_1)) =
      data(read_data(pm_phy, segment_reg_data_type)(CS)(s'_2))
{-10} Mem?(data(translate(0,
                        data(read_data(pm_phy, pdb_r_data_type)(PDBR)
                                (state(read_data(pm_phy, seg-
segment_reg_data_type)(CS)
                                (s'_1)))) 'base_addr,
                        addr', ac1',
                        segment_to_priv(data(read_data(pm_phy, seg-
segment_reg_data_type)(CS)
                                (s'_1))))
                                (state(read_data(pm_phy, pdb_r_data_type)(PDBR)
                                        (state(read_data(pm_phy, seg-
segment_reg_data_type)(CS)
                                (s'_1)))))) '2 'type_of)
{-11} data(translate(0,
                    data(read_data(pm_phy, pdb_r_data_type)(PDBR)
                            (state(read_data(pm_phy, seg-
segment_reg_data_type)(CS)
                            (s'_1)))) 'base_addr,
                    addr', ac1',
                    segment_to_priv(data(read_data(pm_phy, seg-
segment_reg_data_type)(CS)
                            (s'_1))))
                            (state(read_data(pm_phy, pdb_r_data_type)(PDBR)
                                    (state(read_data(pm_phy, seg-
segment_reg_data_type)(CS)
                            (s'_1)))))) '2 'offset
                    < max_linear_offset
{-12} OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s'_2))
{-13} OK?(read_data(pm_phy, pdb_r_data_type)(PDBR)
      (state(read_data(pm_phy, segment_reg_data_type)(CS)(s'_2))))
{-14} OK?(translate(0,
                    data(read_data(pm_phy, pdb_r_data_type)(PDBR)
                            (state(read_data(pm_phy, seg-
segment_reg_data_type)(CS)
                            (s'_1)))) 'base_addr,
                    addr', ac2',
                    segment_to_priv(data(read_data(pm_phy, segment_reg_data_type)(CS)
                            (s'_1))))
                            (state(read_data(pm_phy, pdb_r_data_type)(PDBR)
                                    (state(read_data(pm_phy, seg-
segment_reg_data_type)(CS)
                            (s'_2))))))
{-15} OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s'_1))
{-16} OK?(read_data(pm_phy, pdb_r_data_type)(PDBR)
      (state(read_data(pm_phy, segment_reg_data_type)(CS)(s'_1))))
{-17} interpreted_data_type?(segment_reg_data_type)

```

C.117 Proofs for Linear_Memory_Properties (challenge-linear.pvs)

Using lemma `xlat_idx_memory_address`,

Using lemma `xlat_idx_memory_address2`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `linear_resolve_same_result.1.1.1.1.1.1.2.1.2.2.2`.

linear_resolve_same_result.1.1.1.1.1.1.2.1.3:

```

{-1} OK?(translate(1,
                    data(translate(0,
                                   data(read_data(pm_phy, pabr_data_type)(PDBR)
                                       (state(read_data(pm_phy, seg-
ment_reg_data_type)
                                           (CS)
                                           (s'1)))))'base_addr,
                                   addr', ac1',
                                   segment_to_priv(data(read_data(pm_phy, seg-
ment_reg_data_type)
                                           (CS)
                                           (s'1))))
                                       (state(read_data(pm_phy, pabr_data_type)(PDBR)
                                               (state(read_data(pm_phy, seg-
ment_reg_data_type)
                                           (CS)
                                           (s'1))))))'2,
                                   addr', ac2',
                                   segment_to_priv(data(read_data(pm_phy, segment_reg_data_type)(CS)
                                                           (s'1))))
                                       (state(translate(0,
                                                       data(read_data(pm_phy, pabr_data_type)(PDBR)
                                                           (state(read_data(pm_phy, seg-
ment_reg_data_type)
                                           (CS)
                                           (s'1)))))'base_addr,
                                                       addr', ac2',
                                                       segment_to_priv(data(read_data(pm_phy, seg-
ment_reg_data_type)
                                           (CS)
                                           (s'1))))
                                       (state(read_data(pm_phy, pabr_data_type)(PDBR)
                                               (state(read_data(pm_phy,
segment_reg_data_type)
                                           (CS)
                                           (s'2))))))))))
{-2} OK?(translate(1,
                    data(translate(0,
                                   data(read_data(pm_phy, pabr_data_type)(PDBR)
                                       (state(read_data(pm_phy, seg-
ment_reg_data_type)
                                           (CS)
                                           (s'1)))))'base_addr,
                                   addr', ac1',
                                   segment_to_priv(data(read_data(pm_phy, seg-
ment_reg_data_type)
                                           (CS)
                                           (s'1))))
                                       (state(read_data(pm_phy, pabr_data_type)(PDBR)
                                               (state(read_data(pm_phy, seg-
ment_reg_data_type)
                                           (CS)
                                           (s'1))))))'2,
                                   addr', ac1',
                                   segment_to_priv(data(read_data(pm_phy, segment_reg_data_type)(CS)
                                                           (s'1))))
                                       (state(translate(0,
                                                       data(read_data(pm_phy, pabr_data_type)(PDBR)
                                                           (state(read_data(pm_phy, seg-
ment_reg_data_type)
                                           (CS)
                                           (s'1)))))'base_addr,
                                                       (CS)
                                                       (s'1))))'base_addr

```

C.117 Proofs for Linear_Memory_Properties (challenge-linear.pvs)

Hiding formulas: (-1 -2 -3 -14 3),

Using lemma `translate_transformer_invariant`,

Using lemma `expr_transformer_invariant_next_ok`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Using lemma `pm_linear_blessed`,

Expanding the definition of `linear_blessed?`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Hiding formulas: (-1 -2 -3 -4 -6 -7),

Using lemma `subset_transitive`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 2 subgoals:

linear_resolve_same_result.1.1.1.1.1.1.2.1.3.1:

```

{-1} (address_in_pt_range?(state(read_data(pm_phy, segment_reg_data_type)(CS)(s'_1)), (restrict [Adc
{-2} pm' 'states
      (state(read_data(pm_phy, pdb_r_data_type)(PDBR)
              (state(read_data(pm_phy, segment_reg_data_type)(CS)(s'_1))))))
{-3} union(pm' 'ro_addr, pm' 'rw_addr)(addr')
{-4} is_linear_plain_memory?(pm')
{-5} pm' 'states
      (state(read_data(pm_phy, pdb_r_data_type)(PDBR)
              (state(read_data(pm_phy, segment_reg_data_type)(CS)(s'_2))))))
{-6} data(read_data(pm_phy, pdb_r_data_type)(PDBR)
          (state(read_data(pm_phy, segment_reg_data_type)(CS)(s'_1)))) 'base_addr
      =
      data(read_data(pm_phy, pdb_r_data_type)(PDBR)
          (state(read_data(pm_phy, segment_reg_data_type)(CS)(s'_2)))) 'base_addr
{-7} pm' 'states(state(read_data(pm_phy, segment_reg_data_type)(CS)(s'_2)))
{-8} pm' 'states(state(read_data(pm_phy, segment_reg_data_type)(CS)(s'_1)))
{-9} data(read_data(pm_phy, segment_reg_data_type)(CS)(s'_1)) =
      data(read_data(pm_phy, segment_reg_data_type)(CS)(s'_2))
{-10} Mem?(data(translate(0,
                        data(read_data(pm_phy, pdb_r_data_type)(PDBR)
                              (state(read_data(pm_phy, seg-
ment_reg_data_type)(CS)
                                (s'_1)))) 'base_addr,
                        addr', ac1',
                        segment_to_priv(data(read_data(pm_phy, seg-
ment_reg_data_type)(CS)
                                (s'_1))))
                        (state(read_data(pm_phy, pdb_r_data_type)(PDBR)
                                (state(read_data(pm_phy, seg-
ment_reg_data_type)(CS)
                                (s'_1)))))) '2 'type_of)
{-11} data(translate(0,
                    data(read_data(pm_phy, pdb_r_data_type)(PDBR)
                          (state(read_data(pm_phy, seg-
ment_reg_data_type)(CS)
                                (s'_1)))) 'base_addr,
                    addr', ac1',
                    segment_to_priv(data(read_data(pm_phy, seg-
ment_reg_data_type)(CS)
                                (s'_1))))
                    (state(read_data(pm_phy, pdb_r_data_type)(PDBR)
                          (state(read_data(pm_phy, seg-
ment_reg_data_type)(CS)
                                (s'_1)))))) '2 'offset
                    < max_linear_offset
{-12} OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s'_2))
{-13} OK?(read_data(pm_phy, pdb_r_data_type)(PDBR)
        (state(read_data(pm_phy, segment_reg_data_type)(CS)(s'_2))))
{-14} OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s'_1))
{-15} OK?(read_data(pm_phy, pdb_r_data_type)(PDBR)
        (state(read_data(pm_phy, segment_reg_data_type)(CS)(s'_1))))
{-16} OK?(translate(0,
                    data(read_data(pm_phy, pdb_r_data_type)(PDBR)
                          (state(read_data(pm_phy, seg-
ment_reg_data_type)(CS)
                                (s'_1)))) 'base_addr,
                    addr', ac1',
                    segment_to_priv(data(read_data(pm_phy, segment_reg_data_type)(CS)
                                (s'_1))))
                    (state(read_data(pm_phy, pdb_r_data_type)(PDBR)
                          (state(read_data(pm_phy, seg-
ment_reg_data_type)(CS)
                                (s'_1))))))
{-17} interpreted_data_type?(segment_reg_data_type)

```

Keeping (-1 1) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

Case splitting on $\text{restrict}[\text{Address}, \text{Memory_Address_4G}, \text{boolean}] (\text{union}(\text{pm!1'ro_addr}, \text{pm!1'rw_addr})) = \text{union}(\text{restrict}[\text{Address}, \text{Memory_Address_4G}, \text{boolean}](\text{pm!1'ro_addr}), \text{restrict}[\text{Address}, \text{Memory_Address_4G}, \text{boolean}](\text{pm!1'rw_addr}))$,

we get 2 subgoals:

`linear_resolve_same_result.1.1.1.1.1.1.2.1.3.1.1:`

<p>{-1} $\text{restrict}[\text{Address}, \text{Memory_Address_4G}, \text{boolean}]((\text{pm}'\text{ro_addr} \cup \text{pm}'\text{rw_addr})) =$ $(\text{restrict}[\text{Address}, \text{Memory_Address_4G}, \text{boolean}] (\text{pm}'\text{ro_addr}) \cup \text{restrict} [\text{Address}, \text{Memory_Address_4G},$ {-2} $\text{Mem?}(x'\text{type_of})$ {-3} $0 \leq x'\text{offset}$ {-4} $x'\text{offset} < \text{expt}(2, 32)$ {-5} $\text{address_in_pt_range?}(\text{state}(\text{read_data}(\text{pm_phy}, \text{segment_reg_data_type})$ $((\# \text{type_of} := \text{CS_}, \text{offset} := 0\#))(s'_1)),$ $\text{restrict}[\text{Address}, \text{Memory_Address_4G}, \text{boolean}]$ $((\text{pm}'\text{ro_addr} \cup \text{pm}'\text{rw_addr})))$ (x')</p>	$=$ $(\text{restrict}[\text{Address}, \text{Memory_Address_4G}, \text{boolean}] (\text{pm}'\text{ro_addr}) \cup \text{restrict} [\text{Address}, \text{Memory_Address_4G},$ $\text{restrict}[\text{Address}, \text{Memory_Address_4G}, \text{boolean}] (\text{pm}'\text{rw_addr}))$ (x')
<p>{1} $\text{pm_phy}'\text{rw_addr}(x')$ {2} $\text{address_in_pt_range?}(\text{state}(\text{read_data}(\text{pm_phy}, \text{segment_reg_data_type})$ $((\# \text{type_of} := \text{CS_}, \text{offset} := 0\#))(s'_1)),$ $(\text{restrict}[\text{Address}, \text{Memory_Address_4G}, \text{boolean}] (\text{pm}'\text{ro_addr}) \cup \text{restrict} [\text{Address}$ (x')</p>	$=$ $(\text{restrict}[\text{Address}, \text{Memory_Address_4G}, \text{boolean}] (\text{pm}'\text{ro_addr}) \cup \text{restrict} [\text{Address}$ $\text{restrict}[\text{Address}, \text{Memory_Address_4G}, \text{boolean}] (\text{pm}'\text{rw_addr}))$ (x')

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `linear_resolve_same_result.1.1.1.1.1.1.2.1.3.1.1.`

`linear_resolve_same_result.1.1.1.1.1.1.2.1.3.1.2:`

<p>{-1} $\text{Mem?}(x'\text{type_of})$ {-2} $0 \leq x'\text{offset}$ {-3} $x'\text{offset} < \text{expt}(2, 32)$ {-4} $\text{address_in_pt_range?}(\text{state}(\text{read_data}(\text{pm_phy}, \text{segment_reg_data_type})$ $((\# \text{type_of} := \text{CS_}, \text{offset} := 0\#))(s'_1)),$ $\text{restrict}[\text{Address}, \text{Memory_Address_4G}, \text{boolean}]$ $((\text{pm}'\text{ro_addr} \cup \text{pm}'\text{rw_addr})))$ (x')</p>	$=$ $(\text{restrict}[\text{Address}, \text{Memory_Address_4G}, \text{boolean}] (\text{pm}'\text{ro_addr}) \cup \text{restrict} [\text{Address}, \text{Memory_Address_4G},$ $\text{restrict}[\text{Address}, \text{Memory_Address_4G}, \text{boolean}] (\text{pm}'\text{rw_addr}))$ (x')
<p>{1} $\text{restrict}[\text{Address}, \text{Memory_Address_4G}, \text{boolean}]((\text{pm}'\text{ro_addr} \cup \text{pm}'\text{rw_addr})) =$ $(\text{restrict}[\text{Address}, \text{Memory_Address_4G}, \text{boolean}] (\text{pm}'\text{ro_addr}) \cup \text{restrict} [\text{Address}, \text{Memory_Address_4G},$ {2} $\text{pm_phy}'\text{rw_addr}(x')$ {3} $\text{address_in_pt_range?}(\text{state}(\text{read_data}(\text{pm_phy}, \text{segment_reg_data_type})$ $((\# \text{type_of} := \text{CS_}, \text{offset} := 0\#))(s'_1)),$ $(\text{restrict}[\text{Address}, \text{Memory_Address_4G}, \text{boolean}] (\text{pm}'\text{ro_addr}) \cup \text{restrict} [\text{Address}$ (x')</p>	$=$ $(\text{restrict}[\text{Address}, \text{Memory_Address_4G}, \text{boolean}] (\text{pm}'\text{ro_addr}) \cup \text{restrict} [\text{Address}$ $\text{restrict}[\text{Address}, \text{Memory_Address_4G}, \text{boolean}] (\text{pm}'\text{rw_addr}))$ (x')

Keeping (1) and hiding *,

Applying decompose-equality,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `linear_resolve_same_result.1.1.1.1.1.1.2.1.3.1.2.`

linear_resolve_same_result.1.1.1.1.1.1.2.1.3.2:

```

{-1} (address_in_pt_range?(state(read_data(pm_phy, segment_reg_data_type)(CS)(s'_1)), (restrict [Adc
{-2} pm' 'states
      (state(read_data(pm_phy, pdb_r_data_type)(PDBR)
              (state(read_data(pm_phy, segment_reg_data_type)(CS)(s'_1))))))
{-3} union(pm' 'ro_addr, pm' 'rw_addr)(addr')
{-4} is_linear_plain_memory?(pm')
{-5} pm' 'states
      (state(read_data(pm_phy, pdb_r_data_type)(PDBR)
              (state(read_data(pm_phy, segment_reg_data_type)(CS)(s'_2))))))
{-6} data(read_data(pm_phy, pdb_r_data_type)(PDBR)
          (state(read_data(pm_phy, segment_reg_data_type)(CS)(s'_1)))) 'base_addr
      =
      data(read_data(pm_phy, pdb_r_data_type)(PDBR)
          (state(read_data(pm_phy, segment_reg_data_type)(CS)(s'_2)))) 'base_addr
{-7} pm' 'states(state(read_data(pm_phy, segment_reg_data_type)(CS)(s'_2)))
{-8} pm' 'states(state(read_data(pm_phy, segment_reg_data_type)(CS)(s'_1)))
{-9} data(read_data(pm_phy, segment_reg_data_type)(CS)(s'_1)) =
      data(read_data(pm_phy, segment_reg_data_type)(CS)(s'_2))
{-10} Mem?(data(translate(0,
                        data(read_data(pm_phy, pdb_r_data_type)(PDBR)
                              (state(read_data(pm_phy, seg-
ment_reg_data_type)(CS)
                                (s'_1)))) 'base_addr,
                        addr', ac1',
                        segment_to_priv(data(read_data(pm_phy, seg-
ment_reg_data_type)(CS)
                                (s'_1))))
                        (state(read_data(pm_phy, pdb_r_data_type)(PDBR)
                                (state(read_data(pm_phy, seg-
ment_reg_data_type)(CS)
                                (s'_1)))))) '2 'type_of)
{-11} data(translate(0,
                    data(read_data(pm_phy, pdb_r_data_type)(PDBR)
                            (state(read_data(pm_phy, seg-
ment_reg_data_type)(CS)
                                (s'_1)))) 'base_addr,
                    addr', ac1',
                    segment_to_priv(data(read_data(pm_phy, seg-
ment_reg_data_type)(CS)
                                (s'_1))))
                    (state(read_data(pm_phy, pdb_r_data_type)(PDBR)
                            (state(read_data(pm_phy, seg-
ment_reg_data_type)(CS)
                                (s'_1)))))) '2 'offset
                    < max_linear_offset
{-12} OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s'_2))
{-13} OK?(read_data(pm_phy, pdb_r_data_type)(PDBR)
        (state(read_data(pm_phy, segment_reg_data_type)(CS)(s'_2))))
{-14} OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s'_1))
{-15} OK?(read_data(pm_phy, pdb_r_data_type)(PDBR)
        (state(read_data(pm_phy, segment_reg_data_type)(CS)(s'_1))))
{-16} OK?(translate(0,
                    data(read_data(pm_phy, pdb_r_data_type)(PDBR)
                            (state(read_data(pm_phy, seg-
ment_reg_data_type)(CS)
                                (s'_1)))) 'base_addr,
                    addr', ac1',
                    segment_to_priv(data(read_data(pm_phy, segment_reg_data_type)(CS)
                                (s'_1))))
                    (state(read_data(pm_phy, pdb_r_data_type)(PDBR)
                            (state(read_data(pm_phy, seg-
ment_reg_data_type)(CS)
                                (s'_1))))))
{-17} interpreted_data_type?(segment_reg_data_type)

```

Using lemma `pe_in_pt_address_in_pt`,

we get 2 subgoals:

linear_resolve_same_result.1.1.1.1.1.1.2.1.3.2.1:

```

{-1} pe_in_pt_range?(state(read_data(pm_phy, segment_reg_data_type)(CS)(s'_1)),
                        restrict[Address, Memory_Address_4G, boolean]
                        ((pm'ro_addr ∪ pm'rw_addr)),
                        0)
      (xlat_idx(0,
                data(read_data(pm_phy, pdb_r_data_type)(PDBR)
                    (state(read_data(pm_phy, seg-
ment_reg_data_type)
                                (CS)
                                (s'_1))))'base_addr,
                                addr'))
      ⊃
      (address_block(xlat_idx(0, data(read_data(pm_phy, pdb_r_data_type)(PDBR)(state(read_data(p
{-2} (address_in_pt_range?(state(read_data(pm_phy, segment_reg_data_type)(CS)(s'_1)), (restrict [Ad
{-3} pm' states
      (state(read_data(pm_phy, pdb_r_data_type)(PDBR)
                (state(read_data(pm_phy, segment_reg_data_type)(CS)(s'_1))))
{-4} union(pm'ro_addr, pm'rw_addr)(addr')
{-5} is_linear_plain_memory?(pm')
{-6} pm' states
      (state(read_data(pm_phy, pdb_r_data_type)(PDBR)
                (state(read_data(pm_phy, segment_reg_data_type)(CS)(s'_2))))
{-7} data(read_data(pm_phy, pdb_r_data_type)(PDBR)
      (state(read_data(pm_phy, segment_reg_data_type)(CS)(s'_1))))'base_addr
      =
      data(read_data(pm_phy, pdb_r_data_type)(PDBR)
                (state(read_data(pm_phy, segment_reg_data_type)(CS)(s'_2))))'base_addr
{-8} pm' states(state(read_data(pm_phy, segment_reg_data_type)(CS)(s'_2)))
{-9} pm' states(state(read_data(pm_phy, segment_reg_data_type)(CS)(s'_1)))
{-10} data(read_data(pm_phy, segment_reg_data_type)(CS)(s'_1)) =
      data(read_data(pm_phy, segment_reg_data_type)(CS)(s'_2))
{-11} Mem?(data(translate(0,
                        data(read_data(pm_phy, pdb_r_data_type)(PDBR)
                        (state(read_data(pm_phy, seg-
ment_reg_data_type)(CS)
                                (s'_1))))'base_addr,
                                addr', ac1',
                                segment_to_priv(data(read_data(pm_phy, seg-
ment_reg_data_type)(CS)
                                (s'_1))))
                        (state(read_data(pm_phy, pdb_r_data_type)(PDBR)
                                (state(read_data(pm_phy, seg-
ment_reg_data_type)(CS)
                                (s'_1))))))'2'type_of)
{-12} data(translate(0,
                        data(read_data(pm_phy, pdb_r_data_type)(PDBR)
                        (state(read_data(pm_phy, seg-
ment_reg_data_type)(CS)
                                (s'_1))))'base_addr,
                                addr', ac1',
                                segment_to_priv(data(read_data(pm_phy, seg-
ment_reg_data_type)(CS)
                                (s'_1))))
                        (state(read_data(pm_phy, pdb_r_data_type)(PDBR)
                                (state(read_data(pm_phy, seg-
ment_reg_data_type)(CS)
                                (s'_1))))))'2'offset
1960 < max_linear_offset
{-13} OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s'_2))
{-14} OK?(read_data(pm_phy, pdb_r_data_type)(PDBR)
      (state(read_data(pm_phy, segment_reg_data_type)(CS)(s'_2))))
{-15} OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s'_1))
{-16} OK?(read_data(pm_phy, pdb_r_data_type)(PDBR)
      (state(read_data(pm_phy, segment_reg_data_type)(CS)(s'_1))))

```

C.117 Proofs for Linear_Memory_Properties (challenge-linear.pvs)

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Using lemma `xlat_pe_in_pdir_range`,

Expanding the definition of `pe_in_pt_range?`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `linear_resolve_same_result.1.1.1.1.1.1.2.1.3.2.1`.

linear_resolve_same_result.1.1.1.1.1.1.2.1.3.2.2:

```

{-1} (address_in_pt_range?(state(read_data(pm_phy, segment_reg_data_type)(CS)(s'_1)), (restrict [Ad
{-2} pm' 'states
      (state(read_data(pm_phy, pdb_r_data_type)(PDBR)
              (state(read_data(pm_phy, segment_reg_data_type)(CS)(s'_1))))))
{-3} union(pm' 'ro_addr, pm' 'rw_addr)(addr')
{-4} is_linear_plain_memory?(pm')
{-5} pm' 'states
      (state(read_data(pm_phy, pdb_r_data_type)(PDBR)
              (state(read_data(pm_phy, segment_reg_data_type)(CS)(s'_2))))))
{-6} data(read_data(pm_phy, pdb_r_data_type)(PDBR)
          (state(read_data(pm_phy, segment_reg_data_type)(CS)(s'_1)))) 'base_addr
      =
      data(read_data(pm_phy, pdb_r_data_type)(PDBR)
          (state(read_data(pm_phy, segment_reg_data_type)(CS)(s'_2)))) 'base_addr
{-7} pm' 'states(state(read_data(pm_phy, segment_reg_data_type)(CS)(s'_2)))
{-8} pm' 'states(state(read_data(pm_phy, segment_reg_data_type)(CS)(s'_1)))
{-9} data(read_data(pm_phy, segment_reg_data_type)(CS)(s'_1)) =
      data(read_data(pm_phy, segment_reg_data_type)(CS)(s'_2))
{-10} Mem?(data(translate(0,
                        data(read_data(pm_phy, pdb_r_data_type)(PDBR)
                              (state(read_data(pm_phy, seg-
ment_reg_data_type)(CS)
                                (s'_1)))) 'base_addr,
                        addr', ac1',
                        segment_to_priv(data(read_data(pm_phy, seg-
ment_reg_data_type)(CS)
                                (s'_1))))
                        (state(read_data(pm_phy, pdb_r_data_type)(PDBR)
                                (state(read_data(pm_phy, seg-
ment_reg_data_type)(CS)
                                (s'_1)))))) '2 'type_of)
{-11} data(translate(0,
                    data(read_data(pm_phy, pdb_r_data_type)(PDBR)
                            (state(read_data(pm_phy, seg-
ment_reg_data_type)(CS)
                                (s'_1)))) 'base_addr,
                    addr', ac1',
                    segment_to_priv(data(read_data(pm_phy, seg-
ment_reg_data_type)(CS)
                                (s'_1))))
                    (state(read_data(pm_phy, pdb_r_data_type)(PDBR)
                            (state(read_data(pm_phy, seg-
ment_reg_data_type)(CS)
                                (s'_1)))))) '2 'offset
                    < max_linear_offset
{-12} OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s'_2))
{-13} OK?(read_data(pm_phy, pdb_r_data_type)(PDBR)
        (state(read_data(pm_phy, segment_reg_data_type)(CS)(s'_2))))
{-14} OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s'_1))
{-15} OK?(read_data(pm_phy, pdb_r_data_type)(PDBR)
        (state(read_data(pm_phy, segment_reg_data_type)(CS)(s'_1))))
{-16} OK?(translate(0,
                    data(read_data(pm_phy, pdb_r_data_type)(PDBR)
                            (state(read_data(pm_phy, seg-
ment_reg_data_type)(CS)
                                (s'_1)))) 'base_addr,
                    addr', ac1',
                    segment_to_priv(data(read_data(pm_phy, segment_reg_data_type)(CS)
                                (s'_1))))
                    (state(read_data(pm_phy, pdb_r_data_type)(PDBR)
                            (state(read_data(pm_phy, seg-
ment_reg_data_type)(CS)
                                (s'_1))))))
{-17} interpreted_data_type?(segment_reg_data_type)

```

C.117 Proofs for Linear_Memory_Properties (challenge-linear.pvs)

Using lemma `xlat_idx_memory_address`,

Using lemma `xlat_idx_memory_address2`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `linear_resolve_same_result.1.1.1.1.1.1.2.1.3.2.2`.

linear_resolve_same_result.1.1.1.1.1.1.2.2:

```

{-1} OK?(translate(1,
                    data(translate(0,
                                   data(read_data(pm_phy, pdbr_data_type)(PDBR)
                                         (state(read_data(pm_phy, seg-
ment_reg_data_type)
                                         (CS)
                                         (s'_1)))) 'base_addr,
                                   addr', ac1',
                                   segment_to_priv(data(read_data(pm_phy, seg-
ment_reg_data_type)
                                         (CS)
                                         (s'_1))))
                                         (state(read_data(pm_phy, pdbr_data_type)(PDBR)
                                               (state(read_data(pm_phy, seg-
ment_reg_data_type)
                                         (CS)
                                         (s'_1)))))) '2,
                                   addr', ac1',
                                   segment_to_priv(data(read_data(pm_phy, segment_reg_data_type)(CS)
                                         (s'_1)))
                                         (state(translate(0,
                    data(read_data(pm_phy, pdbr_data_type)(PDBR)
                                   (state(read_data(pm_phy, seg-
ment_reg_data_type)
                                   (CS)
                                   (s'_1)))) 'base_addr,
                                   addr', ac1',
                                   segment_to_priv(data(read_data(pm_phy, seg-
ment_reg_data_type)
                                         (CS)
                                         (s'_1)))
                                         (state(read_data(pm_phy, pdbr_data_type)(PDBR)
                                               (state(read_data(pm_phy,
segment_reg_data_type)
                                         (CS)
                                         (s'_1)))))))))
{-2} data(translate(0,
                    data(read_data(pm_phy, pdbr_data_type)(PDBR)
                          (state(read_data(pm_phy, seg-
ment_reg_data_type)(CS)
                          (s'_2)))) 'base_addr,
                    addr', ac2',
                    segment_to_priv(data(read_data(pm_phy, seg-
ment_reg_data_type)(CS)
                          (s'_2))))
                    (state(read_data(pm_phy, pdbr_data_type)(PDBR)
                          (state(read_data(pm_phy, seg-
ment_reg_data_type)(CS)
                          (s'_2))))))
=
data(translate(0,
                    data(read_data(pm_phy, pdbr_data_type)(PDBR)
                          (state(read_data(pm_phy, seg-
ment_reg_data_type)(CS)
                          (s'_1)))) 'base_addr,
                    addr', ac1',
                    segment_to_priv(data(read_data(pm_phy, seg-
ment_reg_data_type)(CS)
                          (s'_1)))
                    (state(read_data(pm_phy, pdbr_data_type)(PDBR)
                          (state(read_data(pm_phy, seg-
ment_reg_data_type)(CS)

```

C.117 Proofs for Linear_Memory_Properties (challenge-linear.pvs)

Revealing hidden formulas,

Hiding formulas: (-2 -5 -6 3),

Expanding the definition of linear_resolve,

Expanding the definition of ##,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 2 subgoals:

linear_resolve_same_result.1.1.1.1.1.1.2.2.1:

```

{-1} data(translate(0,
                    data(read_data(pm_phy, pdb_r_data_type)(PDBR)
                        (state(read_data(pm_phy, seg-
ment_reg_data_type)(CS)
                                (s'_2))))))'base_addr,
                    addr', ac2',
                    segment_to_priv(data(read_data(pm_phy, seg-
ment_reg_data_type)(CS)
                                (s'_2))))
                    (state(read_data(pm_phy, pdb_r_data_type)(PDBR)
                        (state(read_data(pm_phy, seg-
ment_reg_data_type)(CS)
                                (s'_2))))))'1
{-2} OK?(OK(state(translate(0,
                    data(read_data(pm_phy, pdb_r_data_type)(PDBR)
                        (state(read_data(pm_phy, seg-
ment_reg_data_type)
                                (CS)
                                (s'_2))))))'base_addr,
                    addr', ac2',
                    segment_to_priv(data(read_data(pm_phy, seg-
ment_reg_data_type)(CS)
                                (s'_2))))
                    (state(read_data(pm_phy, pdb_r_data_type)(PDBR)
                        (state(read_data(pm_phy, seg-
ment_reg_data_type)
                                (CS)
                                (s'_2))))))
                    xlat_ofs(0,
                    data(translate(0,
                    data(read_data(pm_phy, pdb_r_data_type)(PDBR)
                        (state(read_data
                                (pm_phy, seg-
ment_reg_data_type)
                                (CS)
                                (s'_2))))))'base_addr,
                    addr', ac2',
                    segment_to_priv(data(read_data(pm_phy,
                                seg-
ment_reg_data_type)
                                (CS)
                                (s'_2))))
                    (state(read_data(pm_phy, pdb_r_data_type)(PDBR)
                        (state(read_data
                                (pm_phy, seg-
ment_reg_data_type)
                                (CS)
                                (s'_2))))))'2,
                    addr'))))
{-3} data(translate(0,
                    data(read_data(pm_phy, pdb_r_data_type)(PDBR)
                        (state(read_data(pm_phy, seg-
ment_reg_data_type)(CS)
                                (s'_2))))))'base_addr,
                    addr', ac2',
                    segment_to_priv(data(read_data(pm_phy, seg-
ment_reg_data_type)(CS)
                                (s'_2))))
                    (state(read_data(pm_phy, pdb_r_data_type)(PDBR)
                        (state(read_data(pm_phy, seg-
ment_reg_data_type)(CS)
                                (s'_2))))))'2,
                    =
                    (s'_2))))

```

Replacing using formula -3,

which is trivially true.

This completes the proof of `linear_resolve_same_result.1.1.1.1.1.1.2.2.1`.

linear_resolve_same_result.1.1.1.1.1.1.2.2.2:

```

{-1} OK?(translate(1,
                    data(translate(0,
                                   data(read_data(pm_phy, pabr_data_type)(PABR)
                                       (state(read_data(pm_phy, seg-
ment_reg_data_type)
                                           (CS)
                                           (s'_2)))) 'base_addr,
                                   addr', ac2',
                                   segment_to_priv(data(read_data(pm_phy, seg-
ment_reg_data_type)
                                           (CS)
                                           (s'_2))))
                                       (state(read_data(pm_phy, pabr_data_type)(PABR)
                                               (state(read_data(pm_phy, seg-
ment_reg_data_type)
                                           (CS)
                                           (s'_2)))))) '2,
                                   addr', ac2',
                                   segment_to_priv(data(read_data(pm_phy, segment_reg_data_type)(CS)
                                                           (s'_2))))
                                       (state(translate(0,
                                                       data(read_data(pm_phy, pabr_data_type)(PABR)
                                                           (state(read_data(pm_phy, seg-
ment_reg_data_type)
                                           (CS)
                                           (s'_2)))) 'base_addr,
                                                       addr', ac2',
                                                       segment_to_priv(data(read_data(pm_phy, seg-
ment_reg_data_type)
                                           (CS)
                                           (s'_2))))
                                       (state(read_data(pm_phy, pabr_data_type)(PABR)
                                               (state(read_data(pm_phy,
segment_reg_data_type)
                                           (CS)
                                           (s'_2)))))))))
{-2} OK?(OK(state(translate(1,
                    data(translate(0,
                                   data(read_data(pm_phy, pabr_data_type)(PABR)
                                       (state
                                           (read_data
                                               (pm_phy, seg-
ment_reg_data_type)
                                           (CS)
                                           (s'_2)))) 'base_addr,
                                   addr', ac2',
                                   segment_to_priv(data
                                       (read_data
                                           (pm_phy, seg-
ment_reg_data_type)
                                       (CS)
                                       (s'_2))))
                                       (state(read_data(pm_phy, pabr_data_type)
                                               (PABR)
                                               (state
                                                   (read_data
                                                       (pm_phy, seg-
ment_reg_data_type)
                                                   (CS)
                                                   (s'_2)))))) '2,
                                   addr', ac2',
                                   segment_to_priv(data(read_data(pm_phy, seg-

```

Replacing using formula -3,

which is trivially true.

This completes the proof of `linear_resolve_same_result.1.1.1.1.1.1.2.2.2`.

linear_resolve_same_result.1.1.1.1.1.1.2.3:

```

{-1} data(translate(0,
                    data(read_data(pm_phy, pdb_data_type)(PDBR)
                        (state(read_data(pm_phy, seg-
segment_reg_data_type)(CS)
                                (s'_2)))) 'base_addr,
                    addr', ac2',
                    segment_to_priv(data(read_data(pm_phy, seg-
segment_reg_data_type)(CS)
                                (s'_2))))
                    (state(read_data(pm_phy, pdb_data_type)(PDBR)
                        (state(read_data(pm_phy, seg-
segment_reg_data_type)(CS)
                                (s'_2))))))
=
data(translate(0,
                    data(read_data(pm_phy, pdb_data_type)(PDBR)
                        (state(read_data(pm_phy, seg-
segment_reg_data_type)(CS)
                                (s'_1)))) 'base_addr,
                    addr', ac1',
                    segment_to_priv(data(read_data(pm_phy, seg-
segment_reg_data_type)(CS)
                                (s'_1))))
                    (state(read_data(pm_phy, pdb_data_type)(PDBR)
                        (state(read_data(pm_phy, seg-
segment_reg_data_type)(CS)
                                (s'_1))))))
{-2} pm' 'states
      (state(read_data(pm_phy, pdb_data_type)(PDBR)
          (state(read_data(pm_phy, segment_reg_data_type)(CS)(s'_2))))))
{-3} pm' 'states
      (state(read_data(pm_phy, pdb_data_type)(PDBR)
          (state(read_data(pm_phy, segment_reg_data_type)(CS)(s'_1))))))
{-4} data(read_data(pm_phy, pdb_data_type)(PDBR)
      (state(read_data(pm_phy, segment_reg_data_type)(CS)(s'_1)))) 'base_addr
=
data(read_data(pm_phy, pdb_data_type)(PDBR)
      (state(read_data(pm_phy, segment_reg_data_type)(CS)(s'_2)))) 'base_addr
{-5} pm' 'states(state(read_data(pm_phy, segment_reg_data_type)(CS)(s'_2)))
{-6} pm' 'states(state(read_data(pm_phy, segment_reg_data_type)(CS)(s'_1)))
{-7} data(read_data(pm_phy, segment_reg_data_type)(CS)(s'_1)) =
      data(read_data(pm_phy, segment_reg_data_type)(CS)(s'_2))
{-8} data(translate(0,
                    data(read_data(pm_phy, pdb_data_type)(PDBR)
                        (state(read_data(pm_phy, seg-
segment_reg_data_type)(CS)
                                (s'_1)))) 'base_addr,
                    addr', ac1',
                    segment_to_priv(data(read_data(pm_phy, seg-
segment_reg_data_type)(CS)
                                (s'_1))))
                    (state(read_data(pm_phy, pdb_data_type)(PDBR)
                        (state(read_data(pm_phy, seg-
segment_reg_data_type)(CS)
                                (s'_1)))))) '2' type_of
= Mem_
^
data(translate(0,
                    data(read_data(pm_phy, pdb_data_type)(PDBR)
                        (state(read_data(pm_phy, seg-
segment_reg_data_type)(CS)
                                (s'_1)))) 'base_addr,
                    addr', ac1',
                    segment_to_priv(data(read_data(pm_phy, seg-
segment_reg_data_type)(CS)
                                (s'_1))))
                    (state(read_data(pm_phy, pdb_data_type)(PDBR)
                        (state(read_data(pm_phy, seg-
segment_reg_data_type)(CS)
                                (s'_1))))))

```

Hiding formulas: 3,

Revealing hidden formulas,

Expanding the definition of `linear_resolve`,

Expanding the definition of `##`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `linear_resolve_same_result.1.1.1.1.1.1.2.3`.

linear_resolve_same_result.1.1.1.1.1.2:

```

{-1} pm' 'states
      (state(read_data(pm_phy, pddb_data_type)(PDBR)
              (state(read_data(pm_phy, segment_reg_data_type)(CS)(s'_2))))))
{-2} pm' 'states
      (state(read_data(pm_phy, pddb_data_type)(PDBR)
              (state(read_data(pm_phy, segment_reg_data_type)(CS)(s'_1))))))
{-3} data(read_data(pm_phy, pddb_data_type)(PDBR)
          (state(read_data(pm_phy, segment_reg_data_type)(CS)(s'_1)))) 'base_addr
      =
      data(read_data(pm_phy, pddb_data_type)(PDBR)
          (state(read_data(pm_phy, segment_reg_data_type)(CS)(s'_2)))) 'base_addr
{-4} pm' 'states(state(read_data(pm_phy, segment_reg_data_type)(CS)(s'_2)))
{-5} pm' 'states(state(read_data(pm_phy, segment_reg_data_type)(CS)(s'_1)))
{-6} data(read_data(pm_phy, segment_reg_data_type)(CS)(s'_1)) =
      data(read_data(pm_phy, segment_reg_data_type)(CS)(s'_2))
{-7} aligned?(bits_per_level + pe_size)
      (offset
        (data(read_data(pm_phy, pddb_data_type)(PDBR)
              (state(read_data(pm_phy, segment_reg_data_type)(CS)
                        (s'_1)))) 'base_addr))
      ^
      OK?(translate(0,
                    data(read_data(pm_phy, pddb_data_type)(PDBR)
                          (state(read_data(pm_phy, seg-
ment_reg_data_type)(CS)
                                (s'_1)))) 'base_addr,
                    addr', ac1',
                    segment_to_priv(data(read_data(pm_phy, seg-
ment_reg_data_type)(CS)
                                          (s'_1))))
                    (state(read_data(pm_phy, pddb_data_type)(PDBR)
                          (state(read_data(pm_phy, seg-
ment_reg_data_type)(CS)
                                (s'_1))))))
      ⊃
      data(translate(0,
                    data(read_data(pm_phy, pddb_data_type)(PDBR)
                          (state(read_data(pm_phy, seg-
ment_reg_data_type)(CS)
                                (s'_1)))) 'base_addr,
                    addr', ac1',
                    segment_to_priv(data(read_data(pm_phy, seg-
ment_reg_data_type)(CS)
                                          (s'_1))))
                    (state(read_data(pm_phy, pddb_data_type)(PDBR)
                          (state(read_data(pm_phy, seg-
ment_reg_data_type)(CS)
                                (s'_1))))))
                    (s'_1)))))) '2
      < max_linear
{-8} aligned?(bits_per_level + pe_size)
      (offset
        (data(read_data(pm_phy, pddb_data_type)(PDBR)
              (state(read_data(pm_phy, segment_reg_data_type)(CS)
                        (s'_2)))) 'base_addr))
      ^
      OK?(translate(0,
                    data(read_data(pm_phy, pddb_data_type)(PDBR)
                          (state(read_data(pm_phy, seg-
ment_reg_data_type)(CS)
                                (s'_2)))) 'base_addr,
                    addr', ac2',
                    segment_to_priv(data(read_data(pm_phy, seg-
ment_reg_data_type)(CS)
                                          (s'_1))))
                    (s'_1))))

```

Hiding formulas: 2,

Using lemma `translate_same_result`,

we get 2 subgoals:

linear_resolve_same_result.1.1.1.1.1.2.1:

```

{-1} union(pm'ro_addr, pm'rw_addr)(addr') ∧
      is_linear_plain_memory?(pm') ∧
      pe_in_pt_range?(state(read_data(pm_phy, pdbr_data_type)(PDBR)
                              (state(read_data(pm_phy, seg-
segment_reg_data_type)(CS)
                              (s'1))))),
      restrict[Address, Memory_Address_4G, boolean]
      ((pm'ro_addr ∪ pm'rw_addr)),
      0)
      (xlat_idx(0,
                data(read_data(pm_phy, pdbr_data_type)(PDBR)
                      (state(read_data(pm_phy, seg-
segment_reg_data_type)
                              (CS)
                              (s'1))))'base_addr,
                addr'))
      ∧
      OK?(translate(0,
                    data(read_data(pm_phy, pdbr_data_type)(PDBR)
                          (state(read_data(pm_phy, seg-
segment_reg_data_type)(CS)
                          (s'1))))'base_addr,
                    addr', ac1',
                    segment_to_priv(data(read_data(pm_phy, seg-
segment_reg_data_type)(CS)
                    (s'1))))
                    (state(read_data(pm_phy, pdbr_data_type)(PDBR)
                          (state(read_data(pm_phy, seg-
segment_reg_data_type)(CS)
                          (s'1))))))
      ∧
      OK?(translate(0,
                    data(read_data(pm_phy, pdbr_data_type)(PDBR)
                          (state(read_data(pm_phy, seg-
segment_reg_data_type)(CS)
                          (s'1))))'base_addr,
                    addr', ac2',
                    segment_to_priv(data(read_data(pm_phy, seg-
segment_reg_data_type)(CS)
                    (s'1))))
                    (state(read_data(pm_phy, pdbr_data_type)(PDBR)
                          (state(read_data(pm_phy, seg-
segment_reg_data_type)(CS)
                          (s'2))))))
      ⊃
      data(translate(0,
                    data(read_data(pm_phy, pdbr_data_type)(PDBR)
                          (state(read_data(pm_phy, seg-
segment_reg_data_type)(CS)
                          (s'1))))'base_addr,
                    addr', ac1',
                    segment_to_priv(data(read_data(pm_phy, seg-
segment_reg_data_type)(CS)
                    (s'1))))
                    (state(read_data(pm_phy, pdbr_data_type)(PDBR)
                          (state(read_data(pm_phy, seg-
segment_reg_data_type)(CS)
                          (s'1))))))
      =
      data(translate(0,
                    data(read_data(pm_phy, pdbr_data_type)(PDBR)
                          (state(read_data(pm_phy, seg-
segment_reg_data_type)(CS)

```

C.117 Proofs for Linear_Memory_Properties (challenge-linear.pvs)

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Hiding formulas: (-3 -4 -5 -6 -7 -8 -9 -10 2),

Using lemma `xlat_pe_in_pdir_range`,

Expanding the definition of `pe_in_pt_range?`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `linear_resolve_same_result.1.1.1.1.1.2.1`.

linear_resolve_same_result.1.1.1.1.1.2.2:

```

{-1} pm' 'states
      (state(read_data(pm_phy, pddb_data_type)(PDBR)
              (state(read_data(pm_phy, segment_reg_data_type)(CS)(s'_2))))))
{-2} pm' 'states
      (state(read_data(pm_phy, pddb_data_type)(PDBR)
              (state(read_data(pm_phy, segment_reg_data_type)(CS)(s'_1))))))
{-3} data(read_data(pm_phy, pddb_data_type)(PDBR)
          (state(read_data(pm_phy, segment_reg_data_type)(CS)(s'_1)))) 'base_addr
      =
      data(read_data(pm_phy, pddb_data_type)(PDBR)
          (state(read_data(pm_phy, segment_reg_data_type)(CS)(s'_2)))) 'base_addr
{-4} pm' 'states(state(read_data(pm_phy, segment_reg_data_type)(CS)(s'_2)))
{-5} pm' 'states(state(read_data(pm_phy, segment_reg_data_type)(CS)(s'_1)))
{-6} data(read_data(pm_phy, segment_reg_data_type)(CS)(s'_1)) =
      data(read_data(pm_phy, segment_reg_data_type)(CS)(s'_2))
{-7} aligned?(bits_per_level + pe_size)
      (offset
        (data(read_data(pm_phy, pddb_data_type)(PDBR)
              (state(read_data(pm_phy, segment_reg_data_type)(CS)
                        (s'_1)))) 'base_addr))
      ^
      OK?(translate(0,
                    data(read_data(pm_phy, pddb_data_type)(PDBR)
                          (state(read_data(pm_phy, seg-
ment_reg_data_type)(CS)
                                (s'_1)))) 'base_addr,
                    addr', ac1',
                    segment_to_priv(data(read_data(pm_phy, seg-
ment_reg_data_type)(CS)
                                          (s'_1))))
                    (state(read_data(pm_phy, pddb_data_type)(PDBR)
                          (state(read_data(pm_phy, seg-
ment_reg_data_type)(CS)
                                (s'_1))))))
      ⊃
      data(translate(0,
                    data(read_data(pm_phy, pddb_data_type)(PDBR)
                          (state(read_data(pm_phy, seg-
ment_reg_data_type)(CS)
                                (s'_1)))) 'base_addr,
                    addr', ac1',
                    segment_to_priv(data(read_data(pm_phy, seg-
ment_reg_data_type)(CS)
                                          (s'_1))))
                    (state(read_data(pm_phy, pddb_data_type)(PDBR)
                          (state(read_data(pm_phy, seg-
ment_reg_data_type)(CS)
                                (s'_1))))))
                    (s'_1)))))) '2
      < max_linear
{-8} aligned?(bits_per_level + pe_size)
      (offset
        (data(read_data(pm_phy, pddb_data_type)(PDBR)
              (state(read_data(pm_phy, segment_reg_data_type)(CS)
                        (s'_2)))) 'base_addr))
      ^
      OK?(translate(0,
                    data(read_data(pm_phy, pddb_data_type)(PDBR)
                          (state(read_data(pm_phy, seg-
ment_reg_data_type)(CS)
                                (s'_2)))) 'base_addr,
                    addr', ac2',
                    segment_to_priv(data(read_data(pm_phy, seg-
ment_reg_data_type)(CS)
                                          (s'_1))))
                    (s'_1))))

```

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `linear_resolve_same_result.1.1.1.1.1.2.2`.

linear_resolve_same_result.1.1.1.1.1.3:

```

{-1} pm' 'states
      (state(read_data(pm_phy, pddb_data_type)(PDBR)
              (state(read_data(pm_phy, segment_reg_data_type)(CS)(s'_2))))))
{-2} pm' 'states
      (state(read_data(pm_phy, pddb_data_type)(PDBR)
              (state(read_data(pm_phy, segment_reg_data_type)(CS)(s'_1))))))
{-3} data(read_data(pm_phy, pddb_data_type)(PDBR)
          (state(read_data(pm_phy, segment_reg_data_type)(CS)(s'_1)))) 'base_addr
      =
      data(read_data(pm_phy, pddb_data_type)(PDBR)
          (state(read_data(pm_phy, segment_reg_data_type)(CS)(s'_2)))) 'base_addr
{-4} pm' 'states(state(read_data(pm_phy, segment_reg_data_type)(CS)(s'_2)))
{-5} pm' 'states(state(read_data(pm_phy, segment_reg_data_type)(CS)(s'_1)))
{-6} data(read_data(pm_phy, segment_reg_data_type)(CS)(s'_1)) =
      data(read_data(pm_phy, segment_reg_data_type)(CS)(s'_2))
{-7} aligned?(bits_per_level + pe_size)
      (offset
        (data(read_data(pm_phy, pddb_data_type)(PDBR)
              (state(read_data(pm_phy, segment_reg_data_type)(CS)
                        (s'_1)))) 'base_addr))
      ^
      OK?(translate(0,
                  data(read_data(pm_phy, pddb_data_type)(PDBR)
                      (state(read_data(pm_phy, seg-
ment_reg_data_type)(CS)
                          (s'_1)))) 'base_addr,
                  addr', ac1',
                  segment_to_priv(data(read_data(pm_phy, seg-
ment_reg_data_type)(CS)
                                      (s'_1))))
                      (state(read_data(pm_phy, pddb_data_type)(PDBR)
                          (state(read_data(pm_phy, seg-
ment_reg_data_type)(CS)
                              (s'_1))))))
      ⊃
      data(translate(0,
                  data(read_data(pm_phy, pddb_data_type)(PDBR)
                      (state(read_data(pm_phy, seg-
ment_reg_data_type)(CS)
                          (s'_1)))) 'base_addr,
                  addr', ac1',
                  segment_to_priv(data(read_data(pm_phy, seg-
ment_reg_data_type)(CS)
                                      (s'_1))))
                      (state(read_data(pm_phy, pddb_data_type)(PDBR)
                          (state(read_data(pm_phy, seg-
ment_reg_data_type)(CS)
                              (s'_1))))))
      (s'_1)))))) '2
      < max_linear
{-8} aligned?(bits_per_level + pe_size)
      (offset
        (data(read_data(pm_phy, pddb_data_type)(PDBR)
              (state(read_data(pm_phy, segment_reg_data_type)(CS)
                        (s'_2)))) 'base_addr))
      ^
      OK?(translate(0,
                  data(read_data(pm_phy, pddb_data_type)(PDBR)
                      (state(read_data(pm_phy, seg-
ment_reg_data_type)(CS)
                          (s'_2)))) 'base_addr,
                  addr', ac2',
                  segment_to_priv(data(read_data(pm_phy, seg-
ment_reg_data_type)(CS)
                                      (s'_1))))
                      (s'_1))))

```

which is trivially true.

This completes the proof of `linear_resolve_same_result.1.1.1.1.1.3`.

linear_resolve_same_result.1.1.1.1.1.4:

```

{-1} pm' 'states
      (state(read_data(pm_phy, pddb_data_type)(PDBR)
              (state(read_data(pm_phy, segment_reg_data_type)(CS)(s'_2))))))
{-2} pm' 'states
      (state(read_data(pm_phy, pddb_data_type)(PDBR)
              (state(read_data(pm_phy, segment_reg_data_type)(CS)(s'_1))))))
{-3} data(read_data(pm_phy, pddb_data_type)(PDBR)
          (state(read_data(pm_phy, segment_reg_data_type)(CS)(s'_1)))) 'base_addr
      =
      data(read_data(pm_phy, pddb_data_type)(PDBR)
          (state(read_data(pm_phy, segment_reg_data_type)(CS)(s'_2)))) 'base_addr
{-4} pm' 'states(state(read_data(pm_phy, segment_reg_data_type)(CS)(s'_2)))
{-5} pm' 'states(state(read_data(pm_phy, segment_reg_data_type)(CS)(s'_1)))
{-6} data(read_data(pm_phy, segment_reg_data_type)(CS)(s'_1)) =
      data(read_data(pm_phy, segment_reg_data_type)(CS)(s'_2))
{-7} aligned?(bits_per_level + pe_size)
      (offset
        (data(read_data(pm_phy, pddb_data_type)(PDBR)
              (state(read_data(pm_phy, segment_reg_data_type)(CS)
                        (s'_1)))) 'base_addr))
      ^
      OK?(translate(0,
                    data(read_data(pm_phy, pddb_data_type)(PDBR)
                          (state(read_data(pm_phy, seg-
ment_reg_data_type)(CS)
                                (s'_1)))) 'base_addr,
                    addr', ac1',
                    segment_to_priv(data(read_data(pm_phy, seg-
ment_reg_data_type)(CS)
                                          (s'_1))))
                    (state(read_data(pm_phy, pddb_data_type)(PDBR)
                          (state(read_data(pm_phy, seg-
ment_reg_data_type)(CS)
                                (s'_1))))))
      ⊃
      data(translate(0,
                    data(read_data(pm_phy, pddb_data_type)(PDBR)
                          (state(read_data(pm_phy, seg-
ment_reg_data_type)(CS)
                                (s'_1)))) 'base_addr,
                    addr', ac1',
                    segment_to_priv(data(read_data(pm_phy, seg-
ment_reg_data_type)(CS)
                                          (s'_1))))
                    (state(read_data(pm_phy, pddb_data_type)(PDBR)
                          (state(read_data(pm_phy, seg-
ment_reg_data_type)(CS)
                                (s'_1))))))
                    (s'_1)))))) '2
      < max_linear
{-8} aligned?(bits_per_level + pe_size)
      (offset
        (data(read_data(pm_phy, pddb_data_type)(PDBR)
              (state(read_data(pm_phy, segment_reg_data_type)(CS)
                        (s'_2)))) 'base_addr))
      ^
      OK?(translate(0,
                    data(read_data(pm_phy, pddb_data_type)(PDBR)
                          (state(read_data(pm_phy, seg-
ment_reg_data_type)(CS)
                                (s'_2)))) 'base_addr,
                    addr', ac2',
                    segment_to_priv(data(read_data(pm_phy, seg-
ment_reg_data_type)(CS)
                                          (s'_1))))
                    (s'_1))))

```

which is trivially true.

This completes the proof of `linear_resolve_same_result.1.1.1.1.1.4`.

linear_resolve_same_result.1.1.1.1.2:

```

{-1} pm' 'states
      (state(read_data(pm_phy, pddb_data_type)(PDBR)
              (state(read_data(pm_phy, segment_reg_data_type)(CS)(s'_1))))))
{-2} data(read_data(pm_phy, pddb_data_type)(PDBR)
          (state(read_data(pm_phy, segment_reg_data_type)(CS)(s'_1)))) 'base_addr
      =
      data(read_data(pm_phy, pddb_data_type)(PDBR)
            (state(read_data(pm_phy, segment_reg_data_type)(CS)(s'_2)))) 'base_addr
{-3} pm' 'states(state(read_data(pm_phy, segment_reg_data_type)(CS)(s'_2)))
{-4} pm' 'states(state(read_data(pm_phy, segment_reg_data_type)(CS)(s'_1)))
{-5} data(read_data(pm_phy, segment_reg_data_type)(CS)(s'_1)) =
      data(read_data(pm_phy, segment_reg_data_type)(CS)(s'_2))
{-6} aligned?(bits_per_level + pe_size)
      (offset
        (data(read_data(pm_phy, pddb_data_type)(PDBR)
              (state(read_data(pm_phy, segment_reg_data_type)(CS)
                        (s'_1)))) 'base_addr))

      ^
      OK?(translate(0,
                    data(read_data(pm_phy, pddb_data_type)(PDBR)
                          (state(read_data(pm_phy, seg-
ment_reg_data_type)(CS)
                                (s'_1)))) 'base_addr,
                    addr', ac1',
                    segment_to_priv(data(read_data(pm_phy, seg-
ment_reg_data_type)(CS)
                                         (s'_1))))
                    (state(read_data(pm_phy, pddb_data_type)(PDBR)
                          (state(read_data(pm_phy, seg-
ment_reg_data_type)(CS)
                                (s'_1))))))

      ⊃
      data(translate(0,
                    data(read_data(pm_phy, pddb_data_type)(PDBR)
                          (state(read_data(pm_phy, seg-
ment_reg_data_type)(CS)
                                (s'_1)))) 'base_addr,
                    addr', ac1',
                    segment_to_priv(data(read_data(pm_phy, seg-
ment_reg_data_type)(CS)
                                         (s'_1))))
                    (state(read_data(pm_phy, pddb_data_type)(PDBR)
                          (state(read_data(pm_phy, seg-
ment_reg_data_type)(CS)
                                (s'_1)))))) '2

      < max_linear
{-7} aligned?(bits_per_level + pe_size)
      (offset
        (data(read_data(pm_phy, pddb_data_type)(PDBR)
              (state(read_data(pm_phy, segment_reg_data_type)(CS)
                        (s'_2)))) 'base_addr))

      ^
      OK?(translate(0,
                    data(read_data(pm_phy, pddb_data_type)(PDBR)
                          (state(read_data(pm_phy, seg-
ment_reg_data_type)(CS)
                                (s'_2)))) 'base_addr,
                    addr', ac2',
                    segment_to_priv(data(read_data(pm_phy, seg-
ment_reg_data_type)(CS)
                                         (s'_2))))
                    (state(read_data(pm_phy, pddb_data_type)(PDBR)
                          (state(read_data(pm_phy, seg-

```

C.117 Proofs for Linear_Memory_Properties (challenge-linear.pvs)

Hiding formulas: (-1 -2 -6 -7 -10 -13 2),

Using lemma `pm_unchanged_singleton_linear_resolve_reg_transformers`,

Expanding the definition of `linear_resolve_register_transformers`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Using lemma `unchanged_memory_invariant_invariant`,

Using lemma `expr_transformer_invariant_next_ok`,

Using lemma `pm_states`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `linear_resolve_same_result.1.1.1.1.2`.

linear_resolve_same_result.1.1.1.1.3:

```

{-1} data(read_data(pm_phy, pdbrr_data_type)(PDBR)
      (state(read_data(pm_phy, segment_reg_data_type)(CS)(s'_1))))'base_addr
      =
      data(read_data(pm_phy, pdbrr_data_type)(PDBR)
          (state(read_data(pm_phy, segment_reg_data_type)(CS)(s'_2))))'base_addr
{-2} pm' 'states(state(read_data(pm_phy, segment_reg_data_type)(CS)(s'_2)))
{-3} pm' 'states(state(read_data(pm_phy, segment_reg_data_type)(CS)(s'_1)))
{-4} data(read_data(pm_phy, segment_reg_data_type)(CS)(s'_1)) =
      data(read_data(pm_phy, segment_reg_data_type)(CS)(s'_2))
{-5} aligned?(bits_per_level + pe_size)
      (offset
        (data(read_data(pm_phy, pdbrr_data_type)(PDBR)
              (state(read_data(pm_phy, segment_reg_data_type)(CS)
                (s'_1))))'base_addr))

      ^
      OK?(translate(0,
        data(read_data(pm_phy, pdbrr_data_type)(PDBR)
          (state(read_data(pm_phy, seg-
segment_reg_data_type)(CS)
              (s'_1))))'base_addr,
          addr', ac1',
          segment_to_priv(data(read_data(pm_phy, seg-
segment_reg_data_type)(CS)
              (s'_1))))
            (state(read_data(pm_phy, pdbrr_data_type)(PDBR)
              (state(read_data(pm_phy, seg-
segment_reg_data_type)(CS)
              (s'_1))))))

      ⊃
      data(translate(0,
        data(read_data(pm_phy, pdbrr_data_type)(PDBR)
          (state(read_data(pm_phy, seg-
segment_reg_data_type)(CS)
              (s'_1))))'base_addr,
          addr', ac1',
          segment_to_priv(data(read_data(pm_phy, seg-
segment_reg_data_type)(CS)
              (s'_1))))
            (state(read_data(pm_phy, pdbrr_data_type)(PDBR)
              (state(read_data(pm_phy, seg-
segment_reg_data_type)(CS)
              (s'_1))))))'2

      < max_linear
{-6} aligned?(bits_per_level + pe_size)
      (offset
        (data(read_data(pm_phy, pdbrr_data_type)(PDBR)
              (state(read_data(pm_phy, segment_reg_data_type)(CS)
                (s'_2))))'base_addr))

      ^
      OK?(translate(0,
        data(read_data(pm_phy, pdbrr_data_type)(PDBR)
          (state(read_data(pm_phy, seg-
segment_reg_data_type)(CS)
              (s'_2))))'base_addr,
          addr', ac2',
          segment_to_priv(data(read_data(pm_phy, seg-
segment_reg_data_type)(CS)
              (s'_2))))
            (state(read_data(pm_phy, pdbrr_data_type)(PDBR)
              (state(read_data(pm_phy, seg-
segment_reg_data_type)(CS)
              (s'_2))))))

      ⊃

```

C.117 Proofs for Linear_Memory_Properties (challenge-linear.pvs)

Hiding formulas: (-1 -5 -6 -9 -12 2),

Using lemma `pm_unchanged_singleton_linear_resolve_reg_transformers`,

Expanding the definition of `linear_resolve_register_transformers`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Using lemma `unchanged_memory_invariant_invariant`,

Using lemma `expr_transformer_invariant_next_ok`,

Using lemma `pm_states`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `linear_resolve_same_result.1.1.1.1.3`.

linear_resolve_same_result.1.1.1.1.4:

```

{-1} pm' `states(state(read_data(pm_phy, segment_reg_data_type)(CS)(s'_2)))
{-2} pm' `states(state(read_data(pm_phy, segment_reg_data_type)(CS)(s'_1)))
{-3} data(read_data(pm_phy, segment_reg_data_type)(CS)(s'_1)) =
      data(read_data(pm_phy, segment_reg_data_type)(CS)(s'_2))
{-4} aligned?(bits_per_level + pe_size)
      (offset
        (data(read_data(pm_phy, pdbr_data_type)(PDBR)
              (state(read_data(pm_phy, segment_reg_data_type)(CS)
                        (s'_1)))) `base_addr))
      ^
      OK?(translate(0,
                    data(read_data(pm_phy, pdbr_data_type)(PDBR)
                          (state(read_data(pm_phy, seg-
ment_reg_data_type)(CS)
                                (s'_1)))) `base_addr,
                    addr', ac1',
                    segment_to_priv(data(read_data(pm_phy, seg-
ment_reg_data_type)(CS)
                                          (s'_1))))
                    (state(read_data(pm_phy, pdbr_data_type)(PDBR)
                          (state(read_data(pm_phy, seg-
ment_reg_data_type)(CS)
                                (s'_1))))))
      ⊃
      data(translate(0,
                    data(read_data(pm_phy, pdbr_data_type)(PDBR)
                          (state(read_data(pm_phy, seg-
ment_reg_data_type)(CS)
                                (s'_1)))) `base_addr,
                    addr', ac1',
                    segment_to_priv(data(read_data(pm_phy, seg-
ment_reg_data_type)(CS)
                                          (s'_1))))
                    (state(read_data(pm_phy, pdbr_data_type)(PDBR)
                          (state(read_data(pm_phy, seg-
ment_reg_data_type)(CS)
                                (s'_1)))))) `2
      < max_linear
{-5} aligned?(bits_per_level + pe_size)
      (offset
        (data(read_data(pm_phy, pdbr_data_type)(PDBR)
              (state(read_data(pm_phy, segment_reg_data_type)(CS)
                        (s'_2)))) `base_addr))
      ^
      OK?(translate(0,
                    data(read_data(pm_phy, pdbr_data_type)(PDBR)
                          (state(read_data(pm_phy, seg-
ment_reg_data_type)(CS)
                                (s'_2)))) `base_addr,
                    addr', ac2',
                    segment_to_priv(data(read_data(pm_phy, seg-
ment_reg_data_type)(CS)
                                          (s'_2))))
                    (state(read_data(pm_phy, pdbr_data_type)(PDBR)
                          (state(read_data(pm_phy, seg-
ment_reg_data_type)(CS)
                                (s'_2))))))
      ⊃
      data(translate(0,
                    data(read_data(pm_phy, pdbr_data_type)(PDBR)
                          (state(read_data(pm_phy, seg-
ment_reg_data_type)(CS)
                                (s'_2)))) `base_addr,

```

C.117 Proofs for Linear_Memory_Properties (challenge-linear.pvs)

Hiding formulas: (-4 -5 -8 -11 2),

Expanding the definition of `is_linear_plain_memory?`,

Applying disjunctive simplification to flatten sequent,

Instantiating the top quantifier in -18 with the terms: `(state(read_data(pm_phy, segment_reg_data_type)(CS)(s1!1)) state(read_data(pm_phy, segment_reg_data_type)(CS)(s2!1)))`,

we get 3 subgoals:

linear_resolve_same_result.1.1.1.1.4.1:

```

{-1} pm' `states(state(read_data(pm_phy, segment_reg_data_type)(CS)(s'_2)))
{-2} pm' `states(state(read_data(pm_phy, segment_reg_data_type)(CS)(s'_1)))
{-3} data(read_data(pm_phy, segment_reg_data_type)(CS)(s'_1)) =
      data(read_data(pm_phy, segment_reg_data_type)(CS)(s'_2))
{-4} OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s'_2))
{-5} OK?(read_data(pm_phy, pdir_data_type)(PDBR)
      (state(read_data(pm_phy, segment_reg_data_type)(CS)(s'_2))))
{-6} OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s'_1))
{-7} OK?(read_data(pm_phy, pdir_data_type)(PDBR)
      (state(read_data(pm_phy, segment_reg_data_type)(CS)(s'_1))))
{-8} interpreted_data_type?(segment_reg_data_type)
{-9} interpreted_data_type?(pdir_data_type)
{-10} Mem?(addr' `type_of)
{-11} 0 ≤ addr' `offset
{-12} addr' `offset < max_linear_offset
{-13} union(pm' `ro_addr, pm' `rw_addr)(addr')
{-14} pm' `mem = linear_pm
{-15} ∀ (s: (pm' `states)): OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-16} ∀ (s_1, s_2: (pm' `states)):
      data(read_data(pm_phy, segment_reg_data_type)(CS)(s_1)) =
        data(read_data(pm_phy, segment_reg_data_type)(CS)(s_2))
{-17} ∀ (s: (pm' `states)): OK?(read_data(pm_phy, pdir_data_type)(PDBR)(s))
{-18} data(read_data(pm_phy, pdir_data_type)(PDBR)
      (state(read_data(pm_phy, segment_reg_data_type)(CS)(s'_1))))
      =
      data(read_data(pm_phy, pdir_data_type)(PDBR)
      (state(read_data(pm_phy, segment_reg_data_type)(CS)(s'_2))))
{-19} ∀ (s_1, s_2: (pm' `states)):
      pe_in_pdir_range?(s_1,
        restrict[Address, Memory_Address_4G, boolean]
          ((pm' `ro_addr ∪ pm' `rw_addr)))
      =
      pe_in_pdir_range?(s_2,
        restrict[Address, Memory_Address_4G, boolean]
          ((pm' `ro_addr ∪ pm' `rw_addr)))
      ∧
      pe_in_ptab_range?(s_1,
        restrict[Address, Memory_Address_4G, boolean]
          ((pm' `ro_addr ∪ pm' `rw_addr)))
      =
      pe_in_ptab_range?(s_2,
        restrict[Address, Memory_Address_4G, boolean]
          ((pm' `ro_addr ∪ pm' `rw_addr)))
{-20} ∀ (s: (pm' `states), lvl: Level,
      a:
        (pe_in_pt_range?(s,
          restrict[Address, Memory_Address_4G, boolean]
            ((pm' `ro_addr ∪ pm' `rw_addr)),
          lvl))) :
      OK?(read_data(pm_phy, paging_data_type(lvl))(a)(s))
{-21} ∀ (s_1, s_2: (pm' `states), lvl: Level,
      a:
        (pe_in_pt_range?(s_1,
          restrict[Address, Memory_Address_4G, boolean]
            ((pm' `ro_addr ∪ pm' `rw_addr)),
          lvl))) :
      set_reference(data(read_data(pm_phy, paging_data_type(lvl))(a)(s_1)), Write) =
        set_reference(data(read_data(pm_phy, paging_data_type(lvl))(a)(s_2)), Write)
{-22} plain_memory?(pm_phy)
{-23} pm' `states = pm_phy `states
{-24} (pm' `other_actions ⊆ pm_phy `other_actions)
{-25} (linear_resolve_register_transformers ⊆ pm_phy `other_actions)
{-26} ∀ (a: ((pm' `ro_addr ∪ pm' `rw_addr))) :
      Mem?(type_of(a)) ∧ 0 ≤ offset(a) ∧ offset(a) < max_linear_offset

```

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `linear_resolve_same_result.1.1.1.1.4.1`.

linear_resolve_same_result.1.1.1.1.4.2:

```

{-1} pm' states(state(read_data(pm_phy, segment_reg_data_type)(CS)(s'_2)))
{-2} pm' states(state(read_data(pm_phy, segment_reg_data_type)(CS)(s'_1)))
{-3} data(read_data(pm_phy, segment_reg_data_type)(CS)(s'_1)) =
      data(read_data(pm_phy, segment_reg_data_type)(CS)(s'_2))
{-4} OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s'_2))
{-5} OK?(read_data(pm_phy, pdir_data_type)(PDBR)
          (state(read_data(pm_phy, segment_reg_data_type)(CS)(s'_2))))
{-6} OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s'_1))
{-7} OK?(read_data(pm_phy, pdir_data_type)(PDBR)
          (state(read_data(pm_phy, segment_reg_data_type)(CS)(s'_1))))
{-8} interpreted_data_type?(segment_reg_data_type)
{-9} interpreted_data_type?(pdir_data_type)
{-10} Mem?(addr' type_of)
{-11} 0 ≤ addr' offset
{-12} addr' offset < max_linear_offset
{-13} union(pm' ro_addr, pm' rw_addr)(addr')
{-14} pm' mem = linear_pm
{-15} ∀ (s: (pm' states)): OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-16} ∀ (s1, s2: (pm' states)):
      data(read_data(pm_phy, segment_reg_data_type)(CS)(s1)) =
        data(read_data(pm_phy, segment_reg_data_type)(CS)(s2))
{-17} ∀ (s: (pm' states)): OK?(read_data(pm_phy, pdir_data_type)(PDBR)(s))
{-18} ∀ (s1, s2: (pm' states)):
      pe_in_pdir_range?(s1,
                        restrict[Address, Memory_Address_4G, boolean]
                          ((pm' ro_addr ∪ pm' rw_addr)))
      =
      pe_in_pdir_range?(s2,
                        restrict[Address, Memory_Address_4G, boolean]
                          ((pm' ro_addr ∪ pm' rw_addr)))
      ∧
      pe_in_ptab_range?(s1,
                        restrict[Address, Memory_Address_4G, boolean]
                          ((pm' ro_addr ∪ pm' rw_addr)))
      =
      pe_in_ptab_range?(s2,
                        restrict[Address, Memory_Address_4G, boolean]
                          ((pm' ro_addr ∪ pm' rw_addr)))
{-19} ∀ (s: (pm' states), lvl: Level,
        a:
          (pe_in_pt_range?(s,
                            restrict[Address, Memory_Address_4G, boolean]
                              ((pm' ro_addr ∪ pm' rw_addr)),
                              lvl)))):
      OK?(read_data(pm_phy, paging_data_type(lvl))(a)(s))
{-20} ∀ (s1, s2: (pm' states), lvl: Level,
        a:
          (pe_in_pt_range?(s1,
                            restrict[Address, Memory_Address_4G, boolean]
                              ((pm' ro_addr ∪ pm' rw_addr)),
                              lvl)))):
      set_reference(data(read_data(pm_phy, paging_data_type(lvl))(a)(s1)), Write) =
        set_reference(data(read_data(pm_phy, paging_data_type(lvl))(a)(s2)), Write)
{-21} plain_memory?(pm_phy)
{-22} pm' states = pm_phy states
{-23} (pm' other_actions ⊆ pm_phy other_actions)
{-24} (linear_resolve_register_transformers ⊆ pm_phy other_actions)
{-25} ∀ (a: ((pm' ro_addr ∪ pm' rw_addr))):
      Mem?(type_of(a)) ∧ 0 ≤ offset(a) ∧ offset(a) < max_linear_offset
{-26} ∀ (s: (pm' states)):
      linear_blessed?(s, restrict[Address, Memory_Address_4G, boolean](pm' ro_addr),
                      restrict[Address, Memory_Address_4G, boolean](pm' rw_addr))
{-27} pm' states(s'_1)
{-28} pm' states(s'_2)

```

which is trivially true.

This completes the proof of `linear_resolve_same_result.1.1.1.1.4.2`.

linear_resolve_same_result.1.1.1.1.4.3:

```

{-1} pm' states(state(read_data(pm_phy, segment_reg_data_type)(CS)(s'_2)))
{-2} pm' states(state(read_data(pm_phy, segment_reg_data_type)(CS)(s'_1)))
{-3} data(read_data(pm_phy, segment_reg_data_type)(CS)(s'_1)) =
      data(read_data(pm_phy, segment_reg_data_type)(CS)(s'_2))
{-4} OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s'_2))
{-5} OK?(read_data(pm_phy, pdir_data_type)(PDBR)
      (state(read_data(pm_phy, segment_reg_data_type)(CS)(s'_2))))
{-6} OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s'_1))
{-7} OK?(read_data(pm_phy, pdir_data_type)(PDBR)
      (state(read_data(pm_phy, segment_reg_data_type)(CS)(s'_1))))
{-8} interpreted_data_type?(segment_reg_data_type)
{-9} interpreted_data_type?(pdir_data_type)
{-10} Mem?(addr' type_of)
{-11} 0 ≤ addr' offset
{-12} addr' offset < max_linear_offset
{-13} union(pm' ro_addr, pm' rw_addr)(addr')
{-14} pm' mem = linear_pm
{-15} ∀ (s: (pm' states)): OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-16} ∀ (s1, s2: (pm' states)):
      data(read_data(pm_phy, segment_reg_data_type)(CS)(s1)) =
        data(read_data(pm_phy, segment_reg_data_type)(CS)(s2))
{-17} ∀ (s: (pm' states)): OK?(read_data(pm_phy, pdir_data_type)(PDBR)(s))
{-18} ∀ (s1, s2: (pm' states)):
      pe_in_pdir_range?(s1,
                        restrict[Address, Memory_Address_4G, boolean]
                          ((pm' ro_addr ∪ pm' rw_addr)))
      =
      pe_in_pdir_range?(s2,
                        restrict[Address, Memory_Address_4G, boolean]
                          ((pm' ro_addr ∪ pm' rw_addr)))
      ∧
      pe_in_ptab_range?(s1,
                        restrict[Address, Memory_Address_4G, boolean]
                          ((pm' ro_addr ∪ pm' rw_addr)))
      =
      pe_in_ptab_range?(s2,
                        restrict[Address, Memory_Address_4G, boolean]
                          ((pm' ro_addr ∪ pm' rw_addr)))
{-19} ∀ (s: (pm' states), lvl: Level,
        a:
          (pe_in_pt_range?(s,
                            restrict[Address, Memory_Address_4G, boolean]
                              ((pm' ro_addr ∪ pm' rw_addr)),
                              lvl)))):
      OK?(read_data(pm_phy, paging_data_type(lvl))(a)(s))
{-20} ∀ (s1, s2: (pm' states), lvl: Level,
        a:
          (pe_in_pt_range?(s1,
                            restrict[Address, Memory_Address_4G, boolean]
                              ((pm' ro_addr ∪ pm' rw_addr)),
                              lvl)))):
      set_reference(data(read_data(pm_phy, paging_data_type(lvl))(a)(s1)), Write) =
        set_reference(data(read_data(pm_phy, paging_data_type(lvl))(a)(s2)), Write)
{-21} plain_memory?(pm_phy)
{-22} pm' states = pm_phy states
{-23} (pm' other_actions ⊆ pm_phy other_actions)
{-24} (linear_resolve_register_transformers ⊆ pm_phy other_actions)
{-25} ∀ (a: ((pm' ro_addr ∪ pm' rw_addr))):
      Mem?(type_of(a)) ∧ 0 ≤ offset(a) ∧ offset(a) < max_linear_offset
{-26} ∀ (s: (pm' states)):
      linear_blessed?(s, restrict[Address, Memory_Address_4G, boolean](pm' ro_addr),
                      restrict[Address, Memory_Address_4G, boolean](pm' rw_addr))
{-27} pm' states(s'_1)
{-28} pm' states(s'_2)

```

which is trivially true.

This completes the proof of `linear_resolve_same_result.1.1.1.1.4.3`.

linear_resolve_same_result.1.1.1.2:

```

{-1} pm' 'states(state(read_data(pm_phy, segment_reg_data_type)(CS)(s'_1)))
{-2} data(read_data(pm_phy, segment_reg_data_type)(CS)(s'_1)) =
      data(read_data(pm_phy, segment_reg_data_type)(CS)(s'_2))
{-3} aligned?(bits_per_level + pe_size)
      (offset
        (data(read_data(pm_phy, pdbr_data_type)(PDBR)
              (state(read_data(pm_phy, segment_reg_data_type)(CS)
                        (s'_1)))) 'base_addr))

      ^
      OK?(translate(0,
                    data(read_data(pm_phy, pdbr_data_type)(PDBR)
                          (state(read_data(pm_phy, seg-
segment_reg_data_type)(CS)
                                (s'_1)))) 'base_addr,
                    addr', ac1',
                    segment_to_priv(data(read_data(pm_phy, seg-
segment_reg_data_type)(CS)
                                         (s'_1))))
                    (state(read_data(pm_phy, pdbr_data_type)(PDBR)
                          (state(read_data(pm_phy, seg-
segment_reg_data_type)(CS)
                                (s'_1))))))

      ⊃
      data(translate(0,
                    data(read_data(pm_phy, pdbr_data_type)(PDBR)
                          (state(read_data(pm_phy, seg-
segment_reg_data_type)(CS)
                                (s'_1)))) 'base_addr,
                    addr', ac1',
                    segment_to_priv(data(read_data(pm_phy, seg-
segment_reg_data_type)(CS)
                                         (s'_1))))
                    (state(read_data(pm_phy, pdbr_data_type)(PDBR)
                          (state(read_data(pm_phy, seg-
segment_reg_data_type)(CS)
                                (s'_1)))))) '2

      < max_linear
{-4} aligned?(bits_per_level + pe_size)
      (offset
        (data(read_data(pm_phy, pdbr_data_type)(PDBR)
              (state(read_data(pm_phy, segment_reg_data_type)(CS)
                        (s'_2)))) 'base_addr))

      ^
      OK?(translate(0,
                    data(read_data(pm_phy, pdbr_data_type)(PDBR)
                          (state(read_data(pm_phy, seg-
segment_reg_data_type)(CS)
                                (s'_2)))) 'base_addr,
                    addr', ac2',
                    segment_to_priv(data(read_data(pm_phy, seg-
segment_reg_data_type)(CS)
                                         (s'_2))))
                    (state(read_data(pm_phy, pdbr_data_type)(PDBR)
                          (state(read_data(pm_phy, seg-
segment_reg_data_type)(CS)
                                (s'_2))))))

      ⊃
      data(translate(0,
                    data(read_data(pm_phy, pdbr_data_type)(PDBR)
                          (state(read_data(pm_phy, seg-
segment_reg_data_type)(CS)
                                (s'_2)))) 'base_addr,
                    addr', ac2',

```

C.117 Proofs for Linear_Memory_Properties (challenge-linear.pvs)

Hiding formulas: (-1 -2 -3 -4 -6 -7 -9 -10 2),

Using lemma `pm_unchanged_singleton_linear_resolve_reg_transformers`,

Expanding the definition of `linear_resolve_register_transformers`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Using lemma `unchanged_memory_invariant_invariant`,

Using lemma `expr_transformer_invariant_next_ok`,

Using lemma `pm_states`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `linear_resolve_same_result.1.1.1.2`.

linear_resolve_same_result.1.1.1.3:

```

{-1} data(read_data(pm_phy, segment_reg_data_type)(CS)(s'_1)) =
      data(read_data(pm_phy, segment_reg_data_type)(CS)(s'_2))
{-2} aligned?(bits_per_level + pe_size)
      (offset
        (data(read_data(pm_phy, pdbr_data_type)(PDBR)
              (state(read_data(pm_phy, segment_reg_data_type)(CS)
                        (s'_1))))'base_addr))
      ^
      OK?(translate(0,
                    data(read_data(pm_phy, pdbr_data_type)(PDBR)
                          (state(read_data(pm_phy, seg-
segment_reg_data_type)(CS)
                                (s'_1))))'base_addr,
                    addr', ac1',
                    segment_to_priv(data(read_data(pm_phy, seg-
segment_reg_data_type)(CS)
                                        (s'_1))))
                    (state(read_data(pm_phy, pdbr_data_type)(PDBR)
                          (state(read_data(pm_phy, seg-
segment_reg_data_type)(CS)
                                (s'_1))))))
      ⊃
      data(translate(0,
                    data(read_data(pm_phy, pdbr_data_type)(PDBR)
                          (state(read_data(pm_phy, seg-
segment_reg_data_type)(CS)
                                (s'_1))))'base_addr,
                    addr', ac1',
                    segment_to_priv(data(read_data(pm_phy, seg-
segment_reg_data_type)(CS)
                                        (s'_1))))
                    (state(read_data(pm_phy, pdbr_data_type)(PDBR)
                          (state(read_data(pm_phy, seg-
segment_reg_data_type)(CS)
                                (s'_1))))))'2
      < max_linear
{-3} aligned?(bits_per_level + pe_size)
      (offset
        (data(read_data(pm_phy, pdbr_data_type)(PDBR)
              (state(read_data(pm_phy, segment_reg_data_type)(CS)
                        (s'_2))))'base_addr))
      ^
      OK?(translate(0,
                    data(read_data(pm_phy, pdbr_data_type)(PDBR)
                          (state(read_data(pm_phy, seg-
segment_reg_data_type)(CS)
                                (s'_2))))'base_addr,
                    addr', ac2',
                    segment_to_priv(data(read_data(pm_phy, seg-
segment_reg_data_type)(CS)
                                        (s'_2))))
                    (state(read_data(pm_phy, pdbr_data_type)(PDBR)
                          (state(read_data(pm_phy, seg-
segment_reg_data_type)(CS)
                                (s'_2))))))
      ⊃
      data(translate(0,
                    data(read_data(pm_phy, pdbr_data_type)(PDBR)
                          (state(read_data(pm_phy, seg-
segment_reg_data_type)(CS)
                                (s'_2))))'base_addr,
                    addr', ac2',
                    segment_to_priv(data(read_data(pm_phy, seg-

```

C.117 Proofs for Linear_Memory_Properties (challenge-linear.pvs)

Hiding formulas: (-1 -2 -3 -5 -6 -8 -9 2),

Using lemma `pm_unchanged_singleton_linear_resolve_reg_transformers`,

Expanding the definition of `linear_resolve_register_transformers`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Using lemma `unchanged_memory_invariant_invariant`,

Using lemma `expr_transformer_invariant_next_ok`,

Using lemma `pm_states`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `linear_resolve_same_result.1.1.1.3`.

linear_resolve_same_result.1.1.1.4:

```

{-1}  aligned?(bits_per_level + pe_size)
      (offset
        (data(read_data(pm_phy, pdb_r_data_type)(PDBR)
              (state(read_data(pm_phy, segment_reg_data_type)(CS)
                        (s'_1)))) 'base_addr))

      ^
      OK?(translate(0,
                    data(read_data(pm_phy, pdb_r_data_type)(PDBR)
                          (state(read_data(pm_phy, seg-
ment_reg_data_type)(CS)
                                (s'_1)))) 'base_addr,
                    addr', ac1',
                    segment_to_priv(data(read_data(pm_phy, seg-
ment_reg_data_type)(CS)
                                         (s'_1))))
                    (state(read_data(pm_phy, pdb_r_data_type)(PDBR)
                          (state(read_data(pm_phy, seg-
ment_reg_data_type)(CS)
                                (s'_1))))))

      ⊃
      data(translate(0,
                    data(read_data(pm_phy, pdb_r_data_type)(PDBR)
                          (state(read_data(pm_phy, seg-
ment_reg_data_type)(CS)
                                (s'_1)))) 'base_addr,
                    addr', ac1',
                    segment_to_priv(data(read_data(pm_phy, seg-
ment_reg_data_type)(CS)
                                         (s'_1))))
                    (state(read_data(pm_phy, pdb_r_data_type)(PDBR)
                          (state(read_data(pm_phy, seg-
ment_reg_data_type)(CS)
                                (s'_1)))))) '2

< max_linear
{-2}  aligned?(bits_per_level + pe_size)
      (offset
        (data(read_data(pm_phy, pdb_r_data_type)(PDBR)
              (state(read_data(pm_phy, segment_reg_data_type)(CS)
                        (s'_2)))) 'base_addr))

      ^
      OK?(translate(0,
                    data(read_data(pm_phy, pdb_r_data_type)(PDBR)
                          (state(read_data(pm_phy, seg-
ment_reg_data_type)(CS)
                                (s'_2)))) 'base_addr,
                    addr', ac2',
                    segment_to_priv(data(read_data(pm_phy, seg-
ment_reg_data_type)(CS)
                                         (s'_2))))
                    (state(read_data(pm_phy, pdb_r_data_type)(PDBR)
                          (state(read_data(pm_phy, seg-
ment_reg_data_type)(CS)
                                (s'_2))))))

      ⊃
      data(translate(0,
                    data(read_data(pm_phy, pdb_r_data_type)(PDBR)
                          (state(read_data(pm_phy, seg-
ment_reg_data_type)(CS)
                                (s'_2)))) 'base_addr,
                    addr', ac2',
                    segment_to_priv(data(read_data(pm_phy, seg-
ment_reg_data_type)(CS)
                                         (s'_2))))
                    (s'_2))))

```

C.117 Proofs for Linear_Memory_Properties (challenge-linear.pvs)

Hiding formulas: (-1 -2 -4 -5 -7 -8 2),

Expanding the definition of `is_linear_plain_memory?`,

Applying disjunctive simplification to flatten sequent,

Instantiating the top quantifier in -11 with the terms: (s1!1 s2!1),

we get 3 subgoals:

linear_resolve_same_result.1.1.1.4.1:

	{-1}	OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s ₂))
	{-2}	OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s ₁))
	{-3}	interpreted_data_type?(segment_reg_data_type)
	{-4}	interpreted_data_type?(pdir_data_type)
	{-5}	Mem?(addr' type_of)
	{-6}	0 ≤ addr' offset
	{-7}	addr' offset < max_linear_offset
	{-8}	union(pm' ro_addr, pm' rw_addr)(addr')
	{-9}	pm' mem = linear_pm
	{-10}	∀ (s: (pm' states)): OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s))
	{-11}	data(read_data(pm_phy, segment_reg_data_type)(CS)(s ₁)) = data(read_data(pm_phy, segment_reg_data_type)(CS)(s ₂))
	{-12}	∀ (s: (pm' states)): OK?(read_data(pm_phy, pdir_data_type)(PDIR)(s))
	{-13}	∀ (s ₁ , s ₂ : (pm' states)): data(read_data(pm_phy, pdir_data_type)(PDIR)(s ₁)) = data(read_data(pm_phy, pdir_data_type)(PDIR)(s ₂))
	{-14}	∀ (s ₁ , s ₂ : (pm' states)): pe_in_pdir_range?(s ₁ , restrict[Address, Memory_Address_4G, boolean] ((pm' ro_addr ∪ pm' rw_addr))) = pe_in_pdir_range?(s ₂ , restrict[Address, Memory_Address_4G, boolean] ((pm' ro_addr ∪ pm' rw_addr))) ∧ pe_in_ptab_range?(s ₁ , restrict[Address, Memory_Address_4G, boolean] ((pm' ro_addr ∪ pm' rw_addr))) = pe_in_ptab_range?(s ₂ , restrict[Address, Memory_Address_4G, boolean] ((pm' ro_addr ∪ pm' rw_addr)))
	{-15}	∀ (s: (pm' states), lvl: Level, a: pe_in_pt_range?(s, restrict[Address, Memory_Address_4G, boolean] ((pm' ro_addr ∪ pm' rw_addr)), lvl)): OK?(read_data(pm_phy, paging_data_type(lvl))(a)(s))
	{-16}	∀ (s ₁ , s ₂ : (pm' states), lvl: Level, a: pe_in_pt_range?(s ₁ , restrict[Address, Memory_Address_4G, boolean] ((pm' ro_addr ∪ pm' rw_addr)), lvl)): set_reference(data(read_data(pm_phy, paging_data_type(lvl))(a)(s ₁)), Write) = set_reference(data(read_data(pm_phy, paging_data_type(lvl))(a)(s ₂)), Write)
	{-17}	plain_memory?(pm_phy)
	{-18}	pm' states = pm_phy' states
	{-19}	(pm' other_actions ⊆ pm_phy' other_actions)
	{-20}	(linear_resolve_register_transformers ⊆ pm_phy' other_actions)
	{-21}	∀ (a: ((pm' ro_addr ∪ pm' rw_addr))): Mem?(type_of(a)) ∧ 0 ≤ offset(a) ∧ offset(a) < max_linear_offset
	{-22}	∀ (s: (pm' states)): linear_blessed?(s, restrict[Address, Memory_Address_4G, boolean](pm' ro_addr), restrict[Address, Memory_Address_4G, boolean](pm' rw_addr))
2000	{-23}	pm' states(s ₁)
	{-24}	pm' states(s ₂)
	{1}	data(read_data(pm_phy, segment_reg_data_type)(CS)(s ₁)) = data(read_data(pm_phy, segment_reg_data_type)(CS)(s ₂))

which is trivially true.

This completes the proof of `linear_resolve_same_result.1.1.1.4.1`.

linear_resolve_same_result.1.1.1.4.2:

	{-1}	OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s ₂))
	{-2}	OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s ₁))
	{-3}	interpreted_data_type?(segment_reg_data_type)
	{-4}	interpreted_data_type?(pdir_data_type)
	{-5}	Mem?(addr' type_of)
	{-6}	0 ≤ addr' offset
	{-7}	addr' offset < max_linear_offset
	{-8}	union(pm' ro_addr, pm' rw_addr)(addr')
	{-9}	pm' mem = linear_pm
	{-10}	∀ (s: (pm' states)): OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s))
	{-11}	∀ (s: (pm' states)): OK?(read_data(pm_phy, pdir_data_type)(PDIR)(s))
	{-12}	∀ (s ₁ , s ₂ : (pm' states)): data(read_data(pm_phy, pdir_data_type)(PDIR)(s ₁)) = data(read_data(pm_phy, pdir_data_type)(PDIR)(s ₂))
	{-13}	∀ (s ₁ , s ₂ : (pm' states)): pe_in_pdir_range?(s ₁ , restrict[Address, Memory_Address_4G, boolean] ((pm' ro_addr ∪ pm' rw_addr))) = pe_in_pdir_range?(s ₂ , restrict[Address, Memory_Address_4G, boolean] ((pm' ro_addr ∪ pm' rw_addr))) ∧ pe_in_ptab_range?(s ₁ , restrict[Address, Memory_Address_4G, boolean] ((pm' ro_addr ∪ pm' rw_addr))) = pe_in_ptab_range?(s ₂ , restrict[Address, Memory_Address_4G, boolean] ((pm' ro_addr ∪ pm' rw_addr)))
	{-14}	∀ (s: (pm' states), lvl: Level, a: (pe_in_pt_range?(s, restrict[Address, Memory_Address_4G, boolean] ((pm' ro_addr ∪ pm' rw_addr)), lvl))): OK?(read_data(pm_phy, paging_data_type(lvl))(a)(s))
	{-15}	∀ (s ₁ , s ₂ : (pm' states), lvl: Level, a: (pe_in_pt_range?(s ₁ , restrict[Address, Memory_Address_4G, boolean] ((pm' ro_addr ∪ pm' rw_addr)), lvl))): set_reference(data(read_data(pm_phy, paging_data_type(lvl))(a)(s ₁)), Write) = set_reference(data(read_data(pm_phy, paging_data_type(lvl))(a)(s ₂)), Write)
	{-16}	plain_memory?(pm_phy)
	{-17}	pm' states = pm_phy states
	{-18}	(pm' other_actions ⊆ pm_phy other_actions)
	{-19}	(linear_resolve_register_transformers ⊆ pm_phy other_actions)
	{-20}	∀ (a: ((pm' ro_addr ∪ pm' rw_addr))): Mem?(type_of(a)) ∧ 0 ≤ offset(a) ∧ offset(a) < max_linear_offset
	{-21}	∀ (s: (pm' states)): linear_blessed?(s, restrict[Address, Memory_Address_4G, boolean](pm' ro_addr), restrict[Address, Memory_Address_4G, boolean](pm' rw_addr))
2002	{-22}	pm' states(s ₁)
	{-23}	pm' states(s ₂)
	{1}	pm' states(s ₂)
	{2}	data(read_data(pm_phy, segment_reg_data_type)(CS)(s ₁)) = data(read_data(pm_phy, segment_reg_data_type)(CS)(s ₂))

which is trivially true.

This completes the proof of `linear_resolve_same_result.1.1.1.4.2`.

linear_resolve_same_result.1.1.1.4.3:

{-1}	OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s ₂))
{-2}	OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s ₁))
{-3}	interpreted_data_type?(segment_reg_data_type)
{-4}	interpreted_data_type?(pdir_data_type)
{-5}	Mem?(addr' type_of)
{-6}	0 ≤ addr' offset
{-7}	addr' offset < max_linear_offset
{-8}	union(pm' ro_addr, pm' rw_addr)(addr')
{-9}	pm' mem = linear_pm
{-10}	∀ (s: (pm' states)): OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-11}	∀ (s: (pm' states)): OK?(read_data(pm_phy, pdir_data_type)(PDIR)(s))
{-12}	∀ (s ₁ , s ₂ : (pm' states)): data(read_data(pm_phy, pdir_data_type)(PDIR)(s ₁)) = data(read_data(pm_phy, pdir_data_type)(PDIR)(s ₂))
{-13}	∀ (s ₁ , s ₂ : (pm' states)): pe_in_pdir_range?(s ₁ , restrict[Address, Memory_Address_4G, boolean] ((pm' ro_addr ∪ pm' rw_addr))) = pe_in_pdir_range?(s ₂ , restrict[Address, Memory_Address_4G, boolean] ((pm' ro_addr ∪ pm' rw_addr))) ∧ pe_in_ptab_range?(s ₁ , restrict[Address, Memory_Address_4G, boolean] ((pm' ro_addr ∪ pm' rw_addr))) = pe_in_ptab_range?(s ₂ , restrict[Address, Memory_Address_4G, boolean] ((pm' ro_addr ∪ pm' rw_addr)))
{-14}	∀ (s: (pm' states), lvl: Level, a: (pe_in_pt_range?(s, restrict[Address, Memory_Address_4G, boolean] ((pm' ro_addr ∪ pm' rw_addr)), lvl))): OK?(read_data(pm_phy, paging_data_type(lvl))(a)(s))
{-15}	∀ (s ₁ , s ₂ : (pm' states), lvl: Level, a: (pe_in_pt_range?(s ₁ , restrict[Address, Memory_Address_4G, boolean] ((pm' ro_addr ∪ pm' rw_addr)), lvl))): set_reference(data(read_data(pm_phy, paging_data_type(lvl))(a)(s ₁)), Write) = set_reference(data(read_data(pm_phy, paging_data_type(lvl))(a)(s ₂)), Write)
{-16}	plain_memory?(pm_phy)
{-17}	pm' states = pm_phy states
{-18}	(pm' other_actions ⊆ pm_phy other_actions)
{-19}	(linear_resolve_register_transformers ⊆ pm_phy other_actions)
{-20}	∀ (a: ((pm' ro_addr ∪ pm' rw_addr))): Mem?(type_of(a)) ∧ 0 ≤ offset(a) ∧ offset(a) < max_linear_offset
{-21}	∀ (s: (pm' states)): linear_blessed?(s, restrict[Address, Memory_Address_4G, boolean](pm' ro_addr), restrict[Address, Memory_Address_4G, boolean](pm' rw_addr))
{-22}	pm' states(s ₁)
{-23}	pm' states(s ₂)
2004	{1} pm' states(s ₁)
	{2} data(read_data(pm_phy, segment_reg_data_type)(CS)(s ₁)) = data(read_data(pm_phy, segment_reg_data_type)(CS)(s ₂))

which is trivially true.

This completes the proof of `linear_resolve_same_result.1.1.1.4.3`.

`linear_resolve_same_result.1.1.2`:

{-1}	$\begin{aligned} & \text{OK?}(\text{read_data}(\text{pm_phy}, \text{segment_reg_data_type})(\text{CS})(s'_1)) \wedge \\ & \text{OK?}(\text{read_data}(\text{pm_phy}, \text{pdbr_data_type})(\text{PDBR}) \\ & \quad (\text{state}(\text{read_data}(\text{pm_phy}, \text{segment_reg_data_type})(\text{CS})(s'_1)))) \\ & \wedge \\ & \text{OK?}(\text{translate}(0, \\ & \quad \text{data}(\text{read_data}(\text{pm_phy}, \text{pdbr_data_type})(\text{PDBR}) \\ & \quad \quad (\text{state}(\text{read_data}(\text{pm_phy}, \text{segment_reg_data_type})(\text{CS}) \\ & \quad \quad \quad (s'_1)))) \text{'base_addr}, \\ & \quad \text{addr}', \text{ac1}', \\ & \quad \text{segment_to_priv}(\text{data}(\text{read_data}(\text{pm_phy}, \text{segment_reg_data_type})(\text{CS}) \\ & \quad \quad \quad (s'_1)))) \\ & \quad (\text{state}(\text{read_data}(\text{pm_phy}, \text{pdbr_data_type})(\text{PDBR}) \\ & \quad \quad (\text{state}(\text{read_data}(\text{pm_phy}, \text{segment_reg_data_type})(\text{CS}) \\ & \quad \quad \quad (s'_1)))))) \end{aligned}$
{-2}	$\text{interpreted_data_type?}(\text{segment_reg_data_type})$
{-3}	$\text{interpreted_data_type?}(\text{pdbr_data_type})$
{-4}	$\text{Mem?}(\text{addr}' \text{'type_of})$
{-5}	$0 \leq \text{addr}' \text{'offset}$
{-6}	$\text{addr}' \text{'offset} < \text{max_linear_offset}$
{-7}	$\text{union}(\text{pm}' \text{'ro_addr}, \text{pm}' \text{'rw_addr})(\text{addr}')$
{-8}	$\text{is_linear_plain_memory?}(\text{pm}')$
{-9}	$\text{pm}' \text{'states}(s'_1)$
{-10}	$\text{pm}' \text{'states}(s'_2)$
{-11}	$\text{OK?}(\text{linear_resolve}(\text{addr}', \text{ac1}')(s'_1))$
{-12}	$\text{OK?}(\text{linear_resolve}(\text{addr}', \text{ac2}')(s'_2))$
{1}	$\begin{aligned} & \text{OK?}(\text{read_data}(\text{pm_phy}, \text{segment_reg_data_type})(\text{CS})(s'_2)) \wedge \\ & \text{OK?}(\text{read_data}(\text{pm_phy}, \text{pdbr_data_type})(\text{PDBR}) \\ & \quad (\text{state}(\text{read_data}(\text{pm_phy}, \text{segment_reg_data_type})(\text{CS})(s'_2)))) \\ & \wedge \\ & \text{OK?}(\text{translate}(0, \\ & \quad \text{data}(\text{read_data}(\text{pm_phy}, \text{pdbr_data_type})(\text{PDBR}) \\ & \quad \quad (\text{state}(\text{read_data}(\text{pm_phy}, \text{segment_reg_data_type})(\text{CS}) \\ & \quad \quad \quad (s'_2)))) \text{'base_addr}, \\ & \quad \text{addr}', \text{ac2}', \\ & \quad \text{segment_to_priv}(\text{data}(\text{read_data}(\text{pm_phy}, \text{segment_reg_data_type})(\text{CS}) \\ & \quad \quad \quad (s'_2)))) \\ & \quad (\text{state}(\text{read_data}(\text{pm_phy}, \text{pdbr_data_type})(\text{PDBR}) \\ & \quad \quad (\text{state}(\text{read_data}(\text{pm_phy}, \text{segment_reg_data_type})(\text{CS}) \\ & \quad \quad \quad (s'_2)))))) \end{aligned}$
{2}	$\text{data}(\text{linear_resolve}(\text{addr}', \text{ac1}')(s'_1)) = \text{data}(\text{linear_resolve}(\text{addr}', \text{ac2}')(s'_2))$

Hiding formulas: (-1 -8 2),

Expanding the definition of `linear_resolve`,

Expanding the definition of `##`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `linear_resolve_same_result.1.1.2`.

`linear_resolve_same_result.1.1.3`:

<pre> {-1} OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s'_1)) ^ OK?(read_data(pm_phy, pdbr_data_type)(PDBR) (state(read_data(pm_phy, segment_reg_data_type)(CS)(s'_1)))) ^ OK?(translate(0, data(read_data(pm_phy, pdbr_data_type)(PDBR) (state(read_data(pm_phy, seg- segment_reg_data_type)(CS) (s'_1)))) 'base_addr, addr', ac1', segment_to_priv(data(read_data(pm_phy, seg- segment_reg_data_type)(CS) (s'_1)))) (state(read_data(pm_phy, pdbr_data_type)(PDBR) (state(read_data(pm_phy, seg- segment_reg_data_type)(CS) (s'_1))))))) {-2} interpreted_data_type?(segment_reg_data_type) {-3} interpreted_data_type?(pdbr_data_type) {-4} Mem?(addr' 'type_of) {-5} 0 ≤ addr' 'offset {-6} addr' 'offset < max_linear_offset {-7} union(pm' 'ro_addr, pm' 'rw_addr)(addr') {-8} is_linear_plain_memory?(pm') {-9} pm' 'states(s'_1) {-10} pm' 'states(s'_2) {-11} OK?(linear_resolve(addr', ac1')(s'_1)) {-12} OK?(linear_resolve(addr', ac2')(s'_2)) </pre>	<pre> {1} OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s'_2)) ^ OK?(read_data(pm_phy, pdbr_data_type)(PDBR) (state(read_data(pm_phy, segment_reg_data_type)(CS)(s'_2)))) ⊃ OK?[Physical_memory, Pdbr_type] (read_data[Physical_memory, Pdbr_type] (pm_phy, pdbr_data_type)(PDBR) (state[Physical_memory, Segment_Reg_type] (read_data[Physical_memory, Segment_Reg_type] (pm_phy, segment_reg_data_type)(CS)(s'_2)))) ∨ Exception?[Physical_memory, Pdbr_type] (read_data[Physical_memory, Pdbr_type] (pm_phy, pdbr_data_type)(PDBR) (state[Physical_memory, Segment_Reg_type] (read_data[Physical_memory, Segment_Reg_type] (pm_phy, segment_reg_data_type)(CS)(s'_2)))) {2} data(linear_resolve(addr', ac1')(s'_1)) = data(linear_resolve(addr', ac2')(s'_2)) </pre>
---	---

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `linear_resolve_same_result.1.1.3`.

linear_resolve_same_result.1.1.4:

{-1}	$\begin{aligned} & \text{OK?}(\text{read_data}(\text{pm_phy}, \text{segment_reg_data_type})(\text{CS})(s'_1)) \wedge \\ & \text{OK?}(\text{read_data}(\text{pm_phy}, \text{pdbr_data_type})(\text{PDBR}) \\ & \quad (\text{state}(\text{read_data}(\text{pm_phy}, \text{segment_reg_data_type})(\text{CS})(s'_1)))) \\ & \wedge \\ & \text{OK?}(\text{translate}(0, \\ & \quad \text{data}(\text{read_data}(\text{pm_phy}, \text{pdbr_data_type})(\text{PDBR}) \\ & \quad \quad (\text{state}(\text{read_data}(\text{pm_phy}, \text{segment_reg_data_type})(\text{CS}) \\ & \quad \quad \quad (s'_1)))) \text{'base_addr}, \\ & \quad \text{addr}', \text{ac1}', \\ & \quad \text{segment_to_priv}(\text{data}(\text{read_data}(\text{pm_phy}, \text{segment_reg_data_type})(\text{CS}) \\ & \quad \quad \quad (s'_1)))) \\ & \quad (\text{state}(\text{read_data}(\text{pm_phy}, \text{pdbr_data_type})(\text{PDBR}) \\ & \quad \quad (\text{state}(\text{read_data}(\text{pm_phy}, \text{segment_reg_data_type})(\text{CS}) \\ & \quad \quad \quad (s'_1)))))) \end{aligned}$
{-2}	interpreted_data_type?(segment_reg_data_type)
{-3}	interpreted_data_type?(pdbr_data_type)
{-4}	Mem?(addr' type_of)
{-5}	$0 \leq \text{addr}' \text{'offset}$
{-6}	$\text{addr}' \text{'offset} < \text{max_linear_offset}$
{-7}	$\text{union}(\text{pm}' \text{'ro_addr}, \text{pm}' \text{'rw_addr})(\text{addr}')$
{-8}	is_linear_plain_memory?(pm')
{-9}	pm' states(s'_1)
{-10}	pm' states(s'_2)
{-11}	OK?(linear_resolve(addr', ac1')(s'_1))
{-12}	OK?(linear_resolve(addr', ac2')(s'_2))
{1}	$\begin{aligned} & \text{OK?}(\text{read_data}(\text{pm_phy}, \text{segment_reg_data_type})(\text{CS})(s'_2)) \wedge \\ & \text{OK?}(\text{read_data}(\text{pm_phy}, \text{pdbr_data_type})(\text{PDBR}) \\ & \quad (\text{state}(\text{read_data}(\text{pm_phy}, \text{segment_reg_data_type})(\text{CS})(s'_2)))) \\ & \supset \\ & \text{OK?}[\text{Physical_memory}, \text{Segment_Reg_type}] \\ & \quad (\text{read_data}[\text{Physical_memory}, \text{Segment_Reg_type}] \\ & \quad \quad (\text{pm_phy}, \text{segment_reg_data_type})(\text{CS})(s'_2)) \\ & \vee \\ & \text{Exception?}[\text{Physical_memory}, \text{Segment_Reg_type}] \\ & \quad (\text{read_data}[\text{Physical_memory}, \text{Segment_Reg_type}] \\ & \quad \quad (\text{pm_phy}, \text{segment_reg_data_type})(\text{CS})(s'_2)) \end{aligned}$
{2}	$\text{data}(\text{linear_resolve}(\text{addr}', \text{ac1}')(s'_1)) = \text{data}(\text{linear_resolve}(\text{addr}', \text{ac2}')(s'_2))$

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of linear_resolve_same_result.1.1.4.

linear_resolve_same_result.1.1.5:

```

{-1} OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s1)) ∧
      OK?(read_data(pm_phy, pdbr_data_type)(PDBR)
            (state(read_data(pm_phy, segment_reg_data_type)(CS)(s1)))
        ∧
      OK?(translate(0,
                    data(read_data(pm_phy, pdbr_data_type)(PDBR)
                          (state(read_data(pm_phy, segment_reg_data_type)(CS)
                                          (s1)))'base_addr,
                          addr', ac1',
                          segment_to_priv(data(read_data(pm_phy, segment_reg_data_type)(CS)
                                                (s1)))
                                          (state(read_data(pm_phy, pdbr_data_type)(PDBR)
                                                (state(read_data(pm_phy, segment_reg_data_type)(CS)
                                                                (s1))))))
                                          (s1))))))
{-2} interpreted_data_type?(segment_reg_data_type)
{-3} interpreted_data_type?(pdbr_data_type)
{-4} Mem?(addr' 'type_of)
{-5} 0 ≤ addr' 'offset
{-6} addr' 'offset < max_linear_offset
{-7} union(pm' 'ro_addr, pm' 'rw_addr)(addr')
{-8} is_linear_plain_memory?(pm')
{-9} pm' 'states(s1)
{-10} pm' 'states(s2)
{-11} OK?(linear_resolve(addr', ac1')(s1))
{-12} OK?(linear_resolve(addr', ac2')(s2))
-----
{1} OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s2)) ⊃
      OK?[Physical_memory, Segment_Reg_type]
        (read_data[Physical_memory, Segment_Reg_type]
          (pm_phy, segment_reg_data_type)(CS)(s2))
      ∨
      Exception?[Physical_memory, Segment_Reg_type]
        (read_data[Physical_memory, Segment_Reg_type]
          (pm_phy, segment_reg_data_type)(CS)(s2))
{2} data(linear_resolve(addr', ac1')(s1)) = data(linear_resolve(addr', ac2')(s2))

```

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of linear_resolve_same_result.1.1.5.

linear_resolve_same_result.1.2:

{-1}	interpreted_data_type?(segment_reg_data_type)
{-2}	interpreted_data_type?(pdbr_data_type)
{-3}	Mem?(addr' type_of)
{-4}	$0 \leq \text{addr}' \text{ offset}$
{-5}	$\text{addr}' \text{ offset} < \text{max_linear_offset}$
{-6}	union(pm' ro_addr, pm' rw_addr)(addr')
{-7}	is_linear_plain_memory?(pm')
{-8}	pm' states(s'_1)
{-9}	pm' states(s'_2)
{-10}	OK?(linear_resolve(addr', ac1')(s'_1))
{-11}	OK?(linear_resolve(addr', ac2')(s'_2))
{1}	$\begin{aligned} & \text{OK?}(\text{read_data}(\text{pm_phy}, \text{segment_reg_data_type})(\text{CS})(s'_1)) \wedge \\ & \quad \text{OK?}(\text{read_data}(\text{pm_phy}, \text{pdbr_data_type})(\text{PDBR}) \\ & \quad \quad (\text{state}(\text{read_data}(\text{pm_phy}, \text{segment_reg_data_type})(\text{CS})(s'_1)))) \\ & \wedge \\ & \quad \text{OK?}(\text{translate}(0, \\ & \quad \quad \text{data}(\text{read_data}(\text{pm_phy}, \text{pdbr_data_type})(\text{PDBR}) \\ & \quad \quad \quad (\text{state}(\text{read_data}(\text{pm_phy}, \text{segment_reg_data_type})(\text{CS}) \\ & \quad \quad \quad \quad (s'_1)))) \text{'base_addr}, \\ & \quad \quad \text{addr}', \text{ac1}', \\ & \quad \quad \text{segment_to_priv}(\text{data}(\text{read_data}(\text{pm_phy}, \text{segment_reg_data_type})(\text{CS}) \\ & \quad \quad \quad (s'_1)))) \\ & \quad \quad (\text{state}(\text{read_data}(\text{pm_phy}, \text{pdbr_data_type})(\text{PDBR}) \\ & \quad \quad \quad (\text{state}(\text{read_data}(\text{pm_phy}, \text{segment_reg_data_type})(\text{CS}) \\ & \quad \quad \quad \quad (s'_1)))))) \\ & \quad \quad (s'_1)))))) \end{aligned}$
{2}	$\text{data}(\text{linear_resolve}(\text{addr}', \text{ac1}')(\mathbf{s}'_1)) = \text{data}(\text{linear_resolve}(\text{addr}', \text{ac2}')(\mathbf{s}'_2))$

Hiding formulas: (-8 2),

Expanding the definition of linear_resolve,

Expanding the definition of ##,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of linear_resolve_same_result.1.2.

C Proof scripts

linear_resolve_same_result.1.3:

{-1}	interpreted_data_type?(segment_reg_data_type)
{-2}	interpreted_data_type?(pdbr_data_type)
{-3}	Mem?(addr' 'type_of)
{-4}	$0 \leq \text{addr}' \text{'offset}$
{-5}	$\text{addr}' \text{'offset} < \text{max_linear_offset}$
{-6}	union(pm' 'ro_addr, pm' 'rw_addr)(addr')
{-7}	is_linear_plain_memory?(pm')
{-8}	pm' 'states(s'_1)
{-9}	pm' 'states(s'_2)
{-10}	OK?(linear_resolve(addr', ac1')(s'_1))
{-11}	OK?(linear_resolve(addr', ac2')(s'_2))
{1}	$\text{OK?}(\text{read_data}(\text{pm_phy}, \text{segment_reg_data_type})(\text{CS})(s'_1)) \wedge$ $\text{OK?}(\text{read_data}(\text{pm_phy}, \text{pdbr_data_type})(\text{PDBR})$ $(\text{state}(\text{read_data}(\text{pm_phy}, \text{segment_reg_data_type})(\text{CS})(s'_1))))$ \supset $\text{OK?}[\text{Physical_memory}, \text{Pdbr_type}]$ $(\text{read_data}[\text{Physical_memory}, \text{Pdbr_type}]$ $(\text{pm_phy}, \text{pdbr_data_type})(\text{PDBR})$ $(\text{state}[\text{Physical_memory}, \text{Segment_Reg_type}]$ $(\text{read_data}[\text{Physical_memory}, \text{Segment_Reg_type}]$ $(\text{pm_phy}, \text{segment_reg_data_type})(\text{CS})(s'_1))))$ \vee $\text{Exception?}[\text{Physical_memory}, \text{Pdbr_type}]$ $(\text{read_data}[\text{Physical_memory}, \text{Pdbr_type}]$ $(\text{pm_phy}, \text{pdbr_data_type})(\text{PDBR})$ $(\text{state}[\text{Physical_memory}, \text{Segment_Reg_type}]$ $(\text{read_data}[\text{Physical_memory}, \text{Segment_Reg_type}]$ $(\text{pm_phy}, \text{segment_reg_data_type})(\text{CS})(s'_1))))$
{2}	data(linear_resolve(addr', ac1')(s'_1)) = data(linear_resolve(addr', ac2')(s'_2))

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of linear_resolve_same_result.1.3.

linear_resolve_same_result.1.4:

{-1}	interpreted_data_type?(segment_reg_data_type)
{-2}	interpreted_data_type?(pdbr_data_type)
{-3}	Mem?(addr' type_of)
{-4}	$0 \leq \text{addr}' \text{ offset}$
{-5}	$\text{addr}' \text{ offset} < \text{max_linear_offset}$
{-6}	$\text{union}(\text{pm}' \text{ ro_addr}, \text{pm}' \text{ rw_addr})(\text{addr}')$
{-7}	is_linear_plain_memory?(pm')
{-8}	pm' states(s'_1)
{-9}	pm' states(s'_2)
{-10}	OK?(linear_resolve(addr', ac1')(s'_1))
{-11}	OK?(linear_resolve(addr', ac2')(s'_2))
{1}	$\text{OK?}(\text{read_data}(\text{pm_phy}, \text{segment_reg_data_type})(\text{CS})(s'_1)) \wedge$ $\text{OK?}(\text{read_data}(\text{pm_phy}, \text{pdbr_data_type})(\text{PDBR})$ $(\text{state}(\text{read_data}(\text{pm_phy}, \text{segment_reg_data_type})(\text{CS})(s'_1))))$ \supset $\text{OK?}[\text{Physical_memory}, \text{Segment_Reg_type}]$ $(\text{read_data}[\text{Physical_memory}, \text{Segment_Reg_type}]$ $(\text{pm_phy}, \text{segment_reg_data_type})(\text{CS})(s'_1))$ \vee $\text{Exception?}[\text{Physical_memory}, \text{Segment_Reg_type}]$ $(\text{read_data}[\text{Physical_memory}, \text{Segment_Reg_type}]$ $(\text{pm_phy}, \text{segment_reg_data_type})(\text{CS})(s'_1))$
{2}	$\text{data}(\text{linear_resolve}(\text{addr}', \text{ac1}')(\mathbf{s}'_1)) = \text{data}(\text{linear_resolve}(\text{addr}', \text{ac2}')(\mathbf{s}'_2))$

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of linear_resolve_same_result.1.4.

linear_resolve_same_result.1.5:

{-1}	interpreted_data_type?(segment_reg_data_type)
{-2}	interpreted_data_type?(pdbr_data_type)
{-3}	Mem?(addr' type_of)
{-4}	$0 \leq \text{addr}' \text{ offset}$
{-5}	$\text{addr}' \text{ offset} < \text{max_linear_offset}$
{-6}	$\text{union}(\text{pm}' \text{ ro_addr}, \text{pm}' \text{ rw_addr})(\text{addr}')$
{-7}	is_linear_plain_memory?(pm')
{-8}	pm' states(s'_1)
{-9}	pm' states(s'_2)
{-10}	OK?(linear_resolve(addr', ac1')(s'_1))
{-11}	OK?(linear_resolve(addr', ac2')(s'_2))
{1}	$\text{OK?}(\text{read_data}(\text{pm_phy}, \text{segment_reg_data_type})(\text{CS})(s'_1)) \supset$ $\text{OK?}[\text{Physical_memory}, \text{Segment_Reg_type}]$ $(\text{read_data}[\text{Physical_memory}, \text{Segment_Reg_type}]$ $(\text{pm_phy}, \text{segment_reg_data_type})(\text{CS})(s'_1))$ \vee $\text{Exception?}[\text{Physical_memory}, \text{Segment_Reg_type}]$ $(\text{read_data}[\text{Physical_memory}, \text{Segment_Reg_type}]$ $(\text{pm_phy}, \text{segment_reg_data_type})(\text{CS})(s'_1))$
{2}	$\text{data}(\text{linear_resolve}(\text{addr}', \text{ac1}')(\mathbf{s}'_1)) = \text{data}(\text{linear_resolve}(\text{addr}', \text{ac2}')(\mathbf{s}'_2))$

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of linear_resolve_same_result.1.5.

`linear_resolve_same_result.2:`

{-1}	<code>interpreted_data_type?(pdbr_data_type)</code>
{-2}	<code>Mem?(addr' 'type_of)</code>
{-3}	<code>0 ≤ addr' 'offset</code>
{-4}	<code>addr' 'offset < max_linear_offset</code>
{-5}	<code>union(pm' 'ro_addr, pm' 'rw_addr)(addr')</code>
{-6}	<code>is_linear_plain_memory?(pm')</code>
{-7}	<code>pm' 'states(s₁)</code>
{-8}	<code>pm' 'states(s₂)</code>
{-9}	<code>OK?(linear_resolve(addr', ac1')(s₁))</code>
{-10}	<code>OK?(linear_resolve(addr', ac2')(s₂))</code>
{1}	<code>interpreted_data_type?(segment_reg_data_type)</code>
{2}	<code>data(linear_resolve(addr', ac1')(s₁)) = data(linear_resolve(addr', ac2')(s₂))</code>

Adding type constraints for `segment_reg_data_type`,
Expanding the definition of `pod_data_type?`,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `linear_resolve_same_result.2`.

`linear_resolve_same_result.3:`

{-1}	<code>Mem?(addr' 'type_of)</code>
{-2}	<code>0 ≤ addr' 'offset</code>
{-3}	<code>addr' 'offset < max_linear_offset</code>
{-4}	<code>union(pm' 'ro_addr, pm' 'rw_addr)(addr')</code>
{-5}	<code>is_linear_plain_memory?(pm')</code>
{-6}	<code>pm' 'states(s₁)</code>
{-7}	<code>pm' 'states(s₂)</code>
{-8}	<code>OK?(linear_resolve(addr', ac1')(s₁))</code>
{-9}	<code>OK?(linear_resolve(addr', ac2')(s₂))</code>
{1}	<code>interpreted_data_type?(pdbr_data_type)</code>
{2}	<code>data(linear_resolve(addr', ac1')(s₁)) = data(linear_resolve(addr', ac2')(s₂))</code>

Adding type constraints for `pdbr_data_type`,
Expanding the definition of `pod_data_type?`,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `linear_resolve_same_result.3`.
Q.E.D.

C.117.56 Linear_Memory_Properties.delta_memory_address

Terse proof for `delta_memory_address`.

`delta_memory_address:`

{1}	$\forall (\text{addr}: \text{Memory_Address_4G}, \delta: \text{nat}):$ $\text{rem}(\text{expt}(2, \text{min_page}))(\text{offset}(\text{addr})) + \delta < \text{expt}(2, \text{min_page}) \supset$ $\text{offset}(\text{addr} + \delta) < \text{max_linear_offset}$
-----	--

Repeatedly Skolemizing and flattening,
Expanding the definition of `+`,
Using lemma `aligned_add_below`,
we get 2 subgoals:

delta_memory_address.1:

{-1}	$\text{min_page} \leq \text{bus_width} \wedge$ $\text{offset}(\text{addr}') - \text{rem}(\text{expt}(2, \text{min_page}))(\text{offset}(\text{addr}')) < \text{expt}(2, \text{bus_width})$ \wedge $\text{aligned}?(\text{min_page})(\text{offset}(\text{addr}') - \text{rem}(\text{expt}(2, \text{min_page}))(\text{offset}(\text{addr}')))$ $\wedge \text{rem}(\text{expt}(2, \text{min_page}))(\text{offset}(\text{addr}')) + \text{delta}' < \text{expt}(2, \text{min_page})$ \supset $\text{offset}(\text{addr}') - \text{rem}(\text{expt}(2, \text{min_page}))(\text{offset}(\text{addr}')) + \text{rem}(\text{expt}(2, \text{min_page}))(\text{offset}(\text{addr}')) + \text{delta}'$ $< \text{expt}(2, \text{bus_width})$
{-2}	$\text{Mem}?(\text{addr}' \text{'type_of})$
{-3}	$0 \leq \text{addr}' \text{'offset}$
{-4}	$\text{addr}' \text{'offset} < \text{max_linear_offset}$
{-5}	$\text{delta}' \geq 0$
{-6}	$\text{rem}(\text{expt}(2, \text{min_page}))(\text{offset}(\text{addr}')) + \text{delta}' < \text{expt}(2, \text{min_page})$
{1}	$\text{addr}' \text{'offset} + \text{delta}' < \text{max_linear_offset}$

Installing automatic rewrites from: (min_page! bus_width! max_linear_offset!)

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Using lemma rem_def2,

Expanding the definition of aligned?,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of delta_memory_address.1.

delta_memory_address.2:

{-1}	$\text{Mem}?(\text{addr}' \text{'type_of})$
{-2}	$0 \leq \text{addr}' \text{'offset}$
{-3}	$\text{addr}' \text{'offset} < \text{max_linear_offset}$
{-4}	$\text{delta}' \geq 0$
{-5}	$\text{rem}(\text{expt}(2, \text{min_page}))(\text{offset}(\text{addr}')) + \text{delta}' < \text{expt}(2, \text{min_page})$
{1}	$\text{offset}(\text{addr}') - \text{rem}(\text{expt}(2, \text{min_page}))(\text{offset}(\text{addr}')) \geq 0$
{2}	$\text{addr}' \text{'offset} + \text{delta}' < \text{max_linear_offset}$

Using lemma rem_floor,

Case splitting on $\text{expt}(2, \text{min_page}) * \text{floor}(\text{offset}(\text{addr}') / \text{expt}(2, \text{min_page})) \geq 0$,

we get 2 subgoals:

delta_memory_address.2.1:

{-1}	$\text{expt}(2, \text{min_page}) \times \text{floor}(\text{offset}(\text{addr}') / \text{expt}(2, \text{min_page})) \geq 0$
{-2}	$\text{offset}(\text{addr}') =$ $\text{rem}(\text{expt}(2, \text{min_page}))(\text{offset}(\text{addr}')) + \text{expt}(2, \text{min_page}) \times \text{floor}(\text{offset}(\text{addr}') / \text{expt}(2, \text{min_page}))$
{-3}	$\text{Mem}?(\text{addr}' \text{'type_of})$
{-4}	$0 \leq \text{addr}' \text{'offset}$
{-5}	$\text{addr}' \text{'offset} < \text{max_linear_offset}$
{-6}	$\text{delta}' \geq 0$
{-7}	$\text{rem}(\text{expt}(2, \text{min_page}))(\text{offset}(\text{addr}')) + \text{delta}' < \text{expt}(2, \text{min_page})$
{1}	$\text{offset}(\text{addr}') - \text{rem}(\text{expt}(2, \text{min_page}))(\text{offset}(\text{addr}')) \geq 0$
{2}	$\text{addr}' \text{'offset} + \text{delta}' < \text{max_linear_offset}$

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of delta_memory_address.2.1.

delta_memory_address.2.2:

{-1}	offset(addr') =
{-2}	Mem?(addr' type_of)
{-3}	0 ≤ addr' offset
{-4}	addr' offset < max_linear_offset
{-5}	delta' ≥ 0
{-6}	rem(expt(2, min_page))(offset(addr')) + delta' < expt(2, min_page)
{1}	expt(2, min_page) × floor(offset(addr')/expt(2, min_page)) ≥ 0
{2}	offset(addr') - rem(expt(2, min_page))(offset(addr')) ≥ 0
{3}	addr' offset + delta' < max_linear_offset

Using lemma nonneg_floor_is_nat,
we get 2 subgoals:

delta_memory_address.2.2.1:

{-1}	floor(offset(addr')/expt(2, min_page)) ≥ 0
{-2}	offset(addr') =
{-3}	Mem?(addr' type_of)
{-4}	0 ≤ addr' offset
{-5}	addr' offset < max_linear_offset
{-6}	delta' ≥ 0
{-7}	rem(expt(2, min_page))(offset(addr')) + delta' < expt(2, min_page)
{1}	expt(2, min_page) × floor(offset(addr')/expt(2, min_page)) ≥ 0
{2}	offset(addr') - rem(expt(2, min_page))(offset(addr')) ≥ 0
{3}	addr' offset + delta' < max_linear_offset

Rewriting using nnreal_times_nnreal_is_nnreal, matching in *,
This completes the proof of delta_memory_address.2.2.1.

delta_memory_address.2.2.2:

{-1}	offset(addr') =
{-2}	Mem?(addr' type_of)
{-3}	0 ≤ addr' offset
{-4}	addr' offset < max_linear_offset
{-5}	delta' ≥ 0
{-6}	rem(expt(2, min_page))(offset(addr')) + delta' < expt(2, min_page)
{1}	offset(addr')/expt(2, min_page) ≥ 0
{2}	expt(2, min_page) × floor(offset(addr')/expt(2, min_page)) ≥ 0
{3}	offset(addr') - rem(expt(2, min_page))(offset(addr')) ≥ 0
{4}	addr' offset + delta' < max_linear_offset

Rewriting using nnreal_div_posreal_is_nnreal, matching in *,
This completes the proof of delta_memory_address.2.2.2.
Q.E.D.

C.117.57

Linear_Memory_Properties.xlat_idx_same_page_address_TCC1

Terse proof for xlat_idx_same_page_address_TCC1.

xlat_idx_same_page_address_TCC1:

{1} \forall (addr: Memory_Address_4G, lvl: Level, a: Memory_Address_4G): offset(a) \geq 0

Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of xlat_idx_same_page_address_TCC1.
 Q.E.D.

C.117.58

Linear_Memory_Properties.xlat_idx_same_page_address_TCC2

Terse proof for xlat_idx_same_page_address_TCC2.

xlat_idx_same_page_address_TCC2:

{1} \forall (addr: Memory_Address_4G, δ : nat, lvl: Level,
 base:
 {a: Memory_Address_4G |
 aligned?(bits_per_level[Physical_memory, pm_phy] + pe_size)
 (offset(a))}):
 rem(expt(2, min_page))(offset(addr)) + δ < expt(2, min_page) \supset
 0 \leq (addr + δ)'offset \wedge (addr + δ)'offset < max_linear_offset

Repeatedly Skolemizing and flattening,
 Rewriting using delta_memory_address, matching in *,
 Expanding the definition of +,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 This completes the proof of xlat_idx_same_page_address_TCC2.
 Q.E.D.

C.117.59 Linear_Memory_Properties.xlat_idx_same_page_address

Terse proof for xlat_idx_same_page_address.

xlat_idx_same_page_address:

{1} \forall (addr: Memory_Address_4G, δ : nat, lvl: Level,
 base:
 {a: Memory_Address_4G | aligned?(bits_per_level + pe_size)(offset(a))}):
 rem(expt(2, min_page))(offset(addr)) + δ < expt(2, min_page) \supset
 xlat_idx(lvl, base, addr) = xlat_idx(lvl, base, addr + δ)

Repeatedly Skolemizing and flattening,
 Expanding the definition of xlat_idx,
 Expanding the definition of cut_bits,
 Installing automatic rewrites from: (bus_width! bits_per_level! max_level! min_page!)
 Case splitting on bus_width - bits_per_level - bits_per_level * lvl!1 \geq 0,
 we get 2 subgoals:

C Proof scripts

`xlat_idx_same_page_address.1:`

{-1}	$\text{bus_width} - \text{bits_per_level} - \text{bits_per_level} \times \text{lvl}' \geq 0$
{-2}	$\text{Mem?}(\text{addr}' \text{ 'type_of})$
{-3}	$0 \leq \text{addr}' \text{ 'offset}$
{-4}	$\text{addr}' \text{ 'offset} < \text{max_linear_offset}$
{-5}	$\text{delta}' \geq 0$
{-6}	$\text{lvl}' < \text{max_level}$
{-7}	$\text{Mem?}(\text{base}' \text{ 'type_of})$
{-8}	$0 \leq \text{base}' \text{ 'offset}$
{-9}	$\text{base}' \text{ 'offset} < \text{max_linear_offset}$
{-10}	$\text{aligned?}(\text{bits_per_level} + \text{pe_size})(\text{offset}(\text{base}'))$
{-11}	$\text{rem}(\text{expt}(2, \text{min_page}))(\text{offset}(\text{addr}')) + \text{delta}' < \text{expt}(2, \text{min_page})$
{1}	$\text{base}' + \text{shift_bits_left}(\text{rem}(\text{expt}(2, \text{bits_per_level}))(\text{ndiv}(\text{offset}(\text{addr}'), \text{expt}(2, -1 \times \text{bits_per_level})),$ $=$ $\text{base}' + \text{shift_bits_left}(\text{rem}(\text{expt}(2, \text{bits_per_level}))(\text{ndiv}(\text{offset}(\text{addr}' + \text{delta}'), \text{expt}(2, -1 \times \text{bits_per_level})),$

Case splitting on $\text{ndiv}(\text{offset}(\text{addr}!1), \text{expt}(2, -1 * (\text{bits_per_level} * \text{lvl}!1) - \text{bits_per_level} + \text{bus_width}))$
 $= \text{ndiv}(\text{offset}(\text{addr}!1 + \text{delta}!1), \text{expt}(2, -1 * (\text{bits_per_level} * \text{lvl}!1) - \text{bits_per_level} + \text{bus_width}))$,

we get 3 subgoals:

`xlat_idx_same_page_address.1.1:`

{-1}	$\text{ndiv}(\text{offset}(\text{addr}'),$ $\text{expt}(2, -1 \times (\text{bits_per_level} \times \text{lvl}') - \text{bits_per_level} + \text{bus_width}))$ $=$ $\text{ndiv}(\text{offset}(\text{addr}' + \text{delta}'),$ $\text{expt}(2, -1 \times (\text{bits_per_level} \times \text{lvl}') - \text{bits_per_level} + \text{bus_width}))$
{-2}	$\text{bus_width} - \text{bits_per_level} - \text{bits_per_level} \times \text{lvl}' \geq 0$
{-3}	$\text{Mem?}(\text{addr}' \text{ 'type_of})$
{-4}	$0 \leq \text{addr}' \text{ 'offset}$
{-5}	$\text{addr}' \text{ 'offset} < \text{max_linear_offset}$
{-6}	$\text{delta}' \geq 0$
{-7}	$\text{lvl}' < \text{max_level}$
{-8}	$\text{Mem?}(\text{base}' \text{ 'type_of})$
{-9}	$0 \leq \text{base}' \text{ 'offset}$
{-10}	$\text{base}' \text{ 'offset} < \text{max_linear_offset}$
{-11}	$\text{aligned?}(\text{bits_per_level} + \text{pe_size})(\text{offset}(\text{base}'))$
{-12}	$\text{rem}(\text{expt}(2, \text{min_page}))(\text{offset}(\text{addr}')) + \text{delta}' < \text{expt}(2, \text{min_page})$
{1}	$\text{base}' + \text{shift_bits_left}(\text{rem}(\text{expt}(2, \text{bits_per_level}))(\text{ndiv}(\text{offset}(\text{addr}'), \text{expt}(2, -1 \times \text{bits_per_level})),$ $=$ $\text{base}' + \text{shift_bits_left}(\text{rem}(\text{expt}(2, \text{bits_per_level}))(\text{ndiv}(\text{offset}(\text{addr}' + \text{delta}'), \text{expt}(2, -1 \times \text{bits_per_level})),$

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `xlat_idx_same_page_address.1.1`.

xlat_idx_same_page_address.1.2:

{-1}	$\text{bus_width} - \text{bits_per_level} - \text{bits_per_level} \times \text{lvl}' \geq 0$
{-2}	$\text{Mem?}(\text{addr}' \text{ 'type_of})$
{-3}	$0 \leq \text{addr}' \text{ 'offset}$
{-4}	$\text{addr}' \text{ 'offset} < \text{max_linear_offset}$
{-5}	$\text{delta}' \geq 0$
{-6}	$\text{lvl}' < \text{max_level}$
{-7}	$\text{Mem?}(\text{base}' \text{ 'type_of})$
{-8}	$0 \leq \text{base}' \text{ 'offset}$
{-9}	$\text{base}' \text{ 'offset} < \text{max_linear_offset}$
{-10}	$\text{aligned?}(\text{bits_per_level} + \text{pe_size})(\text{offset}(\text{base}'))$
{-11}	$\text{rem}(\text{expt}(2, \text{min_page}))(\text{offset}(\text{addr}') + \text{delta}') < \text{expt}(2, \text{min_page})$
{1}	$\begin{aligned} & \text{ndiv}(\text{offset}(\text{addr}'), \\ & \quad \text{expt}(2, -1 \times (\text{bits_per_level} \times \text{lvl}') - \text{bits_per_level} + \text{bus_width})) \\ & = \\ & \text{ndiv}(\text{offset}(\text{addr}' + \text{delta}'), \\ & \quad \text{expt}(2, -1 \times (\text{bits_per_level} \times \text{lvl}') - \text{bits_per_level} + \text{bus_width})) \end{aligned}$
{2}	$\begin{aligned} & \text{base}' + \text{shift_bits_left}(\text{rem}(\text{expt}(2, \text{bits_per_level}))(\text{ndiv}(\text{offset}(\text{addr}'), \text{expt}(2, -1 \times \text{bits_per_level} \times \text{lvl}' - \text{bits_per_level} + \text{bus_width}))), \\ & = \\ & \text{base}' + \text{shift_bits_left}(\text{rem}(\text{expt}(2, \text{bits_per_level}))(\text{ndiv}(\text{offset}(\text{addr}' + \text{delta}'), \text{expt}(2, -1 \times \text{bits_per_level} \times \text{lvl}' - \text{bits_per_level} + \text{bus_width}))), \end{aligned}$

Hiding formulas: 2,

Expanding the definition of +,

Using lemma ndiv_plus_mod,

we get 2 subgoals:

xlat_idx_same_page_address.1.2.1:

<pre> {-1} divides(expt(2, -1 × (bits_per_level × lvl') - bits_per_level + bus_width), offset(addr') - rem(expt(2, -1 × (bits_per_level × lvl') - bits_per_level + bus_width))(offset(addr') - rem(expt(2, -1 × (bits_per_level × lvl') - bits_per_level + bus_width))))(offset(addr') - rem(expt(2, -1 × (bits_per_level × lvl') - bits_per_level + bus_width))) ⊃ ndiv(offset(addr') - rem(expt(2, -1 × (bits_per_level × lvl') - bits_per_level + bus_width))(offset(addr') - rem(expt(2, -1 × (bits_per_level × lvl') - bits_per_level + bus_width))))(offset(addr') - rem(expt(2, -1 × (bits_per_level × lvl') - bits_per_level + bus_width))) = ndiv(offset(addr') - rem(expt(2, -1 × (bits_per_level × lvl') - bits_per_level + bus_width))(offset(addr') - rem(expt(2, -1 × (bits_per_level × lvl') - bits_per_level + bus_width))))(offset(addr') - rem(expt(2, -1 × (bits_per_level × lvl') - bits_per_level + bus_width))) {-2} bus_width - bits_per_level - bits_per_level × lvl' ≥ 0 {-3} Mem?(addr' 'type_of) {-4} 0 ≤ addr' 'offset {-5} addr' 'offset < max_linear_offset {-6} delta' ≥ 0 {-7} lvl' < max_level {-8} Mem?(base' 'type_of) {-9} 0 ≤ base' 'offset {-10} base' 'offset < max_linear_offset {-11} aligned?(bits_per_level + pe_size)(offset(base')) {-12} rem(expt(2, min_page))(offset(addr')) + delta' < expt(2, min_page) </pre>	<pre> {1} ndiv(offset(addr'), expt(2, -1 × (bits_per_level × lvl') - bits_per_level + bus_width)) = ndiv(addr' 'offset + delta', expt(2, -1 × (bits_per_level × lvl') - bits_per_level + bus_width)) </pre>
---	---

Using lemma `ndiv_plus_mod`,

we get 3 subgoals:

xlat_idx_same_page_address.1.2.1.1:

{-1}	$\text{divides}(\text{expt}(2, -1 \times (\text{bits_per_level} \times \text{lvl}') - \text{bits_per_level} + \text{bus_width}),$ $\text{offset}(\text{addr}') - \text{rem}(\text{expt}(2, -1 \times (\text{bits_per_level} \times \text{lvl}') - \text{bits_per_level} + \text{bus_width}))(\text{offset}(\text{addr}'))$ \supset $\text{ndiv}(\text{offset}(\text{addr}') - \text{rem}(\text{expt}(2, -1 \times (\text{bits_per_level} \times \text{lvl}') - \text{bits_per_level} + \text{bus_width}))(\text{offset}(\text{addr}')) +$ $\text{expt}(2,$ $-1 \times (\text{bits_per_level} \times \text{lvl}') - \text{bits_per_level} + \text{bus_width}))$ $=$ $\text{ndiv}(\text{offset}(\text{addr}') - \text{rem}(\text{expt}(2, -1 \times (\text{bits_per_level} \times \text{lvl}') - \text{bits_per_level} + \text{bus_width}))(\text{offset}(\text{addr}')),$ $\text{expt}(2,$ $-1 \times (\text{bits_per_level} \times \text{lvl}') - \text{bits_per_level} + \text{bus_width}))$
{-2}	$\text{divides}(\text{expt}(2, -1 \times (\text{bits_per_level} \times \text{lvl}') - \text{bits_per_level} + \text{bus_width}),$ $\text{offset}(\text{addr}') - \text{rem}(\text{expt}(2, -1 \times (\text{bits_per_level} \times \text{lvl}') - \text{bits_per_level} + \text{bus_width}))(\text{offset}(\text{addr}'))$ \supset $\text{ndiv}(\text{offset}(\text{addr}') - \text{rem}(\text{expt}(2, -1 \times (\text{bits_per_level} \times \text{lvl}') - \text{bits_per_level} + \text{bus_width}))(\text{offset}(\text{addr}')) +$ $\text{expt}(2,$ $-1 \times (\text{bits_per_level} \times \text{lvl}') - \text{bits_per_level} + \text{bus_width}))$ $=$ $\text{ndiv}(\text{offset}(\text{addr}') - \text{rem}(\text{expt}(2, -1 \times (\text{bits_per_level} \times \text{lvl}') - \text{bits_per_level} + \text{bus_width}))(\text{offset}(\text{addr}')),$ $\text{expt}(2,$ $-1 \times (\text{bits_per_level} \times \text{lvl}') - \text{bits_per_level} + \text{bus_width}))$
{-3}	$\text{bus_width} - \text{bits_per_level} - \text{bits_per_level} \times \text{lvl}' \geq 0$
{-4}	$\text{Mem}?(\text{addr}' \text{'type_of})$
{-5}	$0 \leq \text{addr}' \text{'offset}$
{-6}	$\text{addr}' \text{'offset} < \text{max_linear_offset}$
{-7}	$\text{delta}' \geq 0$
{-8}	$\text{lvl}' < \text{max_level}$
{-9}	$\text{Mem}?(\text{base}' \text{'type_of})$
{-10}	$0 \leq \text{base}' \text{'offset}$
{-11}	$\text{base}' \text{'offset} < \text{max_linear_offset}$
{-12}	$\text{aligned}?(\text{bits_per_level} + \text{pe_size})(\text{offset}(\text{base}'))$
{-13}	$\text{rem}(\text{expt}(2, \text{min_page}))(\text{offset}(\text{addr}')) + \text{delta}' < \text{expt}(2, \text{min_page})$
{1}	$\text{ndiv}(\text{offset}(\text{addr}'),$ $\text{expt}(2, -1 \times (\text{bits_per_level} \times \text{lvl}') - \text{bits_per_level} + \text{bus_width}))$ $=$ $\text{ndiv}(\text{addr}' \text{'offset} + \text{delta}',$ $\text{expt}(2, -1 \times (\text{bits_per_level} \times \text{lvl}') - \text{bits_per_level} + \text{bus_width}))$

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Using lemma `rem_def2`,

we get 2 subgoals:

C Proof scripts

xlat_idx_same_page_address.1.2.1.1.1:

{-1}	TRUE \equiv divides(expt(2, 22 - 10 × lvl'), offset(addr') - rem(expt(2, 22 - 10 × lvl'))(offset(addr')))
{-2}	22 - 10 × lvl' ≥ 0
{-3}	Mem?(addr' type_of)
{-4}	0 ≤ addr' offset
{-5}	addr' offset < max_linear_offset
{-6}	delta' ≥ 0
{-7}	lvl' < 2
{-8}	Mem?(base' type_of)
{-9}	0 ≤ base' offset
{-10}	base' offset < max_linear_offset
{-11}	aligned?(10 + pe_size)(offset(base'))
{-12}	rem(expt(2, 12))(offset(addr')) + delta' < expt(2, 12)
{1}	divides(expt(2, 22 - 10 × lvl'), offset(addr') - rem(expt(2, 22 - 10 × lvl'))(offset(addr')))
{2}	ndiv(offset(addr'), expt(2, 22 - 10 × lvl')) = ndiv(addr' offset + delta', expt(2, 22 - 10 × lvl'))

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of xlat_idx_same_page_address.1.2.1.1.1.

xlat_idx_same_page_address.1.2.1.1.2:

{-1}	22 - 10 × lvl' ≥ 0
{-2}	Mem?(addr' type_of)
{-3}	0 ≤ addr' offset
{-4}	addr' offset < max_linear_offset
{-5}	delta' ≥ 0
{-6}	lvl' < 2
{-7}	Mem?(base' type_of)
{-8}	0 ≤ base' offset
{-9}	base' offset < max_linear_offset
{-10}	aligned?(10 + pe_size)(offset(base'))
{-11}	rem(expt(2, 12))(offset(addr')) + delta' < expt(2, 12)
{1}	rem(expt(2, 22 - 10 × lvl'))(offset(addr')) < expt(2, 22 - 10 × lvl')
{2}	divides(expt(2, 22 - 10 × lvl'), offset(addr') - rem(expt(2, 22 - 10 × lvl'))(offset(addr')))
{3}	ndiv(offset(addr'), expt(2, 22 - 10 × lvl')) = ndiv(addr' offset + delta', expt(2, 22 - 10 × lvl'))

Adding type constraints for rem(expt(2, bus_width - bits_per_level - bits_per_level * lvl!1)) (offset(addr!1)),

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of xlat_idx_same_page_address.1.2.1.1.2.

xlat_idx_same_page_address.1.2.1.2:

{-1}	$\text{divides}(\text{expt}(2, -1 \times (\text{bits_per_level} \times \text{lvl}') - \text{bits_per_level} + \text{bus_width}),$ $\text{offset}(\text{addr}') - \text{rem}(\text{expt}(2, -1 \times (\text{bits_per_level} \times \text{lvl}') - \text{bits_per_level} + \text{bus_width}))(\text{offset}(\text{addr}')) + \text{min_page})$ \supset $\text{ndiv}(\text{offset}(\text{addr}') - \text{rem}(\text{expt}(2, -1 \times (\text{bits_per_level} \times \text{lvl}') - \text{bits_per_level} + \text{bus_width}))(\text{offset}(\text{addr}')) + \text{min_page},$ $\text{expt}(2, -1 \times (\text{bits_per_level} \times \text{lvl}') - \text{bits_per_level} + \text{bus_width}))$ $=$ $\text{ndiv}(\text{offset}(\text{addr}') - \text{rem}(\text{expt}(2, -1 \times (\text{bits_per_level} \times \text{lvl}') - \text{bits_per_level} + \text{bus_width}))(\text{offset}(\text{addr}'))),$ $\text{expt}(2, -1 \times (\text{bits_per_level} \times \text{lvl}') - \text{bits_per_level} + \text{bus_width}))$
{-2}	$\text{bus_width} - \text{bits_per_level} - \text{bits_per_level} \times \text{lvl}' \geq 0$
{-3}	$\text{Mem?}(\text{addr}' \text{ type_of})$
{-4}	$0 \leq \text{addr}' \text{ offset}$
{-5}	$\text{addr}' \text{ offset} < \text{max_linear_offset}$
{-6}	$\text{delta}' \geq 0$
{-7}	$\text{lvl}' < \text{max_level}$
{-8}	$\text{Mem?}(\text{base}' \text{ type_of})$
{-9}	$0 \leq \text{base}' \text{ offset}$
{-10}	$\text{base}' \text{ offset} < \text{max_linear_offset}$
{-11}	$\text{aligned?}(\text{bits_per_level} + \text{pe_size})(\text{offset}(\text{base}'))$
{-12}	$\text{rem}(\text{expt}(2, \text{min_page}))(\text{offset}(\text{addr}')) + \text{delta}' < \text{expt}(2, \text{min_page})$
{1}	$\text{rem}(\text{expt}(2, 22 - 10 \times \text{lvl}'))(\text{offset}(\text{addr}')) + \text{delta}' <$ $\text{expt}(2, 22 - 10 \times \text{lvl}')$
{2}	$\text{ndiv}(\text{offset}(\text{addr}'),$ $\text{expt}(2, -1 \times (\text{bits_per_level} \times \text{lvl}') - \text{bits_per_level} + \text{bus_width}))$ $=$ $\text{ndiv}(\text{addr}' \text{ offset} + \text{delta}',$ $\text{expt}(2, -1 \times (\text{bits_per_level} \times \text{lvl}') - \text{bits_per_level} + \text{bus_width}))$

Keeping (-2 -6 -7 -11 -12 1) and hiding *,

Using lemma aligned_add_below,

we get 3 subgoals:

xlat_idx_same_page_address.1.2.1.2.1:

{-1}	$\text{min_page} \leq \text{bus_width} - \text{bits_per_level} - \text{bits_per_level} \times \text{lvl}' \wedge$ $\text{rem}(\text{expt}(2, \text{bus_width} - \text{bits_per_level} - \text{bits_per_level} \times \text{lvl}'))(\text{offset}(\text{addr}')) - \text{rem}(\text{expt}(2, \text{min_page}))(\text{offset}(\text{addr}'))$ $< \text{expt}(2, \text{bus_width} - \text{bits_per_level} - \text{bits_per_level} \times \text{lvl}')$ \wedge $\text{aligned?}(\text{min_page})$ $(\text{rem}(\text{expt}(2, \text{bus_width} - \text{bits_per_level} - \text{bits_per_level} \times \text{lvl}'))(\text{offset}(\text{addr}')) - \text{rem}(\text{expt}(2, \text{min_page}))(\text{offset}(\text{addr}')))$ $\wedge \text{rem}(\text{expt}(2, \text{min_page}))(\text{offset}(\text{addr}')) + \text{delta}' < \text{expt}(2, \text{min_page})$ \supset $\text{rem}(\text{expt}(2, \text{bus_width} - \text{bits_per_level} - \text{bits_per_level} \times \text{lvl}'))(\text{offset}(\text{addr}')) - \text{rem}(\text{expt}(2, \text{min_page}))(\text{offset}(\text{addr}'))$ $< \text{expt}(2, \text{bus_width} - \text{bits_per_level} - \text{bits_per_level} \times \text{lvl}')$
{-2}	$\text{bus_width} - \text{bits_per_level} - \text{bits_per_level} \times \text{lvl}' \geq 0$
{-3}	$\text{delta}' \geq 0$
{-4}	$\text{lvl}' < \text{max_level}$
{-5}	$\text{aligned?}(\text{bits_per_level} + \text{pe_size})(\text{offset}(\text{base}'))$
{-6}	$\text{rem}(\text{expt}(2, \text{min_page}))(\text{offset}(\text{addr}')) + \text{delta}' < \text{expt}(2, \text{min_page})$
{1}	$\text{rem}(\text{expt}(2, 22 - 10 \times \text{lvl}'))(\text{offset}(\text{addr}')) + \text{delta}' <$ $\text{expt}(2, 22 - 10 \times \text{lvl}')$

C Proof scripts

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
we get 2 subgoals:

`xlat_idx_same_page_address.1.2.1.2.1.1:`

{-1}	$\text{rem}(\text{expt}(2, 12))(\text{offset}(\text{addr}')) + \text{delta}' < \text{expt}(2, 12)$
{-2}	$22 - 10 \times \text{lvl}' \geq 0$
{-3}	$\text{delta}' \geq 0$
{-4}	$\text{lvl}' < 2$
{-5}	$\text{aligned?}(10 + \text{pe_size})(\text{offset}(\text{base}'))$
{1}	$\text{aligned?}(12)$ $\text{rem}(\text{expt}(2, 22 - 10 \times \text{lvl}'))(\text{offset}(\text{addr}')) - \text{rem}(\text{expt}(2, 12))(\text{offset}(\text{addr}'))$
{2}	$\text{rem}(\text{expt}(2, 22 - 10 \times \text{lvl}'))(\text{offset}(\text{addr}')) + \text{delta}' < \text{expt}(2, 22 - 10 \times \text{lvl}')$

Expanding the definition of `aligned?`,

Using lemma `rem_def2`,

Rewriting using `rem_rem`, matching in `*`,

we get 2 subgoals:

`xlat_idx_same_page_address.1.2.1.2.1.1.1:`

{-1}	$\text{TRUE} \equiv \text{divides}(\text{expt}(2, \text{min_page}), \text{rem}(\text{expt}(2, \text{bus_width} - \text{bits_per_level} - \text{bits_per_level} \times \text{lvl}'))(\text{offset}(\text{addr}')) - \text{rem}(\text{expt}(2, 12))(\text{offset}(\text{addr}')))$
{-2}	$\text{rem}(\text{expt}(2, 12))(\text{offset}(\text{addr}')) + \text{delta}' < \text{expt}(2, 12)$
{-3}	$22 - 10 \times \text{lvl}' \geq 0$
{-4}	$\text{delta}' \geq 0$
{-5}	$\text{lvl}' < 2$
{-6}	$\text{divides}(\text{expt}(2, 10 + \text{pe_size}), \text{offset}(\text{base}'))$
{1}	$\text{divides}(\text{expt}(2, 12), \text{rem}(\text{expt}(2, 22 - 10 \times \text{lvl}'))(\text{offset}(\text{addr}')) - \text{rem}(\text{expt}(2, 12))(\text{offset}(\text{addr}')))$
{2}	$\text{rem}(\text{expt}(2, 22 - 10 \times \text{lvl}'))(\text{offset}(\text{addr}')) + \text{delta}' < \text{expt}(2, 22 - 10 \times \text{lvl}')$

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `xlat_idx_same_page_address.1.2.1.2.1.1.1`.

`xlat_idx_same_page_address.1.2.1.2.1.1.2:`

{-1}	$\text{rem}(\text{expt}(2, 12))(\text{offset}(\text{addr}')) + \text{delta}' < \text{expt}(2, 12)$
{-2}	$22 - 10 \times \text{lvl}' \geq 0$
{-3}	$\text{delta}' \geq 0$
{-4}	$\text{lvl}' < 2$
{-5}	$\text{divides}(\text{expt}(2, 10 + \text{pe_size}), \text{offset}(\text{base}'))$
{1}	$\text{divides}(\text{expt}(2, 12), \text{expt}(2, 22 - 10 \times \text{lvl}'))$
{2}	$\text{rem}(\text{expt}(2, 12))(\text{rem}(\text{expt}(2, 22 - 10 \times \text{lvl}'))(\text{offset}(\text{addr}')) - \text{rem}(\text{expt}(2, 12))(\text{offset}(\text{addr}')))$
{3}	$\text{divides}(\text{expt}(2, 12), \text{rem}(\text{expt}(2, 22 - 10 \times \text{lvl}'))(\text{offset}(\text{addr}')) - \text{rem}(\text{expt}(2, 12))(\text{offset}(\text{addr}')))$
{4}	$\text{rem}(\text{expt}(2, 22 - 10 \times \text{lvl}'))(\text{offset}(\text{addr}')) + \text{delta}' < \text{expt}(2, 22 - 10 \times \text{lvl}')$

Keeping (-2 1) and hiding `*`,

Using lemma `divides_expt_gt`,

Rewriting using `divides_reflexive`, matching in `*`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `xlat_idx_same_page_address.1.2.1.2.1.1.2`.

xlat_idx_same_page_address.1.2.1.2.1.2:

<pre> {-1} rem(expt(2, 12))(offset(addr')) + delta' < expt(2, 12) {-2} 22 - 10 × lvl' ≥ 0 {-3} delta' ≥ 0 {-4} lvl' < 2 {-5} aligned?(10 + pe_size)(offset(base')) </pre>	<hr style="border: 0.5px solid black;"/> <pre> {1} rem(expt(2, 22 - 10 × lvl'))(offset(addr')) - rem(expt(2, 12))(offset(addr')) < expt(2, 22 - 10 × lvl') {2} rem(expt(2, 22 - 10 × lvl'))(offset(addr')) + delta' < expt(2, 22 - 10 × lvl') </pre>
--	--

Adding type constraints for $\text{rem}(\text{expt}(2, \text{bus_width} - \text{bits_per_level} - \text{bits_per_level} * \text{lvl}1))$ ($\text{offset}(\text{addr}1)$),

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of xlat_idx_same_page_address.1.2.1.2.1.2.

xlat_idx_same_page_address.1.2.1.2.2:

<pre> {-1} bus_width - bits_per_level - bits_per_level × lvl' ≥ 0 {-2} delta' ≥ 0 {-3} lvl' < max_level {-4} aligned?(bits_per_level + pe_size)(offset(base')) {-5} rem(expt(2, min_page))(offset(addr')) + delta' < expt(2, min_page) </pre>	<hr style="border: 0.5px solid black;"/> <pre> {1} rem(expt(2, 22 - 10 × lvl'))(offset(addr')) - rem(expt(2, 12))(offset(addr')) ≥ 0 {2} rem(expt(2, 22 - 10 × lvl'))(offset(addr')) + delta' < expt(2, 22 - 10 × lvl') </pre>
--	---

Using lemma `rem_floor`,

we get 2 subgoals:

xlat_idx_same_page_address.1.2.1.2.2.1:

<pre> {-1} rem(expt(2, bus_width - bits_per_level - bits_per_level × lvl')) (offset(addr')) = rem(expt(2, 12))(rem(expt(2, bus_width - bits_per_level - bits_per_level × lvl'))(offset(addr'))) + expt(2, 12) {-2} bus_width - bits_per_level - bits_per_level × lvl' ≥ 0 {-3} delta' ≥ 0 {-4} lvl' < max_level {-5} aligned?(bits_per_level + pe_size)(offset(base')) {-6} rem(expt(2, min_page))(offset(addr')) + delta' < expt(2, min_page) </pre>	<hr style="border: 0.5px solid black;"/> <pre> {1} rem(expt(2, 22 - 10 × lvl'))(offset(addr')) - rem(expt(2, 12))(offset(addr')) ≥ 0 {2} rem(expt(2, 22 - 10 × lvl'))(offset(addr')) + delta' < expt(2, 22 - 10 × lvl') </pre>
---	---

Rewriting using `rem_rem`, matching in `*`,

we get 2 subgoals:

C Proof scripts

xlat_idx_same_page_address.1.2.1.2.2.1.1.1:

{-1}	$\text{rem}(\text{expt}(2, \text{bus_width} - \text{bits_per_level} - \text{bits_per_level} \times \text{lvl}'))$
	$(\text{offset}(\text{addr}'))$
	=
	$\text{rem}(\text{expt}(2, 12))(\text{offset}(\text{addr}')) + \text{expt}(2, 12) \times \text{floor}(\text{rem}(\text{expt}(2, \text{bus_width} - \text{bits_per_level} - \text{bits_per_level} \times \text{lvl}'))$
{-2}	$\text{bus_width} - \text{bits_per_level} - \text{bits_per_level} \times \text{lvl}' \geq 0$
{-3}	$\text{delta}' \geq 0$
{-4}	$\text{lvl}' < \text{max_level}$
{-5}	$\text{aligned}?(\text{bits_per_level} + \text{pe_size})(\text{offset}(\text{base}'))$
{-6}	$\text{rem}(\text{expt}(2, \text{min_page}))(\text{offset}(\text{addr}')) + \text{delta}' < \text{expt}(2, \text{min_page})$
{1}	$\text{rem}(\text{expt}(2, 22 - 10 \times \text{lvl}'))(\text{offset}(\text{addr}')) - \text{rem}(\text{expt}(2, 12))(\text{offset}(\text{addr}'))$
	≥ 0
{2}	$\text{rem}(\text{expt}(2, 22 - 10 \times \text{lvl}'))(\text{offset}(\text{addr}')) + \text{delta}' <$
	$\text{expt}(2, 22 - 10 \times \text{lvl}')$

Adding type constraints for $\text{expt}(2, 12) * \text{floor}(\text{rem}(\text{expt}(2, \text{bus_width} - \text{bits_per_level} - \text{bits_per_level} * \text{lvl}')) (\text{offset}(\text{addr}!1)) / \text{expt}(2, \text{min_page}))$,

we get 2 subgoals:

xlat_idx_same_page_address.1.2.1.2.2.1.1.1.1:

{-1}	$\text{expt}(2, 12) \times \text{floor}(\text{rem}(\text{expt}(2, \text{bus_width} - \text{bits_per_level} - \text{bits_per_level} \times \text{lvl}'))(\text{offset}(\text{addr}')))/\text{expt}(2, \text{min_page})$
	≥ 0
{-2}	$\text{divides}(\text{expt}(2, 12))$
	$(\text{expt}(2, 12) \times \text{floor}(\text{rem}(\text{expt}(2, \text{bus_width} - \text{bits_per_level} - \text{bits_per_level} \times \text{lvl}'))(\text{offset}(\text{addr}')))/\text{expt}(2, \text{min_page}))$
{-3}	$\text{divides}(\text{floor}(\text{rem}(\text{expt}(2, -1 \times \text{bits_per_level} \times \text{lvl}' - \text{bits_per_level} + \text{bus_width}))(\text{offset}(\text{addr}')))/\text{expt}(2, \text{min_page}))$
	$(\text{expt}(2, 12) \times \text{floor}(\text{rem}(\text{expt}(2, \text{bus_width} - \text{bits_per_level} - \text{bits_per_level} \times \text{lvl}'))(\text{offset}(\text{addr}')))/\text{expt}(2, \text{min_page}))$
{-4}	$\text{rem}(\text{expt}(2, \text{bus_width} - \text{bits_per_level} - \text{bits_per_level} \times \text{lvl}'))$
	$(\text{offset}(\text{addr}'))$
	=
	$\text{rem}(\text{expt}(2, 12))(\text{offset}(\text{addr}')) + \text{expt}(2, 12) \times \text{floor}(\text{rem}(\text{expt}(2, \text{bus_width} - \text{bits_per_level} - \text{bits_per_level} \times \text{lvl}'))$
{-5}	$\text{bus_width} - \text{bits_per_level} - \text{bits_per_level} \times \text{lvl}' \geq 0$
{-6}	$\text{delta}' \geq 0$
{-7}	$\text{lvl}' < \text{max_level}$
{-8}	$\text{aligned}?(\text{bits_per_level} + \text{pe_size})(\text{offset}(\text{base}'))$
{-9}	$\text{rem}(\text{expt}(2, \text{min_page}))(\text{offset}(\text{addr}')) + \text{delta}' < \text{expt}(2, \text{min_page})$
{1}	$\text{rem}(\text{expt}(2, 22 - 10 \times \text{lvl}'))(\text{offset}(\text{addr}')) - \text{rem}(\text{expt}(2, 12))(\text{offset}(\text{addr}'))$
	≥ 0
{2}	$\text{rem}(\text{expt}(2, 22 - 10 \times \text{lvl}'))(\text{offset}(\text{addr}')) + \text{delta}' <$
	$\text{expt}(2, 22 - 10 \times \text{lvl}')$

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of xlat_idx_same_page_address.1.2.1.2.2.1.1.1.

xlat_idx_same_page_address.1.2.1.2.2.1.1.2:

{-1}	$\text{rem}(\text{expt}(2, \text{bus_width} - \text{bits_per_level} - \text{bits_per_level} \times \text{lvl}'))$ $(\text{offset}(\text{addr}'))$
	=
{-2}	$\text{rem}(\text{expt}(2, 12))(\text{offset}(\text{addr}')) + \text{expt}(2, 12) \times \text{floor}(\text{rem}(\text{expt}(2, \text{bus_width} - \text{bits_per_level} - \text{bits_per_level} \times \text{lvl}')) / \text{expt}(2, 12))$
{-3}	$\text{delta}' \geq 0$
{-4}	$\text{lvl}' < \text{max_level}$
{-5}	$\text{aligned}?(\text{bits_per_level} + \text{pe_size})(\text{offset}(\text{base}'))$
{-6}	$\text{rem}(\text{expt}(2, \text{min_page}))(\text{offset}(\text{addr}')) + \text{delta}' < \text{expt}(2, \text{min_page})$
{1}	$22 - 10 \times \text{lvl}' \geq 0$
{2}	$\text{rem}(\text{expt}(2, 22 - 10 \times \text{lvl}'))(\text{offset}(\text{addr}')) - \text{rem}(\text{expt}(2, 12))(\text{offset}(\text{addr}'))$ ≥ 0
{3}	$\text{rem}(\text{expt}(2, 22 - 10 \times \text{lvl}'))(\text{offset}(\text{addr}')) + \text{delta}' <$ $\text{expt}(2, 22 - 10 \times \text{lvl}')$

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of xlat_idx_same_page_address.1.2.1.2.2.1.1.2.

xlat_idx_same_page_address.1.2.1.2.2.1.2:

{-1}	$\text{rem}(\text{expt}(2, 22 - 10 \times \text{lvl}'))(\text{offset}(\text{addr}')) =$ $\text{rem}(\text{expt}(2, 12))(\text{rem}(\text{expt}(2, 22 - 10 \times \text{lvl}'))(\text{offset}(\text{addr}')))) + \text{expt}(2, 12) \times \text{floor}(\text{rem}(\text{expt}(2, 22 - 10 \times \text{lvl}')) / \text{expt}(2, 12))$
{-2}	$22 - 10 \times \text{lvl}' \geq 0$
{-3}	$\text{delta}' \geq 0$
{-4}	$\text{lvl}' < 2$
{-5}	$\text{aligned}?(10 + \text{pe_size})(\text{offset}(\text{base}'))$
{-6}	$\text{rem}(\text{expt}(2, 12))(\text{offset}(\text{addr}')) + \text{delta}' < \text{expt}(2, 12)$
{1}	$\text{divides}(\text{expt}(2, 12), \text{expt}(2, 22 - 10 \times \text{lvl}'))$
{2}	$\text{rem}(\text{expt}(2, 22 - 10 \times \text{lvl}'))(\text{offset}(\text{addr}')) - \text{rem}(\text{expt}(2, 12))(\text{offset}(\text{addr}'))$ ≥ 0
{3}	$\text{rem}(\text{expt}(2, 22 - 10 \times \text{lvl}'))(\text{offset}(\text{addr}')) + \text{delta}' <$ $\text{expt}(2, 22 - 10 \times \text{lvl}')$

Keeping (-2 1) and hiding *,
Using lemma divides_expt_gt,
Rewriting using divides_reflexive, matching in *,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of xlat_idx_same_page_address.1.2.1.2.2.1.2.

xlat_idx_same_page_address.1.2.1.2.2.2:

{-1}	$\text{bus_width} - \text{bits_per_level} - \text{bits_per_level} \times \text{lvl}' \geq 0$
{-2}	$\text{delta}' \geq 0$
{-3}	$\text{lvl}' < \text{max_level}$
{-4}	$\text{aligned}?(\text{bits_per_level} + \text{pe_size})(\text{offset}(\text{base}'))$
{-5}	$\text{rem}(\text{expt}(2, \text{min_page}))(\text{offset}(\text{addr}')) + \text{delta}' < \text{expt}(2, \text{min_page})$
{1}	$22 - 10 \times \text{lvl}' \geq 0$
{2}	$\text{rem}(\text{expt}(2, 22 - 10 \times \text{lvl}'))(\text{offset}(\text{addr}')) - \text{rem}(\text{expt}(2, 12))(\text{offset}(\text{addr}'))$ ≥ 0
{3}	$\text{rem}(\text{expt}(2, 22 - 10 \times \text{lvl}'))(\text{offset}(\text{addr}')) + \text{delta}' <$ $\text{expt}(2, 22 - 10 \times \text{lvl}')$

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of xlat_idx_same_page_address.1.2.1.2.2.2.

C Proof scripts

`xlat_idx_same_page_address.1.2.1.2.3:`

{-1}	$\text{bus_width} - \text{bits_per_level} - \text{bits_per_level} \times \text{lvl}' \geq 0$
{-2}	$\text{delta}' \geq 0$
{-3}	$\text{lvl}' < \text{max_level}$
{-4}	$\text{aligned}?(\text{bits_per_level} + \text{pe_size})(\text{offset}(\text{base}'))$
{-5}	$\text{rem}(\text{expt}(2, \text{min_page}))(\text{offset}(\text{addr}')) + \text{delta}' < \text{expt}(2, \text{min_page})$
{1}	$22 - 10 \times \text{lvl}' \geq 0$
{2}	$\text{rem}(\text{expt}(2, 22 - 10 \times \text{lvl}'))(\text{offset}(\text{addr}')) + \text{delta}' < \text{expt}(2, 22 - 10 \times \text{lvl}')$

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `xlat_idx_same_page_address.1.2.1.2.3`.

`xlat_idx_same_page_address.1.2.1.3:`

{-1}	$\text{divides}(\text{expt}(2, -1 \times (\text{bits_per_level} \times \text{lvl}') - \text{bits_per_level} + \text{bus_width}), \text{offset}(\text{addr}') - \text{rem}(\text{expt}(2, -1 \times (\text{bits_per_level} \times \text{lvl}') - \text{bits_per_level} + \text{bus_width})))$ \supset $\text{ndiv}(\text{offset}(\text{addr}') - \text{rem}(\text{expt}(2, -1 \times (\text{bits_per_level} \times \text{lvl}') - \text{bits_per_level} + \text{bus_width})))$ $\text{expt}(2, -1 \times (\text{bits_per_level} \times \text{lvl}') - \text{bits_per_level} + \text{bus_width}))$ $=$ $\text{ndiv}(\text{offset}(\text{addr}') - \text{rem}(\text{expt}(2, -1 \times (\text{bits_per_level} \times \text{lvl}') - \text{bits_per_level} + \text{bus_width})))$ $\text{expt}(2, -1 \times (\text{bits_per_level} \times \text{lvl}') - \text{bits_per_level} + \text{bus_width}))$
{-2}	$\text{bus_width} - \text{bits_per_level} - \text{bits_per_level} \times \text{lvl}' \geq 0$
{-3}	$\text{Mem}?(\text{addr}' \text{'type_of})$
{-4}	$0 \leq \text{addr}' \text{'offset}$
{-5}	$\text{addr}' \text{'offset} < \text{max_linear_offset}$
{-6}	$\text{delta}' \geq 0$
{-7}	$\text{lvl}' < \text{max_level}$
{-8}	$\text{Mem}?(\text{base}' \text{'type_of})$
{-9}	$0 \leq \text{base}' \text{'offset}$
{-10}	$\text{base}' \text{'offset} < \text{max_linear_offset}$
{-11}	$\text{aligned}?(\text{bits_per_level} + \text{pe_size})(\text{offset}(\text{base}'))$
{-12}	$\text{rem}(\text{expt}(2, \text{min_page}))(\text{offset}(\text{addr}')) + \text{delta}' < \text{expt}(2, \text{min_page})$
{1}	$22 - 10 \times \text{lvl}' \geq 0$
{2}	$\text{ndiv}(\text{offset}(\text{addr}'), \text{expt}(2, -1 \times (\text{bits_per_level} \times \text{lvl}') - \text{bits_per_level} + \text{bus_width}))$ $=$ $\text{ndiv}(\text{addr}' \text{'offset} + \text{delta}', \text{expt}(2, -1 \times (\text{bits_per_level} \times \text{lvl}') - \text{bits_per_level} + \text{bus_width}))$

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `xlat_idx_same_page_address.1.2.1.3`.

xlat_idx_same_page_address.1.2.2:

{-1}	$\text{bus_width} - \text{bits_per_level} - \text{bits_per_level} \times \text{lvl}' \geq 0$
{-2}	$\text{Mem?}(\text{addr}'\text{'type_of})$
{-3}	$0 \leq \text{addr}'\text{'offset}$
{-4}	$\text{addr}'\text{'offset} < \text{max_linear_offset}$
{-5}	$\text{delta}' \geq 0$
{-6}	$\text{lvl}' < \text{max_level}$
{-7}	$\text{Mem?}(\text{base}'\text{'type_of})$
{-8}	$0 \leq \text{base}'\text{'offset}$
{-9}	$\text{base}'\text{'offset} < \text{max_linear_offset}$
{-10}	$\text{aligned?}(\text{bits_per_level} + \text{pe_size})(\text{offset}(\text{base}'))$
{-11}	$\text{rem}(\text{expt}(2, \text{min_page}))(\text{offset}(\text{addr}')) + \text{delta}' < \text{expt}(2, \text{min_page})$
{1}	$22 - 10 \times \text{lvl}' \geq 0$
{2}	$\text{ndiv}(\text{offset}(\text{addr}'),$ $\text{expt}(2, -1 \times (\text{bits_per_level} \times \text{lvl}') - \text{bits_per_level} + \text{bus_width}))$ $=$ $\text{ndiv}(\text{addr}'\text{'offset} + \text{delta}',$ $\text{expt}(2, -1 \times (\text{bits_per_level} \times \text{lvl}') - \text{bits_per_level} + \text{bus_width}))$

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of xlat_idx_same_page_address.1.2.2.

xlat_idx_same_page_address.1.3:

{-1}	$\text{bus_width} - \text{bits_per_level} - \text{bits_per_level} \times \text{lvl}' \geq 0$
{-2}	$\text{Mem?}(\text{addr}'\text{'type_of})$
{-3}	$0 \leq \text{addr}'\text{'offset}$
{-4}	$\text{addr}'\text{'offset} < \text{max_linear_offset}$
{-5}	$\text{delta}' \geq 0$
{-6}	$\text{lvl}' < \text{max_level}$
{-7}	$\text{Mem?}(\text{base}'\text{'type_of})$
{-8}	$0 \leq \text{base}'\text{'offset}$
{-9}	$\text{base}'\text{'offset} < \text{max_linear_offset}$
{-10}	$\text{aligned?}(\text{bits_per_level} + \text{pe_size})(\text{offset}(\text{base}'))$
{-11}	$\text{rem}(\text{expt}(2, \text{min_page}))(\text{offset}(\text{addr}')) + \text{delta}' < \text{expt}(2, \text{min_page})$
{1}	$22 - 10 \times \text{lvl}' \geq 0$
{2}	$\text{base}' + \text{shift_bits_left}(\text{rem}(\text{expt}(2, \text{bits_per_level}))(\text{ndiv}(\text{offset}(\text{addr}'), \text{expt}(2, -1 \times \text{bits_per_level} \times \text{lvl}' - \text{bits_per_level} + \text{bus_width}))),$ $=$ $\text{base}' + \text{shift_bits_left}(\text{rem}(\text{expt}(2, \text{bits_per_level}))(\text{ndiv}(\text{offset}(\text{addr}' + \text{delta}'), \text{expt}(2, -1 \times \text{bits_per_level} \times \text{lvl}' - \text{bits_per_level} + \text{bus_width}))))$

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of xlat_idx_same_page_address.1.3.

`xlat_idx_same_page_address.2:`

{-1}	<code>Mem?(addr' 'type_of)</code>
{-2}	<code>0 ≤ addr' 'offset</code>
{-3}	<code>addr' 'offset < max_linear_offset</code>
{-4}	<code>delta' ≥ 0</code>
{-5}	<code>lvl' < max_level</code>
{-6}	<code>Mem?(base' 'type_of)</code>
{-7}	<code>0 ≤ base' 'offset</code>
{-8}	<code>base' 'offset < max_linear_offset</code>
{-9}	<code>aligned?(bits_per_level + pe_size)(offset(base'))</code>
{-10}	<code>rem(expt(2, min_page))(offset(addr')) + delta' < expt(2, min_page)</code>
{1}	<code>bus_width - bits_per_level - bits_per_level × lvl' ≥ 0</code>
{2}	<code>base' + shift_bits_left(rem(expt(2, bits_per_level))(ndiv(offset(addr'), expt(2, -1 × bits_per_level))</code> = <code>base' + shift_bits_left(rem(expt(2, bits_per_level))(ndiv(offset(addr' + delta'), expt(2, -1 × bits_per_level))</code>

Keeping (-5 1) and hiding *,
 Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of `xlat_idx_same_page_address.2`.
 Q.E.D.

C.117.60

Linear_Memory_Properties.xlat_ofs_same_page_address_TCC1

Terse proof for `xlat_ofs_same_page_address_TCC1`.

`xlat_ofs_same_page_address_TCC1:`

{1}	$\forall (addr: \text{Memory_Address_4G}, lvl: \text{Level}, a: \text{Memory_Address_4G}):$ $bus_width - (lvl + 1) \times bits_per_level[\text{Physical_memory}, pm_phy] \geq 0$
-----	--

Repeatedly Skolemizing and flattening,
 Installing automatic rewrites from: (bits_per_level! max_level! bus_width! pe_size!)
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 This completes the proof of `xlat_ofs_same_page_address_TCC1`.
 Q.E.D.

C.117.61

Linear_Memory_Properties.xlat_ofs_same_page_address_TCC2

Terse proof for `xlat_ofs_same_page_address_TCC2`.

`xlat_ofs_same_page_address_TCC2:`

{1}	$\forall (addr: \text{Memory_Address_4G}, \delta: \text{nat}, lvl: \text{Level},$ base: {a: Memory_Address_4G aligned?(bus_width - (lvl + 1) × bits_per_level [Physical_memory, pm_phy]) (offset(a))}): rem(expt(2, min_page))(offset(addr)) + δ < expt(2, min_page) ⊃ 0 ≤ (addr + δ)'offset ∧ (addr + δ)'offset < max_linear_offset
-----	--

Repeatedly Skolemizing and flattening,

Rewriting using delta_memory_address, matching in *,
 Expanding the definition of +,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 This completes the proof of xlat_ofs_same_page_address_TCC2.
 Q.E.D.

C.117.62 Linear_Memory_Properties.xlat_ofs_same_page_address

Terse proof for xlat_ofs_same_page_address.

xlat_ofs_same_page_address:

$\{1\} \quad \forall (\text{addr}: \text{Memory_Address_4G}, \delta: \text{nat}, \text{lvl}: \text{Level},$ $\quad \text{base}:$ $\quad \{a: \text{Memory_Address_4G} \mid$ $\quad \quad \text{aligned?}(\text{bus_width} - (\text{lvl} + 1) \times \text{bits_per_level}$ $\quad \quad \quad \text{offset}(a))\}:$ $\text{rem}(\text{expt}(2, \text{min_page}))(\text{offset}(\text{addr})) + \delta < \text{expt}(2, \text{min_page}) \supset$ $\text{xlat_ofs}(\text{lvl}, \text{base}, \text{addr}) + \delta = \text{xlat_ofs}(\text{lvl}, \text{base}, \text{addr} + \delta)$

Repeatedly Skolemizing and flattening,
 Expanding the definition of xlat_ofs,
 Expanding the definition of cut_bits,
 Installing automatic rewrites from: (expt_x0_aux! ndiv_1! bits_per_level! min_page! bus_width!)
 Simplifying, rewriting, and recording with decision procedures,
 Case splitting on min_page <= bus_width - bits_per_level - bits_per_level * lvl!1,
 we get 2 subgoals:

xlat_ofs_same_page_address.1:

$\{-1\} \quad \text{min_page} \leq \text{bus_width} - \text{bits_per_level} - \text{bits_per_level} \times \text{lvl}'$ $\{-2\} \quad \text{Mem?}(\text{addr}' \text{ type_of})$ $\{-3\} \quad 0 \leq \text{addr}' \text{ offset}$ $\{-4\} \quad \text{addr}' \text{ offset} < \text{max_linear_offset}$ $\{-5\} \quad \text{delta}' \geq 0$ $\{-6\} \quad \text{lvl}' < \text{max_level}$ $\{-7\} \quad \text{Mem?}(\text{base}' \text{ type_of})$ $\{-8\} \quad 0 \leq \text{base}' \text{ offset}$ $\{-9\} \quad \text{base}' \text{ offset} < \text{max_linear_offset}$ $\{-10\} \quad \text{aligned?}(22 - 10 \times \text{lvl}')(\text{offset}(\text{base}'))$ $\{-11\} \quad \text{rem}(\text{expt}(2, 12))(\text{offset}(\text{addr}')) + \text{delta}' < \text{expt}(2, 12)$ <hr/> $\{1\} \quad \text{base}' + \text{rem}(\text{expt}(2, 22 - 10 \times \text{lvl}'))(\text{offset}(\text{addr}')) + \text{delta}' =$ $\quad \text{base}' + \text{rem}(\text{expt}(2, 22 - 10 \times \text{lvl}'))(\text{offset}(\text{addr}' + \text{delta}'))$
--

Case splitting on rem(expt(2, bus_width - bits_per_level - bits_per_level * lvl!1)) (offset(addr!1)) + delta!1 = rem(expt(2, bus_width - bits_per_level - bits_per_level * lvl!1)) (offset(addr!1 + delta!1)),
 we get 3 subgoals:

C Proof scripts

`xlat_ofs_same_page_address.1.1:`

{-1}	$\text{rem}(\text{expt}(2, \text{bus_width} - \text{bits_per_level} - \text{bits_per_level} \times \text{lvl}')(\text{offset}(\text{addr}')) + \text{delta}'$ =
	$\text{rem}(\text{expt}(2, \text{bus_width} - \text{bits_per_level} - \text{bits_per_level} \times \text{lvl}')(\text{offset}(\text{addr}' + \text{delta}'))$
{-2}	$\text{min_page} \leq \text{bus_width} - \text{bits_per_level} - \text{bits_per_level} \times \text{lvl}'$
{-3}	$\text{Mem}?(\text{addr}' \text{'type_of}$
{-4}	$0 \leq \text{addr}' \text{'offset}$
{-5}	$\text{addr}' \text{'offset} < \text{max_linear_offset}$
{-6}	$\text{delta}' \geq 0$
{-7}	$\text{lvl}' < \text{max_level}$
{-8}	$\text{Mem}?(\text{base}' \text{'type_of}$
{-9}	$0 \leq \text{base}' \text{'offset}$
{-10}	$\text{base}' \text{'offset} < \text{max_linear_offset}$
{-11}	$\text{aligned}?(22 - 10 \times \text{lvl}')(\text{offset}(\text{base}'))$
{-12}	$\text{rem}(\text{expt}(2, 12))(\text{offset}(\text{addr}')) + \text{delta}' < \text{expt}(2, 12)$
{1}	$\text{base}' + \text{rem}(\text{expt}(2, 22 - 10 \times \text{lvl}')(\text{offset}(\text{addr}')) + \text{delta}' =$ $\text{base}' + \text{rem}(\text{expt}(2, 22 - 10 \times \text{lvl}')(\text{offset}(\text{addr}' + \text{delta}'))$

Expanding the definition of +,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `xlat_ofs_same_page_address.1.1`.

`xlat_ofs_same_page_address.1.2:`

{-1}	$\text{min_page} \leq \text{bus_width} - \text{bits_per_level} - \text{bits_per_level} \times \text{lvl}'$
{-2}	$\text{Mem}?(\text{addr}' \text{'type_of}$
{-3}	$0 \leq \text{addr}' \text{'offset}$
{-4}	$\text{addr}' \text{'offset} < \text{max_linear_offset}$
{-5}	$\text{delta}' \geq 0$
{-6}	$\text{lvl}' < \text{max_level}$
{-7}	$\text{Mem}?(\text{base}' \text{'type_of}$
{-8}	$0 \leq \text{base}' \text{'offset}$
{-9}	$\text{base}' \text{'offset} < \text{max_linear_offset}$
{-10}	$\text{aligned}?(22 - 10 \times \text{lvl}')(\text{offset}(\text{base}'))$
{-11}	$\text{rem}(\text{expt}(2, 12))(\text{offset}(\text{addr}')) + \text{delta}' < \text{expt}(2, 12)$
{1}	$\text{rem}(\text{expt}(2, \text{bus_width} - \text{bits_per_level} - \text{bits_per_level} \times \text{lvl}')(\text{offset}(\text{addr}')) + \text{delta}'$ =
	$\text{rem}(\text{expt}(2, \text{bus_width} - \text{bits_per_level} - \text{bits_per_level} \times \text{lvl}')(\text{offset}(\text{addr}' + \text{delta}'))$
{2}	$\text{base}' + \text{rem}(\text{expt}(2, 22 - 10 \times \text{lvl}')(\text{offset}(\text{addr}')) + \text{delta}' =$ $\text{base}' + \text{rem}(\text{expt}(2, 22 - 10 \times \text{lvl}')(\text{offset}(\text{addr}' + \text{delta}'))$

Hiding formulas: 2,

Expanding the definition of +,

Using lemma `rem_sum_assoc`,

we get 2 subgoals:

xlat_ofs_same_page_address.1.2.1:

{-1}	$\text{rem}(\text{expt}(2, \text{bus_width} - \text{bits_per_level} - \text{bits_per_level} \times \text{lvl}'))$ $(\text{addr}'\text{'offset} + \text{delta}')$ $=$ $\text{rem}(\text{expt}(2, \text{bus_width} - \text{bits_per_level} - \text{bits_per_level} \times \text{lvl}'))(\text{addr}'\text{'offset}) + \text{delta}'$ \equiv $\text{rem}(\text{expt}(2, \text{bus_width} - \text{bits_per_level} - \text{bits_per_level} \times \text{lvl}'))$ $(\text{addr}'\text{'offset})$ $< \text{expt}(2, \text{bus_width} - \text{bits_per_level} - \text{bits_per_level} \times \text{lvl}') - \text{delta}'$
{-2}	$\text{min_page} \leq \text{bus_width} - \text{bits_per_level} - \text{bits_per_level} \times \text{lvl}'$
{-3}	$\text{Mem}'(\text{addr}'\text{'type_of})$
{-4}	$0 \leq \text{addr}'\text{'offset}$
{-5}	$\text{addr}'\text{'offset} < \text{max_linear_offset}$
{-6}	$\text{delta}' \geq 0$
{-7}	$\text{lvl}' < \text{max_level}$
{-8}	$\text{Mem}'(\text{base}'\text{'type_of})$
{-9}	$0 \leq \text{base}'\text{'offset}$
{-10}	$\text{base}'\text{'offset} < \text{max_linear_offset}$
{-11}	$\text{aligned}'(22 - 10 \times \text{lvl}')(\text{offset}(\text{base}'))$
{-12}	$\text{rem}(\text{expt}(2, 12))(\text{offset}(\text{addr}')) + \text{delta}' < \text{expt}(2, 12)$
{1}	$\text{rem}(\text{expt}(2, -1 \times \text{bits_per_level} \times \text{lvl}' - \text{bits_per_level} + \text{bus_width}))(\text{offset}(\text{addr}')) + \text{delta}'$ $=$ $\text{rem}(\text{expt}(2, -1 \times \text{bits_per_level} \times \text{lvl}' - \text{bits_per_level} + \text{bus_width}))$ $(\text{addr}'\text{'offset} + \text{delta}')$

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Using lemma `aligned_add_below`,

we get 2 subgoals:

xlat_ofs_same_page_address.1.2.1.1:

{-1}	$\text{min_page} \leq \text{bus_width} - \text{bits_per_level} - \text{bits_per_level} \times \text{lvl}' \wedge$ $\text{rem}(\text{expt}(2, \text{bus_width} - \text{bits_per_level} - \text{bits_per_level} \times \text{lvl}')(\text{offset}(\text{addr}')) - \text{rem}(\text{expt}(2, \text{min_page}))$ $< \text{expt}(2, \text{bus_width} - \text{bits_per_level} - \text{bits_per_level} \times \text{lvl}')$ \wedge $\text{aligned}?(\text{min_page})$ $(\text{rem}(\text{expt}(2, \text{bus_width} - \text{bits_per_level} - \text{bits_per_level} \times \text{lvl}')(\text{offset}(\text{addr}')) - \text{rem}(\text{expt}(2, \text{min_page})))$ $\wedge \text{rem}(\text{expt}(2, \text{min_page}))(\text{offset}(\text{addr}')) + \text{delta}' < \text{expt}(2, \text{min_page})$ \supset $\text{rem}(\text{expt}(2, \text{bus_width} - \text{bits_per_level} - \text{bits_per_level} \times \text{lvl}')(\text{offset}(\text{addr}')) - \text{rem}(\text{expt}(2, \text{min_page}))$ $< \text{expt}(2, \text{bus_width} - \text{bits_per_level} - \text{bits_per_level} \times \text{lvl}')$
{-2}	$12 \leq 22 - 10 \times \text{lvl}'$
{-3}	$\text{Mem}?(\text{addr}' \text{'type_of})$
{-4}	$0 \leq \text{addr}' \text{'offset}$
{-5}	$\text{addr}' \text{'offset} < \text{max_linear_offset}$
{-6}	$\text{delta}' \geq 0$
{-7}	$\text{lvl}' < \text{max_level}$
{-8}	$\text{Mem}?(\text{base}' \text{'type_of})$
{-9}	$0 \leq \text{base}' \text{'offset}$
{-10}	$\text{base}' \text{'offset} < \text{max_linear_offset}$
{-11}	$\text{aligned}?(22 - 10 \times \text{lvl}')(\text{offset}(\text{base}'))$
{-12}	$\text{rem}(\text{expt}(2, 12))(\text{offset}(\text{addr}')) + \text{delta}' < \text{expt}(2, 12)$
{1}	$\text{rem}(\text{expt}(2, 22 - 10 \times \text{lvl}')(\text{addr}' \text{'offset} + \text{delta}')) =$ $\text{rem}(\text{expt}(2, 22 - 10 \times \text{lvl}')(\text{addr}' \text{'offset}) + \text{delta}'$
{2}	$\text{rem}(\text{expt}(2, 22 - 10 \times \text{lvl}')(\text{addr}' \text{'offset}) <$ $\text{expt}(2, 22 - 10 \times \text{lvl}') - \text{delta}'$
{3}	$\text{rem}(\text{expt}(2, 22 - 10 \times \text{lvl}')(\text{offset}(\text{addr}')) + \text{delta}' =$ $\text{rem}(\text{expt}(2, 22 - 10 \times \text{lvl}')(\text{addr}' \text{'offset} + \text{delta}'))$

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting, we get 2 subgoals:

xlat_ofs_same_page_address.1.2.1.1.1:

{-1}	$12 \leq 22 - 10 \times \text{lvl}'$
{-2}	$\text{Mem}?(\text{addr}' \text{'type_of})$
{-3}	$0 \leq \text{addr}' \text{'offset}$
{-4}	$\text{addr}' \text{'offset} < \text{max_linear_offset}$
{-5}	$\text{delta}' \geq 0$
{-6}	$\text{lvl}' < \text{max_level}$
{-7}	$\text{Mem}?(\text{base}' \text{'type_of})$
{-8}	$0 \leq \text{base}' \text{'offset}$
{-9}	$\text{base}' \text{'offset} < \text{max_linear_offset}$
{-10}	$\text{aligned}?(22 - 10 \times \text{lvl}')(\text{offset}(\text{base}'))$
{-11}	$\text{rem}(\text{expt}(2, 12))(\text{offset}(\text{addr}')) + \text{delta}' < \text{expt}(2, 12)$
{1}	$\text{aligned}?(12)$ $(\text{rem}(\text{expt}(2, 22 - 10 \times \text{lvl}')(\text{offset}(\text{addr}')) - \text{rem}(\text{expt}(2, 12))(\text{offset}(\text{addr}'))))$
{2}	$\text{rem}(\text{expt}(2, 22 - 10 \times \text{lvl}')(\text{addr}' \text{'offset} + \text{delta}')) =$ $\text{rem}(\text{expt}(2, 22 - 10 \times \text{lvl}')(\text{addr}' \text{'offset}) + \text{delta}'$
{3}	$\text{rem}(\text{expt}(2, 22 - 10 \times \text{lvl}')(\text{addr}' \text{'offset}) <$ $\text{expt}(2, 22 - 10 \times \text{lvl}') - \text{delta}'$
{4}	$\text{rem}(\text{expt}(2, 22 - 10 \times \text{lvl}')(\text{offset}(\text{addr}')) + \text{delta}' =$ $\text{rem}(\text{expt}(2, 22 - 10 \times \text{lvl}')(\text{addr}' \text{'offset} + \text{delta}'))$

Expanding the definition of aligned?,

Using lemma `rem_def2`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Rewriting using `rem_rem`, matching in `*`,

Using lemma `divides_expt_gt`,

Rewriting using `divides_reflexive`, matching in `*`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `xlat_ofs_same_page_address.1.2.1.1.1`.

`xlat_ofs_same_page_address.1.2.1.1.2`:

{-1}	$12 \leq 22 - 10 \times \text{lvl}'$
{-2}	<code>Mem?(addr' type_of)</code>
{-3}	$0 \leq \text{addr}'\text{offset}$
{-4}	$\text{addr}'\text{offset} < \text{max_linear_offset}$
{-5}	$\text{delta}' \geq 0$
{-6}	$\text{lvl}' < \text{max_level}$
{-7}	<code>Mem?(base' type_of)</code>
{-8}	$0 \leq \text{base}'\text{offset}$
{-9}	$\text{base}'\text{offset} < \text{max_linear_offset}$
{-10}	<code>aligned?(22 - 10 × lvl')(offset(base'))</code>
{-11}	$\text{rem}(\text{expt}(2, 12))(\text{offset}(\text{addr}')) + \text{delta}' < \text{expt}(2, 12)$
{1}	$\text{rem}(\text{expt}(2, 22 - 10 \times \text{lvl}'))(\text{offset}(\text{addr}')) - \text{rem}(\text{expt}(2, 12))(\text{offset}(\text{addr}')) < \text{expt}(2, 22 - 10 \times \text{lvl}')$
{2}	$\text{rem}(\text{expt}(2, 22 - 10 \times \text{lvl}'))(\text{addr}'\text{offset} + \text{delta}') = \text{rem}(\text{expt}(2, 22 - 10 \times \text{lvl}'))(\text{addr}'\text{offset}) + \text{delta}'$
{3}	$\text{rem}(\text{expt}(2, 22 - 10 \times \text{lvl}'))(\text{addr}'\text{offset}) < \text{expt}(2, 22 - 10 \times \text{lvl}') - \text{delta}'$
{4}	$\text{rem}(\text{expt}(2, 22 - 10 \times \text{lvl}'))(\text{offset}(\text{addr}')) + \text{delta}' = \text{rem}(\text{expt}(2, 22 - 10 \times \text{lvl}'))(\text{addr}'\text{offset} + \text{delta}')$

Adding type constraints for `rem(expt(2, bus_width - bits_per_level - bits_per_level * lvl!)) (offset(addr!))`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `xlat_ofs_same_page_address.1.2.1.1.2`.

`xlat_ofs_same_page_address.1.2.1.2`:

{-1}	$12 \leq 22 - 10 \times \text{lvl}'$
{-2}	<code>Mem?(addr' type_of)</code>
{-3}	$0 \leq \text{addr}'\text{offset}$
{-4}	$\text{addr}'\text{offset} < \text{max_linear_offset}$
{-5}	$\text{delta}' \geq 0$
{-6}	$\text{lvl}' < \text{max_level}$
{-7}	<code>Mem?(base' type_of)</code>
{-8}	$0 \leq \text{base}'\text{offset}$
{-9}	$\text{base}'\text{offset} < \text{max_linear_offset}$
{-10}	<code>aligned?(22 - 10 × lvl')(offset(base'))</code>
{-11}	$\text{rem}(\text{expt}(2, 12))(\text{offset}(\text{addr}')) + \text{delta}' < \text{expt}(2, 12)$
{1}	$\text{rem}(\text{expt}(2, 22 - 10 \times \text{lvl}'))(\text{offset}(\text{addr}')) - \text{rem}(\text{expt}(2, 12))(\text{offset}(\text{addr}')) \geq 0$
{2}	$\text{rem}(\text{expt}(2, 22 - 10 \times \text{lvl}'))(\text{addr}'\text{offset} + \text{delta}') = \text{rem}(\text{expt}(2, 22 - 10 \times \text{lvl}'))(\text{addr}'\text{offset}) + \text{delta}'$
{3}	$\text{rem}(\text{expt}(2, 22 - 10 \times \text{lvl}'))(\text{addr}'\text{offset}) < \text{expt}(2, 22 - 10 \times \text{lvl}') - \text{delta}'$
{4}	$\text{rem}(\text{expt}(2, 22 - 10 \times \text{lvl}'))(\text{offset}(\text{addr}')) + \text{delta}' = \text{rem}(\text{expt}(2, 22 - 10 \times \text{lvl}'))(\text{addr}'\text{offset} + \text{delta}')$

C Proof scripts

Using lemma `rem_floor`,

Rewriting using `rem_rem`, matching in `*`,

we get 2 subgoals:

`xlat_ofs_same_page_address.1.2.1.2.1`:

$$\begin{array}{l}
 \{-1\} \quad \text{rem}(\text{expt}(2, \text{bus_width} - \text{bits_per_level} - \text{bits_per_level} \times \text{lvl}')) \\
 \quad \quad (\text{offset}(\text{addr}')) \\
 \quad \quad = \\
 \quad \quad \text{rem}(\text{expt}(2, 12))(\text{offset}(\text{addr}')) + \text{expt}(2, 12) \times \text{floor}(\text{rem}(\text{expt}(2, \text{bus_width} - \text{bits_per_level} - \\
 \{-2\} \quad 12 \leq 22 - 10 \times \text{lvl}') \\
 \{-3\} \quad \text{Mem}?(\text{addr}' \text{'type_of}) \\
 \{-4\} \quad 0 \leq \text{addr}' \text{'offset} \\
 \{-5\} \quad \text{addr}' \text{'offset} < \text{max_linear_offset} \\
 \{-6\} \quad \text{delta}' \geq 0 \\
 \{-7\} \quad \text{lvl}' < \text{max_level} \\
 \{-8\} \quad \text{Mem}(\text{base}' \text{'type_of}) \\
 \{-9\} \quad 0 \leq \text{base}' \text{'offset} \\
 \{-10\} \quad \text{base}' \text{'offset} < \text{max_linear_offset} \\
 \{-11\} \quad \text{aligned}?(22 - 10 \times \text{lvl}')(\text{offset}(\text{base}')) \\
 \{-12\} \quad \text{rem}(\text{expt}(2, 12))(\text{offset}(\text{addr}')) + \text{delta}' < \text{expt}(2, 12) \\
 \hline
 \{1\} \quad \text{rem}(\text{expt}(2, 22 - 10 \times \text{lvl}')(\text{offset}(\text{addr}')) - \text{rem}(\text{expt}(2, 12))(\text{offset}(\text{addr}')) \\
 \quad \quad \geq 0 \\
 \{2\} \quad \text{rem}(\text{expt}(2, 22 - 10 \times \text{lvl}')(\text{addr}' \text{'offset} + \text{delta}')) = \\
 \quad \quad \text{rem}(\text{expt}(2, 22 - 10 \times \text{lvl}')(\text{addr}' \text{'offset}) + \text{delta}') \\
 \{3\} \quad \text{rem}(\text{expt}(2, 22 - 10 \times \text{lvl}')(\text{addr}' \text{'offset}) < \\
 \quad \quad \text{expt}(2, 22 - 10 \times \text{lvl}') - \text{delta}' \\
 \{4\} \quad \text{rem}(\text{expt}(2, 22 - 10 \times \text{lvl}')(\text{offset}(\text{addr}')) + \text{delta}') = \\
 \quad \quad \text{rem}(\text{expt}(2, 22 - 10 \times \text{lvl}')(\text{addr}' \text{'offset} + \text{delta}'))
 \end{array}$$

Adding type constraints for `expt(2, min_page) * floor(rem(expt(2, bus_width - bits_per_level - bits_per_level * lvl!1)) (offset(addr!1)) / expt(2, min_page))`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `xlat_ofs_same_page_address.1.2.1.2.1`.

xlat_ofs_same_page_address.1.2.1.2.2:

{-1}	$\text{rem}(\text{expt}(2, 22 - 10 \times \text{lvl}')(\text{offset}(\text{addr}')) =$
	$\text{rem}(\text{expt}(2, 12))(\text{rem}(\text{expt}(2, 22 - 10 \times \text{lvl}')(\text{offset}(\text{addr}')))) + \text{expt}(2, 12) \times \text{floor}(\text{rem}(\text{expt}(2, 22 - 10 \times$
{-2}	$12 \leq 22 - 10 \times \text{lvl}'$
{-3}	$\text{Mem}?(\text{addr}' \text{'type_of})$
{-4}	$0 \leq \text{addr}' \text{'offset}$
{-5}	$\text{addr}' \text{'offset} < \text{max_linear_offset}$
{-6}	$\text{delta}' \geq 0$
{-7}	$\text{lvl}' < \text{max_level}$
{-8}	$\text{Mem}?(\text{base}' \text{'type_of})$
{-9}	$0 \leq \text{base}' \text{'offset}$
{-10}	$\text{base}' \text{'offset} < \text{max_linear_offset}$
{-11}	$\text{aligned}?(22 - 10 \times \text{lvl}')(\text{offset}(\text{base}'))$
{-12}	$\text{rem}(\text{expt}(2, 12))(\text{offset}(\text{addr}')) + \text{delta}' < \text{expt}(2, 12)$
{1}	$\text{divides}(\text{expt}(2, 12), \text{expt}(2, 22 - 10 \times \text{lvl}'))$
{2}	$\text{rem}(\text{expt}(2, 22 - 10 \times \text{lvl}')(\text{offset}(\text{addr}')) - \text{rem}(\text{expt}(2, 12))(\text{offset}(\text{addr}'))$
	≥ 0
{3}	$\text{rem}(\text{expt}(2, 22 - 10 \times \text{lvl}')(\text{addr}' \text{'offset} + \text{delta}') =$
	$\text{rem}(\text{expt}(2, 22 - 10 \times \text{lvl}')(\text{addr}' \text{'offset}) + \text{delta}'$
{4}	$\text{rem}(\text{expt}(2, 22 - 10 \times \text{lvl}')(\text{addr}' \text{'offset}) <$
	$\text{expt}(2, 22 - 10 \times \text{lvl}') - \text{delta}'$
{5}	$\text{rem}(\text{expt}(2, 22 - 10 \times \text{lvl}')(\text{offset}(\text{addr}')) + \text{delta}' =$
	$\text{rem}(\text{expt}(2, 22 - 10 \times \text{lvl}')(\text{addr}' \text{'offset} + \text{delta}')$

Using lemma divides_expt_gt,

Rewriting using divides_reflexive, matching in *,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of xlat_ofs_same_page_address.1.2.1.2.2.

xlat_ofs_same_page_address.1.2.2:

{-1}	$\text{min_page} \leq \text{bus_width} - \text{bits_per_level} - \text{bits_per_level} \times \text{lvl}'$
{-2}	$\text{Mem}?(\text{addr}' \text{'type_of})$
{-3}	$0 \leq \text{addr}' \text{'offset}$
{-4}	$\text{addr}' \text{'offset} < \text{max_linear_offset}$
{-5}	$\text{delta}' \geq 0$
{-6}	$\text{lvl}' < \text{max_level}$
{-7}	$\text{Mem}?(\text{base}' \text{'type_of})$
{-8}	$0 \leq \text{base}' \text{'offset}$
{-9}	$\text{base}' \text{'offset} < \text{max_linear_offset}$
{-10}	$\text{aligned}?(22 - 10 \times \text{lvl}')(\text{offset}(\text{base}'))$
{-11}	$\text{rem}(\text{expt}(2, 12))(\text{offset}(\text{addr}')) + \text{delta}' < \text{expt}(2, 12)$
{1}	$22 - 10 \times \text{lvl}' \geq 0$
{2}	$\text{rem}(\text{expt}(2, -1 \times \text{bits_per_level} \times \text{lvl}' - \text{bits_per_level} + \text{bus_width}))(\text{offset}(\text{addr}')) + \text{delta}'$
	$=$
	$\text{rem}(\text{expt}(2, -1 \times \text{bits_per_level} \times \text{lvl}' - \text{bits_per_level} + \text{bus_width}))$
	$(\text{addr}' \text{'offset} + \text{delta}')$

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of xlat_ofs_same_page_address.1.2.2.

`xlat_ofs_same_page_address.1.3:`

{-1}	$\text{min_page} \leq \text{bus_width} - \text{bits_per_level} - \text{bits_per_level} \times \text{lvl}'$
{-2}	$\text{Mem?}(\text{addr}' \text{ 'type_of})$
{-3}	$0 \leq \text{addr}' \text{ 'offset}$
{-4}	$\text{addr}' \text{ 'offset} < \text{max_linear_offset}$
{-5}	$\text{delta}' \geq 0$
{-6}	$\text{lvl}' < \text{max_level}$
{-7}	$\text{Mem?}(\text{base}' \text{ 'type_of})$
{-8}	$0 \leq \text{base}' \text{ 'offset}$
{-9}	$\text{base}' \text{ 'offset} < \text{max_linear_offset}$
{-10}	$\text{aligned?}(22 - 10 \times \text{lvl}')(\text{offset}(\text{base}'))$
{-11}	$\text{rem}(\text{expt}(2, 12))(\text{offset}(\text{addr}')) + \text{delta}' < \text{expt}(2, 12)$
{1}	$22 - 10 \times \text{lvl}' \geq 0$
{2}	$\text{base}' + \text{rem}(\text{expt}(2, 22 - 10 \times \text{lvl}'))(\text{offset}(\text{addr}')) + \text{delta}' = \text{base}' + \text{rem}(\text{expt}(2, 22 - 10 \times \text{lvl}'))(\text{offset}(\text{addr}' + \text{delta}'))$

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `xlat_ofs_same_page_address.1.3`.

`xlat_ofs_same_page_address.2:`

{-1}	$\text{Mem?}(\text{addr}' \text{ 'type_of})$
{-2}	$0 \leq \text{addr}' \text{ 'offset}$
{-3}	$\text{addr}' \text{ 'offset} < \text{max_linear_offset}$
{-4}	$\text{delta}' \geq 0$
{-5}	$\text{lvl}' < \text{max_level}$
{-6}	$\text{Mem?}(\text{base}' \text{ 'type_of})$
{-7}	$0 \leq \text{base}' \text{ 'offset}$
{-8}	$\text{base}' \text{ 'offset} < \text{max_linear_offset}$
{-9}	$\text{aligned?}(22 - 10 \times \text{lvl}')(\text{offset}(\text{base}'))$
{-10}	$\text{rem}(\text{expt}(2, 12))(\text{offset}(\text{addr}')) + \text{delta}' < \text{expt}(2, 12)$
{1}	$\text{min_page} \leq \text{bus_width} - \text{bits_per_level} - \text{bits_per_level} \times \text{lvl}'$
{2}	$\text{base}' + \text{rem}(\text{expt}(2, 22 - 10 \times \text{lvl}'))(\text{offset}(\text{addr}')) + \text{delta}' = \text{base}' + \text{rem}(\text{expt}(2, 22 - 10 \times \text{lvl}'))(\text{offset}(\text{addr}' + \text{delta}'))$

Keeping (-5 1) and hiding *,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `xlat_ofs_same_page_address.2`.
Q.E.D.

C.117.63

Linear_Memory_Properties.translate_same_page_address_ok

Terse proof for `translate_same_page_address_ok`.

`translate_same_page_address_ok:`

{1}	$\forall (s: \text{Linear_memory}[\text{Physical_memory}, \text{pm_phy}], \text{addr}: \text{Memory_Address_4G}, \delta: \text{nat}, \text{lvl}: \text{Level},$ $\text{base}: \{a: \text{Memory_Address_4G} \mid \text{aligned?}(\text{bits_per_level} + \text{pe_size})(\text{offset}(a))\},$ $\text{access}: \text{Memory_access}, \text{priv}: \text{Memory_privilege}):$ $\text{rem}(\text{expt}(2, \text{min_page}))(\text{offset}(\text{addr})) + \delta < \text{expt}(2, \text{min_page}) \wedge$ $\text{OK?}(\text{translate}(\text{lvl}, \text{base}, \text{addr}, \text{access}, \text{priv})(s))$ $\supset \text{OK?}(\text{translate}(\text{lvl}, \text{base}, \text{addr} + \delta, \text{access}, \text{priv})(s))$
-----	--

Repeatedly Skolemizing and flattening,

Expanding the definition of translate,

Case splitting on $xlat_idx(lvl!1, base!1, addr!1 + delta!1) = xlat_idx(lvl!1, base!1, addr!1)$,

we get 2 subgoals:

`translate_same_page_address_ok.1:`

```

{-1}  xlat_idx(lvl', base', addr' + delta') = xlat_idx(lvl', base', addr')
{-2}  Mem?(addr' type_of)
{-3}  0 ≤ addr' offset
{-4}  addr' offset < max_linear_offset
{-5}  delta' ≥ 0
{-6}  lvl' < max_level
{-7}  Mem?(base' type_of)
{-8}  0 ≤ base' offset
{-9}  base' offset < max_linear_offset
{-10} aligned?(bits_per_level + pe_size)(offset(base'))
{-11} rem(expt(2, min_page))(offset(addr')) + delta' < expt(2, min_page)
{-12} OK?((read_data(pm_phy, paging_data_type(lvl'))(xlat_idx(lvl', base', addr')) ##
        (λ (pe: (range_pt(lvl'))):
          IF paging_type?(lvl', pe)
            THEN IF present?(pe)
              THEN IF accessible?(pe, access') ∧ privileged?(pe, priv')
                THEN write_data(pm_phy, paging_data_type(lvl'))
                      (xlat_idx(lvl', base', addr'),
                       set_reference(pe, access'))
                ## ok_result(is_leaf?(pe), base(pe))
              ELSE raise_fault(priv', access', TRUE, addr')
            ENDIF
          ELSE raise_fault(priv', access', FALSE, addr')
          ENDIF
        ELSE fatal_result
        ENDIF))
        (s'))
-----
{1}  OK?((read_data(pm_phy, paging_data_type(lvl'))
        (xlat_idx(lvl', base', addr' + delta'))
        ##
        (λ (pe: (range_pt(lvl'))):
          IF paging_type?(lvl', pe)
            THEN IF present?(pe)
              THEN IF accessible?(pe, access') ∧ privileged?(pe, priv')
                THEN write_data(pm_phy, paging_data_type(lvl'))
                      (xlat_idx(lvl', base', addr' + delta'),
                       set_reference(pe, access'))
                ## ok_result(is_leaf?(pe), base(pe))
              ELSE raise_fault(priv', access', TRUE, addr' + delta')
            ENDIF
          ELSE raise_fault(priv', access', FALSE, addr' + delta')
          ENDIF
        ELSE fatal_result
        ENDIF))
        (s'))

```

Installing automatic rewrites from: (##! expr_2_super! expr_2_super_res! raise_fault! excep-

tion_result! ok_result! fatal_result!)

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `translate_same_page_address_ok.1`.

`translate_same_page_address_ok.2`:

```

{-1} Mem?(addr' 'type_of)
{-2} 0 ≤ addr' 'offset
{-3} addr' 'offset < max_linear_offset
{-4} delta' ≥ 0
{-5} lvl' < max_level
{-6} Mem?(base' 'type_of)
{-7} 0 ≤ base' 'offset
{-8} base' 'offset < max_linear_offset
{-9} aligned?(bits_per_level + pe_size)(offset(base'))
{-10} rem(expt(2, min_page))(offset(addr')) + delta' < expt(2, min_page)
{-11} OK?((read_data(pm_phy, paging_data_type(lvl'))(xlat_idx(lvl', base', addr'))) ##
      (λ (pe: (range_pt(lvl'))):
        IF paging_type?(lvl', pe)
          THEN IF present?(pe)
            THEN IF accessible?(pe, access') ∧ privileged?(pe, priv')
              THEN write_data(pm_phy, paging_data_type(lvl'))
                    (xlat_idx(lvl', base', addr'),
                     set_reference(pe, access'))
              ## ok_result(is_leaf?(pe), base(pe))
            ELSE raise_fault(priv', access', TRUE, addr')
          ENDIF
        ELSE raise_fault(priv', access', FALSE, addr')
        ENDIF
      ELSE fatal_result
      ENDIF))
      (s'))
-----
{1} xlat_idx(lvl', base', addr' + delta') = xlat_idx(lvl', base', addr')
{2} OK?((read_data(pm_phy, paging_data_type(lvl'))
      (xlat_idx(lvl', base', addr' + delta')))
      ##
      (λ (pe: (range_pt(lvl'))):
        IF paging_type?(lvl', pe)
          THEN IF present?(pe)
            THEN IF accessible?(pe, access') ∧ privileged?(pe, priv')
              THEN write_data(pm_phy, paging_data_type(lvl'))
                    (xlat_idx(lvl', base', addr' + delta'),
                     set_reference(pe, access'))
              ## ok_result(is_leaf?(pe), base(pe))
            ELSE raise_fault(priv', access', TRUE, addr' + delta')
          ENDIF
        ELSE raise_fault(priv', access', FALSE, addr' + delta')
        ENDIF
      ELSE fatal_result
      ENDIF))
      (s'))

```

Hiding formulas: (-9 2),

Using lemma `xlat_idx_same_page_address`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `translate_same_page_address_ok.2`.

Q.E.D.

C.117.64

Linear_Memory_Properties.translate_same_page_address_result

Terse proof for `translate_same_page_address_result`.

`translate_same_page_address_result`:

<pre>{1} ∀ (s: Linear_memory[Physical_memory, pm_phy], addr: Memory_Address_4G, δ: nat, lvl: Level, base: {a: Memory_Address_4G aligned?(bits_per_level + pe_size)(offset(a))}, access: Memory_access, priv: Memory_privilege): rem(expt(2, min_page))(offset(addr)) + δ < expt(2, min_page) ∧ OK?(translate(lvl, base, addr, access, priv)(s)) ⊃ translate(lvl, base, addr, access, priv)(s) = translate(lvl, base, addr + δ, access, priv)(s)</pre>

Repeatedly Skolemizing and flattening,

Using lemma `translate_same_page_address_ok`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Expanding the definition of `translate`,

Case splitting on `xlat_idx(lvl!1, base!1, addr!1 + delta!1) = xlat_idx(lvl!1, base!1, addr!1)`,

we get 2 subgoals:


```
translate_same_page_address_result.1:
```

```
{-1}  xlat_idx(lvl', base', addr' + delta') = xlat_idx(lvl', base', addr')
{-2}  rem(expt(2, min_page))(offset(addr')) + delta' < expt(2, min_page)
{-3}  OK?((read_data(pm_phy, paging_data_type(lvl'))(xlat_idx(lvl', base', addr'))) ##
      (λ (pe: (range_pt(lvl'))):
        IF paging_type?(lvl', pe)
          THEN IF present?(pe)
            THEN IF accessible?(pe, access') ∧ privileged?(pe, priv')
              THEN write_data(pm_phy, paging_data_type(lvl'))
                    (xlat_idx(lvl', base', addr'),
                     set_reference(pe, access'))
                    ## ok_result(is_leaf?(pe), base(pe))
              ELSE raise_fault(priv', access', TRUE, addr')
            ENDIF
          ELSE raise_fault(priv', access', FALSE, addr')
          ENDIF
        ELSE fatal_result
        ENDIF))
      (s'))
{-4}  OK?((read_data(pm_phy, paging_data_type(lvl'))
      (xlat_idx(lvl', base', addr' + delta')))
      ##
      (λ (pe: (range_pt(lvl'))):
        IF paging_type?(lvl', pe)
          THEN IF present?(pe)
            THEN IF accessible?(pe, access') ∧ privileged?(pe, priv')
              THEN write_data(pm_phy, paging_data_type(lvl'))
                    (xlat_idx(lvl', base', addr' + delta'),
                     set_reference(pe, access'))
                    ## ok_result(is_leaf?(pe), base(pe))
              ELSE raise_fault(priv', access', TRUE, addr' + delta')
            ENDIF
          ELSE raise_fault(priv', access', FALSE, addr' + delta')
          ENDIF
        ELSE fatal_result
        ENDIF))
      (s'))
{-5}  Mem?(addr' type_of)
{-6}  0 ≤ addr' offset
{-7}  addr' offset < max_linear_offset
{-8}  delta' ≥ 0
{-9}  lvl' < max_level
{-10} Mem?(base' type_of)
{-11} 0 ≤ base' offset
{-12} base' offset < max_linear_offset
{-13} aligned?(bits_per_level + pe_size)(offset(base'))


---


{1}  (read_data(pm_phy, paging_data_type(lvl'))(xlat_idx(lvl', base', addr'))) ##
      (λ (pe: (range_pt(lvl'))):
        IF paging_type?(lvl', pe)
          THEN IF present?(pe)
            THEN IF accessible?(pe, access') ∧ privileged?(pe, priv')
              THEN write_data(pm_phy, paging_data_type(lvl'))
                    (xlat_idx(lvl', base', addr'),
                     set_reference(pe, access'))
                    ## ok_result(is_leaf?(pe), base(pe))
              ELSE raise_fault(priv', access', TRUE, addr')
            ENDIF
          ELSE raise_fault(priv', access', FALSE, addr')
          ENDIF
        ELSE fatal_result
        ENDIF))
      (s'))
=
(read_data(pm_phy, paging_data_type(lvl'))
```

C.117 Proofs for Linear_Memory_Properties (challenge-linear.pvs)

Installing automatic rewrites from: (`##!` `expr_2_super!` `expr_2_super_res!` `raise_fault!` `exception_result!` `fatal_result!` `ok_result!`)

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `translate_same_page_address_result.1`.

translate_same_page_address_result.2:

```

{-1}  rem(expt(2, min_page))(offset(addr')) + delta' < expt(2, min_page)
{-2}  OK?((read_data(pm_phy, paging_data_type(lvl'))(xlat_idx(lvl', base', addr'))) ##
      (λ (pe: (range_pt(lvl'))):
        IF paging_type?(lvl', pe)
        THEN IF present?(pe)
          THEN IF accessible?(pe, access') ∧ privileged?(pe, priv')
            THEN write_data(pm_phy, paging_data_type(lvl'))
                  (xlat_idx(lvl', base', addr'),
                   set_reference(pe, access'))
            ## ok_result(is_leaf?(pe), base(pe))
          ELSE raise_fault(priv', access', TRUE, addr')
        ENDIF
        ELSE raise_fault(priv', access', FALSE, addr')
      ENDIF
      ELSE fatal_result
    ENDIF))
      (s'))
{-3}  OK?((read_data(pm_phy, paging_data_type(lvl'))
      (xlat_idx(lvl', base', addr' + delta')))
      ##
      (λ (pe: (range_pt(lvl'))):
        IF paging_type?(lvl', pe)
        THEN IF present?(pe)
          THEN IF accessible?(pe, access') ∧ privileged?(pe, priv')
            THEN write_data(pm_phy, paging_data_type(lvl'))
                  (xlat_idx(lvl', base', addr' + delta'),
                   set_reference(pe, access'))
            ## ok_result(is_leaf?(pe), base(pe))
          ELSE raise_fault(priv', access', TRUE, addr' + delta')
        ENDIF
        ELSE raise_fault(priv', access', FALSE, addr' + delta')
      ENDIF
      ELSE fatal_result
    ENDIF))
      (s'))
{-4}  Mem?(addr' type_of)
{-5}  0 ≤ addr' offset
{-6}  addr' offset < max_linear_offset
{-7}  delta' ≥ 0
{-8}  lvl' < max_level
{-9}  Mem?(base' type_of)
{-10} 0 ≤ base' offset
{-11} base' offset < max_linear_offset
{-12} aligned?(bits_per_level + pe_size)(offset(base'))
-----
{1}  xlat_idx(lvl', base', addr' + delta') = xlat_idx(lvl', base', addr')
{2}  (read_data(pm_phy, paging_data_type(lvl'))(xlat_idx(lvl', base', addr'))) ##
      (λ (pe: (range_pt(lvl'))):
        IF paging_type?(lvl', pe)
        THEN IF present?(pe)
          THEN IF accessible?(pe, access') ∧ privileged?(pe, priv')
            THEN write_data(pm_phy, paging_data_type(lvl'))
                  (xlat_idx(lvl', base', addr'),
                   set_reference(pe, access'))
            ## ok_result(is_leaf?(pe), base(pe))
          ELSE raise_fault(priv', access', TRUE, addr')
        ENDIF
        ELSE raise_fault(priv', access', FALSE, addr')
      ENDIF
      ELSE fatal_result
    ENDIF))
      (s'))
=
(read_data(pm_phy, paging_data_type(lvl'))

```

Hiding formulas: (-2 -3 2),
 Using lemma xlat_idx_same_page_address,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 This completes the proof of translate_same_page_address_result.2.
 Q.E.D.

C.117.65 Linear_Memory_Properties.linear_resolve_same_page_address_ok_TCC1

Terse proof for linear_resolve_same_page_address_ok_TCC1.

linear_resolve_same_page_address_ok_TCC1:

$\{1\} \quad \forall (s: \text{Linear_memory}[\text{Physical_memory}, \text{pm_phy}], \text{addr}: \text{Memory_Address_4G}, \delta: \text{nat}, \text{ac}: \text{Memory_access}):$ $\text{rem}(\text{expt}(2, \text{min_page}))(\text{offset}(\text{addr})) + \delta < \text{expt}(2, \text{min_page}) \wedge$ $\text{OK?}(\text{linear_resolve}(\text{addr}, \text{ac})(s))$ $\supset 0 \leq (\text{addr} + \delta)' \text{offset} \wedge (\text{addr} + \delta)' \text{offset} < \text{max_linear_offset}$
--

Repeatedly Skolemizing and flattening,
 Rewriting using delta_memory_address, matching in *,
 Expanding the definition of +,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 This completes the proof of linear_resolve_same_page_address_ok_TCC1.
 Q.E.D.

C.117.66 Linear_Memory_Properties.linear_resolve_same_page_address_ok

Terse proof for linear_resolve_same_page_address_ok.

linear_resolve_same_page_address_ok:

$\{1\} \quad \forall (s: \text{Linear_memory}[\text{Physical_memory}, \text{pm_phy}], \text{addr}: \text{Memory_Address_4G}, \delta: \text{nat}, \text{ac}: \text{Memory_access}):$ $\text{rem}(\text{expt}(2, \text{min_page}))(\text{offset}(\text{addr})) + \delta < \text{expt}(2, \text{min_page}) \wedge$ $\text{OK?}(\text{linear_resolve}(\text{addr}, \text{ac})(s))$ $\supset \text{OK?}(\text{linear_resolve}(\text{addr} + \delta, \text{ac})(s))$
--

Repeatedly Skolemizing and flattening,
 Expanding the definition of linear_resolve,
 Installing automatic rewrites from: (##! translate_same_page_address_ok! translate_same_page_address_result! min_linear! max_linear! Mem! Address_Helpers.<=! <! xlat_idx_memory_address! xlat_idx_memory_address2! ptab_lvl! pdir_lvl! max_level! expr_2_super! expr_2_super_res! ok_result! bus_width! bits_per_level! pe_size!)
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 we get 2 subgoals:

```
linear_resolve_same_page_address_ok.1:
```

```
{-1} Mem?(addr' 'type_of)
{-2} 0 ≤ addr' 'offset
{-3} addr' 'offset < max_linear_offset
{-4} delta' ≥ 0
{-5} rem(expt(2, min_page))(offset(addr')) + delta' < expt(2, min_page)
{-6} OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s'))
{-7} OK?(read_data(pm_phy, pdbrr_data_type)(PDBR)
      (state(read_data(pm_phy, segment_reg_data_type)(CS)(s'))))
{-8} OK?(translate(0,
      data(read_data(pm_phy, pdbrr_data_type)(PDBR)
            (state(read_data(pm_phy, seg-
segment_reg_data_type)(CS)
            (s')))) 'base_addr,
      addr', ac',
      segment_to_priv(data(read_data(pm_phy, seg-
segment_reg_data_type)(CS)(s'))))
      (state(read_data(pm_phy, pdbrr_data_type)(PDBR)
            (state(read_data(pm_phy, seg-
segment_reg_data_type)(CS)
            (s'))))))))
{-9} data(translate(0,
      data(read_data(pm_phy, pdbrr_data_type)(PDBR)
            (state(read_data(pm_phy, seg-
segment_reg_data_type)(CS)
            (s')))) 'base_addr,
      addr', ac',
      segment_to_priv(data(read_data(pm_phy, seg-
segment_reg_data_type)(CS)
            (s'))))
      (state(read_data(pm_phy, pdbrr_data_type)(PDBR)
            (state(read_data(pm_phy, seg-
segment_reg_data_type)(CS)
            (s')))))))) '1
{-10} OK?(OK(state(translate(0,
      data(read_data(pm_phy, pdbrr_data_type)(PDBR)
            (state(read_data(pm_phy, seg-
segment_reg_data_type)
            (CS)
            (s')))) 'base_addr,
      addr', ac',
      segment_to_priv(data(read_data(pm_phy, seg-
segment_reg_data_type)(CS)
            (s'))))
      (state(read_data(pm_phy, pdbrr_data_type)(PDBR)
            (state(read_data(pm_phy, seg-
segment_reg_data_type)
            (CS)
            (s'))))))),
      xlat_ofs(0,
      data(translate(0,
      data(read_data(pm_phy, pdbrr_data_type)(PDBR)
            (state(read_data
            (pm_phy, seg-
segment_reg_data_type)
            (CS)
            (s')))) 'base_addr,
      addr', ac',
      segment_to_priv(data(read_data(pm_phy,
segment_reg_data_type)
            (CS)
            (s'))))
      (state(read_data(pm_phy, pdbrr_data_type)(PDBR)
```

C.117 Proofs for Linear_Memory_Properties (challenge-linear.pvs)

Using lemma `translate_same_page_address_ok`,

Using lemma `translate_same_page_address_result`,

Expanding the definition of `linear_resolve`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `linear_resolve_same_page_address_ok.1`.

linear_resolve_same_page_address_ok.2:

```

{-1} Mem?(addr' 'type_of)
{-2} 0 ≤ addr' 'offset
{-3} addr' 'offset < max_linear_offset
{-4} delta' ≥ 0
{-5} rem(expt(2, min_page))(offset(addr')) + delta' < expt(2, min_page)
{-6} OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s'))
{-7} OK?(read_data(pm_phy, pdb_r_data_type)(PDBR)
      (state(read_data(pm_phy, segment_reg_data_type)(CS)(s'))))
{-8} OK?(translate(0,
      data(read_data(pm_phy, pdb_r_data_type)(PDBR)
            (state(read_data(pm_phy, seg-
segment_reg_data_type)(CS)
            (s')))) 'base_addr,
      addr', ac',
      segment_to_priv(data(read_data(pm_phy, seg-
segment_reg_data_type)(CS)(s'))))
      (state(read_data(pm_phy, pdb_r_data_type)(PDBR)
            (state(read_data(pm_phy, seg-
segment_reg_data_type)(CS)
            (s'))))))))
{-9} OK?(translate(1,
      data(translate(0,
            data(read_data(pm_phy, pdb_r_data_type)(PDBR)
                  (state(read_data(pm_phy, seg-
segment_reg_data_type)
                  (CS)
                  (s')))) 'base_addr,
            addr', ac',
            segment_to_priv(data(read_data(pm_phy, seg-
segment_reg_data_type)
            (CS)
            (s'))))
            (state(read_data(pm_phy, pdb_r_data_type)(PDBR)
                  (state(read_data(pm_phy, seg-
segment_reg_data_type)
            (CS)
            (s')))))))) '2,
      addr', ac',
      segment_to_priv(data(read_data(pm_phy, seg-
segment_reg_data_type)(CS)(s'))))
      (state(translate(0,
            data(read_data(pm_phy, pdb_r_data_type)(PDBR)
                  (state(read_data(pm_phy, seg-
segment_reg_data_type)
            (CS)
            (s')))) 'base_addr
            addr', ac',
            segment_to_priv(data(read_data(pm_phy, seg-
segment_reg_data_type)
            (CS)
            (s'))))
            (state(read_data(pm_phy, pdb_r_data_type)(PDBR)
                  (state(read_data(pm_phy, seg-
segment_reg_data_type)
            (CS)
            (s'))))))))
2046 {-10} OK?(OK(state(translate(1,
      data(translate(0,
            data(read_data(pm_phy, pdb_r_data_type)(P
            (state
            (read_data
            (pm_phy, seg-

```

C.117 Proofs for Linear_Memory_Properties (challenge-linear.pvs)

Using lemma `translate_same_page_address_result`,

Using lemma `translate_same_page_address_ok`,

Expanding the definition of `linear_resolve`,

Case splitting on $0 \leq (\text{addr!1} + \text{delta!1})\text{'offset}$, $(\text{addr!1} + \text{delta!1})\text{'offset} < \text{max_linear_offset}$,

we get 3 subgoals:

linear_resolve_same_page_address_ok.2.1:

```

{-1} (addr' + delta)'offset < max_linear_offset
{-2} 0 ≤ (addr' + delta)'offset
{-3} rem(expt(2, min_page))(offset(addr')) + delta' < expt(2, min_page) ∧
      OK?(translate(0,
                    data(read_data(pm_phy, pabr_data_type)(PABR)
                        (state(read_data(pm_phy, seg-
segment_reg_data_type)(CS)
                                (s'))))'base_addr,
                                addr', ac',
                                segment_to_priv(data(read_data(pm_phy, seg-
segment_reg_data_type)(CS)
                                (s'))))
                                (state(read_data(pm_phy, pabr_data_type)(PABR)
                                (state(read_data(pm_phy, seg-
segment_reg_data_type)(CS)
                                (s'))))))))
      ⊃
      OK?(translate(0,
                    data(read_data(pm_phy, pabr_data_type)(PABR)
                        (state(read_data(pm_phy, seg-
segment_reg_data_type)(CS)
                                (s'))))'base_addr,
                                addr' + delta', ac',
                                segment_to_priv(data(read_data(pm_phy, seg-
segment_reg_data_type)(CS)
                                (s'))))
                                (state(read_data(pm_phy, pabr_data_type)(PABR)
                                (state(read_data(pm_phy, seg-
segment_reg_data_type)(CS)
                                (s'))))))))
      ⊃
      OK?(translate(0,
                    data(read_data(pm_phy, pabr_data_type)(PABR)
                        (state(read_data(pm_phy, seg-
segment_reg_data_type)(CS)
                                (s'))))'base_addr,
                                addr', ac',
                                segment_to_priv(data(read_data(pm_phy, seg-
segment_reg_data_type)(CS)
                                (s'))))
                                (state(read_data(pm_phy, pabr_data_type)(PABR)
                                (state(read_data(pm_phy, seg-
segment_reg_data_type)(CS)
                                (s'))))))))
{-4} rem(expt(2, min_page))(offset(addr')) + delta' < expt(2, min_page) ∧
      OK?(translate(0,
                    data(read_data(pm_phy, pabr_data_type)(PABR)
                        (state(read_data(pm_phy, seg-
segment_reg_data_type)(CS)
                                (s'))))'base_addr,
                                addr', ac',
                                segment_to_priv(data(read_data(pm_phy, seg-
segment_reg_data_type)(CS)
                                (s'))))
                                (state(read_data(pm_phy, pabr_data_type)(PABR)
                                (state(read_data(pm_phy, seg-
segment_reg_data_type)(CS)
                                (s'))))))))
      ⊃
      translate(0,
                data(read_data(pm_phy, pabr_data_type)(PABR)
                    (state(read_data(pm_phy, segment_reg_data_type)(CS)
                            (s'))))'base_addr,
                    addr', ac',
                    segment_to_priv(data(read_data(pm_phy, seg-
segment_reg_data_type)(CS)(s'))))
                    (state(read_data(pm_phy, pabr_data_type)(PABR)
                            (state(read_data(pm_phy, segment_reg_data_type)(CS)
                                    (s'))))))))
      =
      translate(0,
                data(read_data(pm_phy, pabr_data_type)(PABR)
                    (state(read_data(pm_phy, segment_reg_data_type)(CS)
                            (s'))))'base_addr,
                    addr' + delta', ac',
                    segment_to_priv(data(read_data(pm_phy, seg-
segment_reg_data_type)(CS)(s'))))
                    (state(read_data(pm_phy, pabr_data_type)(PABR)
                            (state(read_data(pm_phy, segment_reg_data_type)(CS)
                                    (s'))))))))

```

Case splitting on $OK?(translate(1, data(translate(0, data(read_data (pm_phy, pabr_data_type) (PDBR) (state (read_data (pm_phy, segment_reg_data_type) (CS) (s!1))))'base_addr, addr!1, ac!1, segment_to_priv(data (read_data (pm_phy, segment_reg_data_type) (CS) (s!1)))) (state(read_data (pm_phy, pabr_data_type) (PDBR) (state (read_data (pm_phy, segment_reg_data_type) (CS) (s!1))))))'2, addr!1, ac!1, segment_to_priv(data(read_data (pm_phy, segment_reg_data_type) (CS) (s!1)))) (state(translate(0, data(read_data (pm_phy, pabr_data_type) (PDBR) (state (read_data (pm_phy, segment_reg_data_type) (CS) (s!1))))'base_addr, addr!1, ac!1, segment_to_priv(data (read_data (pm_phy, segment_reg_data_type) (CS) (s!1)))) (state(read_data (pm_phy, pabr_data_type) (PDBR) (state (read_data (pm_phy, segment_reg_data_type) (CS) (s!1))))))) = OK?(translate(1, data(translate(0, data(read_data (pm_phy, pabr_data_type) (PDBR) (state (read_data (pm_phy, segment_reg_data_type) (CS) (s!1))))'base_addr, addr!1 + delta!1, ac!1, segment_to_priv(data (read_data (pm_phy, segment_reg_data_type) (CS) (s!1)))) (state(read_data (pm_phy, pabr_data_type) (PDBR) (state (read_data (pm_phy, segment_reg_data_type) (CS) (s!1))))))'2, addr!1 + delta!1, ac!1, segment_to_priv(data(read_data (pm_phy, segment_reg_data_type) (CS) (s!1)))) (state(translate(0, data(read_data (pm_phy, pabr_data_type) (PDBR) (state (read_data (pm_phy, segment_reg_data_type) (CS) (s!1))))'base_addr, addr!1 + delta!1, ac!1, segment_to_priv(data (read_data (pm_phy, segment_reg_data_type) (CS) (s!1)))) (state(read_data (pm_phy, pabr_data_type) (PDBR) (state (read_data (pm_phy, segment_reg_data_type) (CS) (s!1)))))))))$,

we get 7 subgoals:

linear_resolve_same_page_address_ok.2.1.1:

```

{-1} OK?(translate(1,
                                data(translate(0,
                                                data(read_data(pm_phy, pabr_data_type)(PABR)
                                                    (state(read_data(pm_phy, seg-
ment_reg_data_type)
                                                    (CS)
                                                    (s'))))'base_addr,
                                addr', ac',
                                segment_to_priv(data(read_data(pm_phy, seg-
ment_reg_data_type)
                                                    (CS)
                                                    (s'))))
                                                (state(read_data(pm_phy, pabr_data_type)(PABR)
                                                    (state(read_data(pm_phy, seg-
ment_reg_data_type)
                                                    (CS)
                                                    (s'))))))'2,
                                addr', ac',
                                segment_to_priv(data(read_data(pm_phy, seg-
ment_reg_data_type)(CS)(s'))))
                                                (state(translate(0,
                                                data(read_data(pm_phy, pabr_data_type)(PABR)
                                                    (state(read_data(pm_phy, seg-
ment_reg_data_type)
                                                    (CS)
                                                    (s'))))'base_addr
                                addr', ac',
                                segment_to_priv(data(read_data(pm_phy, seg-
ment_reg_data_type)
                                                    (CS)
                                                    (s'))))
                                                (state(read_data(pm_phy, pabr_data_type)(PABR)
                                                    (state(read_data(pm_phy,
segment_reg_data_type)
                                                    (CS)
                                                    (s'))))))))
=
OK?(translate(1,
                                data(translate(0,
                                                data(read_data(pm_phy, pabr_data_type)(PABR)
                                                    (state(read_data(pm_phy, seg-
ment_reg_data_type)
                                                    (CS)
                                                    (s'))))'base_addr
                                addr' + delta', ac',
                                segment_to_priv(data(read_data(pm_phy, seg-
ment_reg_data_type)
                                                    (CS)
                                                    (s'))))
                                                (state(read_data(pm_phy, pabr_data_type)(PABR)
                                                    (state(read_data(pm_phy,
segment_reg_data_type)
                                                    (CS)
                                                    (s'))))))'2,
                                addr' + delta', ac',
                                segment_to_priv(data(read_data(pm_phy, seg-
ment_reg_data_type)(CS)
                                                    (s'))))
                                                (state(translate(0,
                                                data(read_data(pm_phy, pabr_data_type)(PABR)
                                                    (state(read_data(pm_phy,

```

2050

C.117 Proofs for Linear_Memory_Properties (challenge-linear.pvs)

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `linear_resolve_same_page_address_ok.2.1.1`.

linear_resolve_same_page_address_ok.2.1.2:

```

{-1} (addr' + delta)'offset < max_linear_offset
{-2} 0 ≤ (addr' + delta)'offset
{-3} rem(expt(2, min_page))(offset(addr')) + delta' < expt(2, min_page) ∧
      OK?(translate(0,
                    data(read_data(pm_phy, pabr_data_type)(PABR)
                        (state(read_data(pm_phy, seg-
segment_reg_data_type)(CS)
                                (s'))))'base_addr,
                                addr', ac',
                                segment_to_priv(data(read_data(pm_phy, seg-
segment_reg_data_type)(CS)
                                (s'))))
                                (state(read_data(pm_phy, pabr_data_type)(PABR)
                                (state(read_data(pm_phy, seg-
segment_reg_data_type)(CS)
                                (s'))))))))
      ⊃
      OK?(translate(0,
                    data(read_data(pm_phy, pabr_data_type)(PABR)
                        (state(read_data(pm_phy, seg-
segment_reg_data_type)(CS)
                                (s'))))'base_addr,
                                addr' + delta', ac',
                                segment_to_priv(data(read_data(pm_phy, seg-
segment_reg_data_type)(CS)
                                (s'))))
                                (state(read_data(pm_phy, pabr_data_type)(PABR)
                                (state(read_data(pm_phy, seg-
segment_reg_data_type)(CS)
                                (s'))))))))
      ⊃
      OK?(translate(0,
                    data(read_data(pm_phy, pabr_data_type)(PABR)
                        (state(read_data(pm_phy, seg-
segment_reg_data_type)(CS)
                                (s'))))'base_addr,
                                addr', ac',
                                segment_to_priv(data(read_data(pm_phy, seg-
segment_reg_data_type)(CS)
                                (s'))))
                                (state(read_data(pm_phy, pabr_data_type)(PABR)
                                (state(read_data(pm_phy, seg-
segment_reg_data_type)(CS)
                                (s'))))))))
{-4} rem(expt(2, min_page))(offset(addr')) + delta' < expt(2, min_page) ∧
      OK?(translate(0,
                    data(read_data(pm_phy, pabr_data_type)(PABR)
                        (state(read_data(pm_phy, seg-
segment_reg_data_type)(CS)
                                (s'))))'base_addr,
                                addr', ac',
                                segment_to_priv(data(read_data(pm_phy, seg-
segment_reg_data_type)(CS)
                                (s'))))
                                (state(read_data(pm_phy, pabr_data_type)(PABR)
                                (state(read_data(pm_phy, seg-
segment_reg_data_type)(CS)
                                (s'))))))))
      ⊃
      translate(0,
                data(read_data(pm_phy, pabr_data_type)(PABR)
                    (state(read_data(pm_phy, segment_reg_data_type)(CS)
                            (s'))))'base_addr,
                    addr', ac',
                    segment_to_priv(data(read_data(pm_phy, seg-
segment_reg_data_type)(CS)(s'))))
                    (state(read_data(pm_phy, pabr_data_type)(PABR)
                            (state(read_data(pm_phy, segment_reg_data_type)(CS)
                                    (s'))))))))
      =
      translate(0,
                data(read_data(pm_phy, pabr_data_type)(PABR)
                    (state(read_data(pm_phy, segment_reg_data_type)(CS)
                            (s'))))'base_addr,
                    addr' + delta', ac',
                    segment_to_priv(data(read_data(pm_phy, seg-
segment_reg_data_type)(CS)(s'))))
                    (state(read_data(pm_phy, pabr_data_type)(PABR)
                            (state(read_data(pm_phy, segment_reg_data_type)(CS)
                                    (s'))))))))

```

Hiding formulas: 3,

Using lemma `translate_same_page_address_ok`,

we get 2 subgoals:

linear_resolve_same_page_address_ok.2.1.2.1:

{-1} rem(expt(2, min_page))(offset(addr') + delta' < expt(2, min_page) ^
 OK?(translate(1,
 data(translate(0,
 data(read_data(pm_phy, pdb_data_type)(PDBR
 (state(read_data(pm_phy, seg-
 ment_reg_data_type)
 (CS
 (s')))) 'base_addr
 addr' + delta', ac',
 segment_to_priv(data(read_data(pm_phy, seg-
 ment_reg_data_type)
 (CS
 (s'))))
 (state(read_data(pm_phy, pdb_data_type)(PDBR
 (state(read_data(pm_phy,
 seg-
 ment_reg_data_type)
 (CS
 (s'))))))) '2,
 addr', ac',
 segment_to_priv(data(read_data(pm_phy, seg-
 ment_reg_data_type)(CS
 (s'))))
 (state(translate(0,
 data(read_data(pm_phy, pdb_data_type)(PDBR
 (state(read_data(pm_phy,
 seg-
 ment_reg_data_type)
 (CS
 (s')))) 'base_addr
 addr' + delta', ac',
 segment_to_priv(data(read_data(pm_phy, seg-
 ment_reg_data_type)
 (CS
 (s'))))
 (state(read_data(pm_phy, pdb_data_type)(PDBR
 (state(read_data
 (pm_phy, seg-
 ment_reg_data_type)
 (CS
 (s')))))))
 ⊃
 OK?(translate(1,
 data(translate(0,
 data(read_data(pm_phy, pdb_data_type)(PDBR
 (state(read_data(pm_phy, seg-
 ment_reg_data_type)
 (CS
 (s')))) 'base_addr
 addr' + delta', ac',
 segment_to_priv(data(read_data(pm_phy, seg-
 ment_reg_data_type)
 (CS
 (s'))))
 (state(read_data(pm_phy, pdb_data_type)(PDBR
 (state(read_data(pm_phy,
 seg-
 ment_reg_data_type)
 (CS
 (s'))))))) '2,
 addr' + delta', ac',
 segment_to_priv(data(read_data(pm_phy, seg-
 ment_reg_data_type)(CS

C.117 Proofs for Linear_Memory_Properties (challenge-linear.pvs)

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Replacing using formula -7,

which is trivially true.

This completes the proof of `linear_resolve_same_page_address_ok.2.1.2.1`.

linear_resolve_same_page_address_ok.2.1.2.2:

```

{-1} (addr' + delta)'offset < max_linear_offset
{-2} 0 ≤ (addr' + delta)'offset
{-3} rem(expt(2, min_page))(offset(addr')) + delta' < expt(2, min_page) ∧
      OK?(translate(0,
                    data(read_data(pm_phy, pabr_data_type)(PABR)
                        (state(read_data(pm_phy, seg-
segment_reg_data_type)(CS)
                                (s'))))'base_addr,
                                addr', ac',
                                segment_to_priv(data(read_data(pm_phy, seg-
segment_reg_data_type)(CS)
                                (s'))))
                                (state(read_data(pm_phy, pabr_data_type)(PABR)
                                (state(read_data(pm_phy, seg-
segment_reg_data_type)(CS)
                                (s'))))))))
      ⊃
      OK?(translate(0,
                    data(read_data(pm_phy, pabr_data_type)(PABR)
                        (state(read_data(pm_phy, seg-
segment_reg_data_type)(CS)
                                (s'))))'base_addr,
                                addr' + delta', ac',
                                segment_to_priv(data(read_data(pm_phy, seg-
segment_reg_data_type)(CS)
                                (s'))))
                                (state(read_data(pm_phy, pabr_data_type)(PABR)
                                (state(read_data(pm_phy, seg-
segment_reg_data_type)(CS)
                                (s'))))))))
      ⊃
      OK?(translate(0,
                    data(read_data(pm_phy, pabr_data_type)(PABR)
                        (state(read_data(pm_phy, seg-
segment_reg_data_type)(CS)
                                (s'))))'base_addr,
                                addr', ac',
                                segment_to_priv(data(read_data(pm_phy, seg-
segment_reg_data_type)(CS)
                                (s'))))
                                (state(read_data(pm_phy, pabr_data_type)(PABR)
                                (state(read_data(pm_phy, seg-
segment_reg_data_type)(CS)
                                (s'))))))))
      ⊃
      translate(0,
                data(read_data(pm_phy, pabr_data_type)(PABR)
                    (state(read_data(pm_phy, segment_reg_data_type)(CS)
                            (s'))))'base_addr,
                    addr', ac',
                    segment_to_priv(data(read_data(pm_phy, seg-
segment_reg_data_type)(CS)(s'))))
                    (state(read_data(pm_phy, pabr_data_type)(PABR)
                            (state(read_data(pm_phy, segment_reg_data_type)(CS)
                                    (s'))))))))
      =
      translate(0,
                data(read_data(pm_phy, pabr_data_type)(PABR)
                    (state(read_data(pm_phy, segment_reg_data_type)(CS)
                            (s'))))'base_addr,
                    addr' + delta', ac',
                    segment_to_priv(data(read_data(pm_phy, seg-
segment_reg_data_type)(CS)(s'))))
                    (state(read_data(pm_phy, pabr_data_type)(PABR)
                            (state(read_data(pm_phy, segment_reg_data_type)(CS)
                                    (s'))))))))

```

Hiding formulas: (-13 -14 2),

Using lemma `translate_result_no_leaf`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Replacing using formula -7,

which is trivially true.

This completes the proof of `linear_resolve_same_page_address_ok.2.1.2.2`.

linear_resolve_same_page_address_ok.2.1.3:

```

{-1} (addr' + delta)'offset < max_linear_offset
{-2} 0 ≤ (addr' + delta)'offset
{-3} rem(expt(2, min_page))(offset(addr')) + delta' < expt(2, min_page) ∧
      OK?(translate(0,
                    data(read_data(pm_phy, pabr_data_type)(PABR)
                        (state(read_data(pm_phy, seg-
segment_reg_data_type)(CS)
                                (s'))))'base_addr,
                                addr', ac',
                                segment_to_priv(data(read_data(pm_phy, seg-
segment_reg_data_type)(CS)
                                (s'))))
                                (state(read_data(pm_phy, pabr_data_type)(PABR)
                                (state(read_data(pm_phy, seg-
segment_reg_data_type)(CS)
                                (s'))))))))
      ⊃
      OK?(translate(0,
                    data(read_data(pm_phy, pabr_data_type)(PABR)
                        (state(read_data(pm_phy, seg-
segment_reg_data_type)(CS)
                                (s'))))'base_addr,
                                addr' + delta', ac',
                                segment_to_priv(data(read_data(pm_phy, seg-
segment_reg_data_type)(CS)
                                (s'))))
                                (state(read_data(pm_phy, pabr_data_type)(PABR)
                                (state(read_data(pm_phy, seg-
segment_reg_data_type)(CS)
                                (s'))))))))
      ⊃
      OK?(translate(0,
                    data(read_data(pm_phy, pabr_data_type)(PABR)
                        (state(read_data(pm_phy, seg-
segment_reg_data_type)(CS)
                                (s'))))'base_addr,
                                addr', ac',
                                segment_to_priv(data(read_data(pm_phy, seg-
segment_reg_data_type)(CS)
                                (s'))))
                                (state(read_data(pm_phy, pabr_data_type)(PABR)
                                (state(read_data(pm_phy, seg-
segment_reg_data_type)(CS)
                                (s'))))))))
      ⊃
      translate(0,
                data(read_data(pm_phy, pabr_data_type)(PABR)
                    (state(read_data(pm_phy, segment_reg_data_type)(CS)
                            (s'))))'base_addr,
                    addr', ac',
                    segment_to_priv(data(read_data(pm_phy, seg-
segment_reg_data_type)(CS)(s'))))
                    (state(read_data(pm_phy, pabr_data_type)(PABR)
                            (state(read_data(pm_phy, segment_reg_data_type)(CS)
                                    (s'))))))))
      =
      translate(0,
                data(read_data(pm_phy, pabr_data_type)(PABR)
                    (state(read_data(pm_phy, segment_reg_data_type)(CS)
                            (s'))))'base_addr,
                    addr' + delta', ac',
                    segment_to_priv(data(read_data(pm_phy, seg-
segment_reg_data_type)(CS)(s'))))
                    (state(read_data(pm_phy, pabr_data_type)(PABR)
                            (state(read_data(pm_phy, segment_reg_data_type)(CS)
                                    (s'))))))))

```

C.117 Proofs for Linear_Memory_Properties (challenge-linear.pvs)

Using lemma `translate_memory_address`,

we get 2 subgoals:

linear_resolve_same_page_address_ok.2.1.3.1:

```

{-1}  aligned?(bits_per_level + pe_size)
      (offset
       (data[Physical_memory, Pdbr_type]
        (read_data[Physical_memory, Pdbr_type]
         (pm_phy, pdbr_data_type)(PDBR)
         (state[Physical_memory, Segment_Reg_type]
          (read_data[Physical_memory, Segment_Reg_type]
           (pm_phy, segment_reg_data_type)(CS)
           (s'))))'base_addr))

      ^
      OK?(translate(0,
                   data[Physical_memory, Pdbr_type]
                   (read_data[Physical_memory, Pdbr_type]
                    (pm_phy, pdbr_data_type)(PDBR)
                    (state[Physical_memory, Segment_Reg_type]
                     (read_data[Physical_memory, Segment_Reg_type]
                      (pm_phy, segment_reg_data_type)(CS)
                      (s'))))'base_addr,
                   addr' + delta', ac',
                   segment_to_priv(data[Physical_memory, Segment_Reg_type]
                                   (read_data[Physical_memory, Segment_Reg_type]
                                    (pm_phy, segment_reg_data_type)(CS)(s'))))
                                   (state[Physical_memory, Pdbr_type]
                                    (read_data[Physical_memory, Pdbr_type]
                                     (pm_phy, pdbr_data_type)(PDBR)
                                     (state[Physical_memory, Segment_Reg_type]
                                      (read_data[Physical_memory, Segment_Reg_type]
                                       (pm_phy, segment_reg_data_type)(CS)(s'))))))))
      ⊃
      data(translate(0,
                    data[Physical_memory, Pdbr_type]
                    (read_data[Physical_memory, Pdbr_type]
                     (pm_phy, pdbr_data_type)(PDBR)
                     (state[Physical_memory, Segment_Reg_type]
                      (read_data[Physical_memory, Segment_Reg_type]
                       (pm_phy, segment_reg_data_type)(CS)
                       (s'))))'base_addr,
                    addr' + delta', ac',
                    segment_to_priv(data[Physical_memory, Segment_Reg_type]
                                    (read_data[Physical_memory, Segment_Reg_type]
                                     (pm_phy, segment_reg_data_type)(CS)(s'))))
                                    (state[Physical_memory, Pdbr_type]
                                     (read_data[Physical_memory, Pdbr_type]
                                      (pm_phy, pdbr_data_type)(PDBR)
                                      (state[Physical_memory, Segment_Reg_type]
                                       (read_data[Physical_memory, Segment_Reg_type]
                                        (pm_phy, segment_reg_data_type)(CS)
                                        (s'))))))))'2
                    < max_linear
                    (addr' + delta)'offset < max_linear_offset
{-2}  0 ≤ (addr' + delta)'offset
{-3}  0 ≤ (addr' + delta)'offset
{-4}  rem(expt(2, min_page))(offset(addr')) + delta' < expt(2, min_page) ∧
      OK?(translate(0,
                    data(read_data(pm_phy, pdbr_data_type)(PDBR)
                          (state(read_data(pm_phy, seg-

```

C.117 Proofs for Linear_Memory_Properties (challenge-linear.pvs)

Using lemma `translate_memory_address2`,

we get 2 subgoals:

linear_resolve_same_page_address_ok.2.1.3.1.1:

```

{-1}  aligned?(bits_per_level + pe_size)
      (offset
        (data[Physical_memory, Pdbr_type]
          (read_data[Physical_memory, Pdbr_type]
            (pm_phy, pdbr_data_type)(PDBR)
            (state[Physical_memory, Segment_Reg_type]
              (read_data[Physical_memory, Segment_Reg_type]
                (pm_phy, segment_reg_data_type)(CS)
                (s')))) 'base_addr))

      ^
      OK?(translate(0,
        data[Physical_memory, Pdbr_type]
          (read_data[Physical_memory, Pdbr_type]
            (pm_phy, pdbr_data_type)(PDBR)
            (state[Physical_memory, Segment_Reg_type]
              (read_data[Physical_memory, Segment_Reg_type]
                (pm_phy, segment_reg_data_type)(CS)
                (s')))) 'base_addr,
        addr' + delta', ac',
        segment_to_priv(data[Physical_memory, Segment_Reg_type]
          (read_data[Physical_memory, Segment_Reg_type]
            (pm_phy, segment_reg_data_type)(CS)(s'))))
          (state[Physical_memory, Pdbr_type]
            (read_data[Physical_memory, Pdbr_type]
              (pm_phy, pdbr_data_type)(PDBR)
              (state[Physical_memory, Segment_Reg_type]
                (read_data[Physical_memory, Segment_Reg_type]
                  (pm_phy, segment_reg_data_type)(CS)(s'))))))))
      ⊃
      0 ≤
      offset
      (data(translate(0,
        data[Physical_memory, Pdbr_type]
          (read_data[Physical_memory, Pdbr_type]
            (pm_phy, pdbr_data_type)(PDBR)
            (state[Physical_memory, Segment_Reg_type]
              (read_data[Physical_memory, Segment_Reg_type]
                (pm_phy, segment_reg_data_type)(CS)
                (s')))) 'base_addr,
          addr' + delta', ac',
          segment_to_priv(data[Physical_memory, Segment_Reg_type]
            (read_data[Physical_memory, Segment_Reg_type]
              (pm_phy, segment_reg_data_type)(CS)
              (s'))))
            (state[Physical_memory, Pdbr_type]
              (read_data[Physical_memory, Pdbr_type]
                (pm_phy, pdbr_data_type)(PDBR)
                (state[Physical_memory, Segment_Reg_type]
                  (read_data[Physical_memory, Segment_Reg_type]
                    (pm_phy, segment_reg_data_type)(CS)
                    (s')))))))) '2)
      (offset

```

C.117 Proofs for Linear_Memory_Properties (challenge-linear.pvs)

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `linear_resolve_same_page_address_ok.2.1.3.1.1`.

linear_resolve_same_page_address_ok.2.1.3.1.2:

```

{-1}  aligned?(bits_per_level + pe_size)
      (offset
       (data[Physical_memory, Pdbr_type]
        (read_data[Physical_memory, Pdbr_type]
         (pm_phy, pdbr_data_type)(PDBR)
         (state[Physical_memory, Segment_Reg_type]
          (read_data[Physical_memory, Segment_Reg_type]
           (pm_phy, segment_reg_data_type)(CS)
           (s'))))'base_addr))

      ^
      OK?(translate(0,
                   data[Physical_memory, Pdbr_type]
                     (read_data[Physical_memory, Pdbr_type]
                      (pm_phy, pdbr_data_type)(PDBR)
                      (state[Physical_memory, Segment_Reg_type]
                       (read_data[Physical_memory, Segment_Reg_type]
                        (pm_phy, segment_reg_data_type)(CS)
                        (s'))))'base_addr,
                   addr' + delta', ac',
                   segment_to_priv(data[Physical_memory, Segment_Reg_type]
                                   (read_data[Physical_memory, Segment_Reg_type]
                                    (pm_phy, segment_reg_data_type)(CS)(s'))))
                                   (state[Physical_memory, Pdbr_type]
                                    (read_data[Physical_memory, Pdbr_type]
                                     (pm_phy, pdbr_data_type)(PDBR)
                                     (state[Physical_memory, Segment_Reg_type]
                                      (read_data[Physical_memory, Segment_Reg_type]
                                       (pm_phy, segment_reg_data_type)(CS)(s'))))))))
      ⊃
      data(translate(0,
                    data[Physical_memory, Pdbr_type]
                      (read_data[Physical_memory, Pdbr_type]
                       (pm_phy, pdbr_data_type)(PDBR)
                       (state[Physical_memory, Segment_Reg_type]
                        (read_data[Physical_memory, Segment_Reg_type]
                         (pm_phy, segment_reg_data_type)(CS)
                         (s'))))'base_addr,
                    addr' + delta', ac',
                    segment_to_priv(data[Physical_memory, Segment_Reg_type]
                                    (read_data[Physical_memory, Segment_Reg_type]
                                     (pm_phy, segment_reg_data_type)(CS)(s'))))
                                    (state[Physical_memory, Pdbr_type]
                                     (read_data[Physical_memory, Pdbr_type]
                                      (pm_phy, pdbr_data_type)(PDBR)
                                      (state[Physical_memory, Segment_Reg_type]
                                       (read_data[Physical_memory, Segment_Reg_type]
                                        (pm_phy, segment_reg_data_type)(CS)
                                        (s'))))))))'2
                    < max_linear
                    (addr' + delta)'offset < max_linear_offset
{-2}  0 ≤ (addr' + delta)'offset
{-3}  0 ≤ (addr' + delta)'offset
{-4}  rem(expt(2, min_page))(offset(addr')) + delta' < expt(2, min_page) ∧
      OK?(translate(0,
                    data(read_data(pm_phy, pdbr_data_type)(PDBR)
                          (state(read_data(pm_phy, seg-

```

C.117 Proofs for Linear_Memory_Properties (challenge-linear.pvs)

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `linear_resolve_same_page_address_ok.2.1.3.1.2`.

linear_resolve_same_page_address_ok.2.1.3.2:

```

{-1} (addr' + delta)'offset < max_linear_offset
{-2} 0 ≤ (addr' + delta)'offset
{-3} rem(expt(2, min_page))(offset(addr')) + delta' < expt(2, min_page) ∧
      OK?(translate(0,
                    data(read_data(pm_phy, pabr_data_type)(PABR)
                        (state(read_data(pm_phy, seg-
segment_reg_data_type)(CS)
                                (s'))))'base_addr,
                                addr', ac',
                                segment_to_priv(data(read_data(pm_phy, seg-
segment_reg_data_type)(CS)
                                                (s'))))
                                (state(read_data(pm_phy, pabr_data_type)(PABR)
                                    (state(read_data(pm_phy, seg-
segment_reg_data_type)(CS)
                                                (s'))))))))
      ⊃
      OK?(translate(0,
                    data(read_data(pm_phy, pabr_data_type)(PABR)
                        (state(read_data(pm_phy, seg-
segment_reg_data_type)(CS)
                                (s'))))'base_addr,
                                addr' + delta', ac',
                                segment_to_priv(data(read_data(pm_phy, seg-
segment_reg_data_type)(CS)
                                                (s'))))
                                (state(read_data(pm_phy, pabr_data_type)(PABR)
                                    (state(read_data(pm_phy, seg-
segment_reg_data_type)(CS)
                                                (s'))))))))
      ⊃
      OK?(translate(0,
                    data(read_data(pm_phy, pabr_data_type)(PABR)
                        (state(read_data(pm_phy, seg-
segment_reg_data_type)(CS)
                                (s'))))'base_addr,
                                addr', ac',
                                segment_to_priv(data(read_data(pm_phy, seg-
segment_reg_data_type)(CS)
                                                (s'))))
                                (state(read_data(pm_phy, pabr_data_type)(PABR)
                                    (state(read_data(pm_phy, seg-
segment_reg_data_type)(CS)
                                                (s'))))))))
      ⊃
      translate(0,
                data(read_data(pm_phy, pabr_data_type)(PABR)
                    (state(read_data(pm_phy, segment_reg_data_type)(CS)
                                (s'))))'base_addr,
                    addr', ac',
                    segment_to_priv(data(read_data(pm_phy, seg-
segment_reg_data_type)(CS)(s'))))
                    (state(read_data(pm_phy, pabr_data_type)(PABR)
                                (state(read_data(pm_phy, segment_reg_data_type)(CS)
                                            (s'))))))))
      =
      translate(0,
                data(read_data(pm_phy, pabr_data_type)(PABR)
                    (state(read_data(pm_phy, segment_reg_data_type)(CS)
                                (s'))))'base_addr,
                    addr' + delta', ac',
                    segment_to_priv(data(read_data(pm_phy, seg-
segment_reg_data_type)(CS)(s'))))
                    (state(read_data(pm_phy, pabr_data_type)(PABR)
                                (state(read_data(pm_phy, segment_reg_data_type)(CS)
                                            (s'))))))))

```

C.117 Proofs for Linear_Memory_Properties (challenge-linear.pvs)

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `linear_resolve_same_page_address_ok.2.1.3.2`.

linear_resolve_same_page_address_ok.2.1.4:

```

{-1} (addr' + delta)'offset < max_linear_offset
{-2} 0 ≤ (addr' + delta)'offset
{-3} rem(expt(2, min_page))(offset(addr')) + delta' < expt(2, min_page) ∧
      OK?(translate(0,
                    data(read_data(pm_phy, pabr_data_type)(PABR)
                        (state(read_data(pm_phy, seg-
segment_reg_data_type)(CS)
                                (s'))))'base_addr,
                    addr', ac',
                    segment_to_priv(data(read_data(pm_phy, seg-
segment_reg_data_type)(CS)
                                (s'))))
                    (state(read_data(pm_phy, pabr_data_type)(PABR)
                        (state(read_data(pm_phy, seg-
segment_reg_data_type)(CS)
                                (s'))))))))
      ⊃
      OK?(translate(0,
                    data(read_data(pm_phy, pabr_data_type)(PABR)
                        (state(read_data(pm_phy, seg-
segment_reg_data_type)(CS)
                                (s'))))'base_addr,
                    addr' + delta', ac',
                    segment_to_priv(data(read_data(pm_phy, seg-
segment_reg_data_type)(CS)
                                (s'))))
                    (state(read_data(pm_phy, pabr_data_type)(PABR)
                        (state(read_data(pm_phy, seg-
segment_reg_data_type)(CS)
                                (s'))))))))
      ⊃
      OK?(translate(0,
                    data(read_data(pm_phy, pabr_data_type)(PABR)
                        (state(read_data(pm_phy, seg-
segment_reg_data_type)(CS)
                                (s'))))'base_addr,
                    addr', ac',
                    segment_to_priv(data(read_data(pm_phy, seg-
segment_reg_data_type)(CS)
                                (s'))))
                    (state(read_data(pm_phy, pabr_data_type)(PABR)
                        (state(read_data(pm_phy, seg-
segment_reg_data_type)(CS)
                                (s'))))))))
      ⊃
      translate(0,
                data(read_data(pm_phy, pabr_data_type)(PABR)
                    (state(read_data(pm_phy, segment_reg_data_type)(CS)
                            (s'))))'base_addr,
                addr', ac',
                segment_to_priv(data(read_data(pm_phy, seg-
segment_reg_data_type)(CS)(s'))))
                (state(read_data(pm_phy, pabr_data_type)(PABR)
                    (state(read_data(pm_phy, segment_reg_data_type)(CS)
                            (s'))))))))
      =
      translate(0,
                data(read_data(pm_phy, pabr_data_type)(PABR)
                    (state(read_data(pm_phy, segment_reg_data_type)(CS)
                            (s'))))'base_addr,
                addr' + delta', ac',
                segment_to_priv(data(read_data(pm_phy, seg-
segment_reg_data_type)(CS)(s'))))
                (state(read_data(pm_phy, pabr_data_type)(PABR)
                    (state(read_data(pm_phy, segment_reg_data_type)(CS)
                            (s'))))))))

```

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `linear_resolve_same_page_address_ok.2.1.4`.

linear_resolve_same_page_address_ok.2.1.5:

```

{-1} (addr' + delta')'offset < max_linear_offset
{-2} 0 ≤ (addr' + delta')'offset
{-3} rem(expt(2, min_page))(offset(addr')) + delta' < expt(2, min_page) ∧
      OK?(translate(0,
                    data(read_data(pm_phy, pabr_data_type)(PABR)
                        (state(read_data(pm_phy, seg-
segment_reg_data_type)(CS)
                                (s'))))'base_addr,
                                addr', ac',
                                segment_to_priv(data(read_data(pm_phy, seg-
segment_reg_data_type)(CS)
                                (s'))))
                                (state(read_data(pm_phy, pabr_data_type)(PABR)
                                (state(read_data(pm_phy, seg-
segment_reg_data_type)(CS)
                                (s'))))))))
      ⊃
      OK?(translate(0,
                    data(read_data(pm_phy, pabr_data_type)(PABR)
                        (state(read_data(pm_phy, seg-
segment_reg_data_type)(CS)
                                (s'))))'base_addr,
                                addr' + delta', ac',
                                segment_to_priv(data(read_data(pm_phy, seg-
segment_reg_data_type)(CS)
                                (s'))))
                                (state(read_data(pm_phy, pabr_data_type)(PABR)
                                (state(read_data(pm_phy, seg-
segment_reg_data_type)(CS)
                                (s'))))))))
      ⊃
      rem(expt(2, min_page))(offset(addr')) + delta' < expt(2, min_page) ∧
      OK?(translate(0,
                    data(read_data(pm_phy, pabr_data_type)(PABR)
                        (state(read_data(pm_phy, seg-
segment_reg_data_type)(CS)
                                (s'))))'base_addr,
                                addr', ac',
                                segment_to_priv(data(read_data(pm_phy, seg-
segment_reg_data_type)(CS)
                                (s'))))
                                (state(read_data(pm_phy, pabr_data_type)(PABR)
                                (state(read_data(pm_phy, seg-
segment_reg_data_type)(CS)
                                (s'))))))))
      ⊃
      translate(0,
                data(read_data(pm_phy, pabr_data_type)(PABR)
                    (state(read_data(pm_phy, segment_reg_data_type)(CS)
                                (s'))))'base_addr,
                    addr', ac',
                    segment_to_priv(data(read_data(pm_phy, seg-
segment_reg_data_type)(CS)(s'))))
                    (state(read_data(pm_phy, pabr_data_type)(PABR)
                                (state(read_data(pm_phy, segment_reg_data_type)(CS)
                                (s'))))))))
      =
      translate(0,
                data(read_data(pm_phy, pabr_data_type)(PABR)
                    (state(read_data(pm_phy, segment_reg_data_type)(CS)
                                (s'))))'base_addr,
                    addr' + delta', ac',
                    segment_to_priv(data(read_data(pm_phy, seg-
segment_reg_data_type)(CS)(s'))))
                    (state(read_data(pm_phy, pabr_data_type)(PABR)
                                (state(read_data(pm_phy, segment_reg_data_type)(CS)
                                (s'))))))))

```

C.117 Proofs for Linear_Memory_Properties (challenge-linear.pvs)

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `linear_resolve_same_page_address_ok.2.1.5`.

linear_resolve_same_page_address_ok.2.1.6:

```

{-1} (addr' + delta)'offset < max_linear_offset
{-2} 0 ≤ (addr' + delta)'offset
{-3} rem(expt(2, min_page))(offset(addr')) + delta' < expt(2, min_page) ∧
      OK?(translate(0,
                    data(read_data(pm_phy, pabr_data_type)(PABR)
                        (state(read_data(pm_phy, seg-
segment_reg_data_type)(CS)
                                (s'))))'base_addr,
                                addr', ac',
                                segment_to_priv(data(read_data(pm_phy, seg-
segment_reg_data_type)(CS)
                                (s'))))
                                (state(read_data(pm_phy, pabr_data_type)(PABR)
                                (state(read_data(pm_phy, seg-
segment_reg_data_type)(CS)
                                (s'))))))))
      ⊃
      OK?(translate(0,
                    data(read_data(pm_phy, pabr_data_type)(PABR)
                        (state(read_data(pm_phy, seg-
segment_reg_data_type)(CS)
                                (s'))))'base_addr,
                                addr' + delta', ac',
                                segment_to_priv(data(read_data(pm_phy, seg-
segment_reg_data_type)(CS)
                                (s'))))
                                (state(read_data(pm_phy, pabr_data_type)(PABR)
                                (state(read_data(pm_phy, seg-
segment_reg_data_type)(CS)
                                (s'))))))))
      ⊃
      OK?(translate(0,
                    data(read_data(pm_phy, pabr_data_type)(PABR)
                        (state(read_data(pm_phy, seg-
segment_reg_data_type)(CS)
                                (s'))))'base_addr,
                                addr', ac',
                                segment_to_priv(data(read_data(pm_phy, seg-
segment_reg_data_type)(CS)
                                (s'))))
                                (state(read_data(pm_phy, pabr_data_type)(PABR)
                                (state(read_data(pm_phy, seg-
segment_reg_data_type)(CS)
                                (s'))))))))
{-4} rem(expt(2, min_page))(offset(addr')) + delta' < expt(2, min_page) ∧
      OK?(translate(0,
                    data(read_data(pm_phy, pabr_data_type)(PABR)
                        (state(read_data(pm_phy, seg-
segment_reg_data_type)(CS)
                                (s'))))'base_addr,
                                addr', ac',
                                segment_to_priv(data(read_data(pm_phy, seg-
segment_reg_data_type)(CS)
                                (s'))))
                                (state(read_data(pm_phy, pabr_data_type)(PABR)
                                (state(read_data(pm_phy, seg-
segment_reg_data_type)(CS)
                                (s'))))))))
      ⊃
      translate(0,
                data(read_data(pm_phy, pabr_data_type)(PABR)
                    (state(read_data(pm_phy, segment_reg_data_type)(CS)
                            (s'))))'base_addr,
                    addr', ac',
                    segment_to_priv(data(read_data(pm_phy, seg-
segment_reg_data_type)(CS)(s'))))
                    (state(read_data(pm_phy, pabr_data_type)(PABR)
                            (state(read_data(pm_phy, segment_reg_data_type)(CS)
                                    (s'))))))))
      =
      translate(0,
                data(read_data(pm_phy, pabr_data_type)(PABR)
                    (state(read_data(pm_phy, segment_reg_data_type)(CS)
                            (s'))))'base_addr,
                    addr' + delta', ac',
                    segment_to_priv(data(read_data(pm_phy, seg-
segment_reg_data_type)(CS)(s'))))
                    (state(read_data(pm_phy, pabr_data_type)(PABR)
                            (state(read_data(pm_phy, segment_reg_data_type)(CS)
                                    (s'))))))))

```

C.117 Proofs for Linear_Memory_Properties (challenge-linear.pvs)

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `linear_resolve_same_page_address_ok.2.1.6`.

linear_resolve_same_page_address_ok.2.1.7:

```

{-1} (addr' + delta)'offset < max_linear_offset
{-2} 0 ≤ (addr' + delta)'offset
{-3} rem(expt(2, min_page))(offset(addr')) + delta' < expt(2, min_page) ∧
      OK?(translate(0,
                    data(read_data(pm_phy, pabr_data_type)(PABR)
                        (state(read_data(pm_phy, seg-
segment_reg_data_type)(CS)
                                (s'))))'base_addr,
                                addr', ac',
                                segment_to_priv(data(read_data(pm_phy, seg-
segment_reg_data_type)(CS)
                                (s'))))
                                (state(read_data(pm_phy, pabr_data_type)(PABR)
                                (state(read_data(pm_phy, seg-
segment_reg_data_type)(CS)
                                (s'))))))))
      ⊃
      OK?(translate(0,
                    data(read_data(pm_phy, pabr_data_type)(PABR)
                        (state(read_data(pm_phy, seg-
segment_reg_data_type)(CS)
                                (s'))))'base_addr,
                                addr' + delta', ac',
                                segment_to_priv(data(read_data(pm_phy, seg-
segment_reg_data_type)(CS)
                                (s'))))
                                (state(read_data(pm_phy, pabr_data_type)(PABR)
                                (state(read_data(pm_phy, seg-
segment_reg_data_type)(CS)
                                (s'))))))))
      ⊃
      OK?(translate(0,
                    data(read_data(pm_phy, pabr_data_type)(PABR)
                        (state(read_data(pm_phy, seg-
segment_reg_data_type)(CS)
                                (s'))))'base_addr,
                                addr', ac',
                                segment_to_priv(data(read_data(pm_phy, seg-
segment_reg_data_type)(CS)
                                (s'))))
                                (state(read_data(pm_phy, pabr_data_type)(PABR)
                                (state(read_data(pm_phy, seg-
segment_reg_data_type)(CS)
                                (s'))))))))
      ⊃
      translate(0,
                data(read_data(pm_phy, pabr_data_type)(PABR)
                    (state(read_data(pm_phy, segment_reg_data_type)(CS)
                            (s'))))'base_addr,
                    addr', ac',
                    segment_to_priv(data(read_data(pm_phy, seg-
segment_reg_data_type)(CS)(s'))))
                    (state(read_data(pm_phy, pabr_data_type)(PABR)
                            (state(read_data(pm_phy, segment_reg_data_type)(CS)
                                    (s'))))))))
      =
      translate(0,
                data(read_data(pm_phy, pabr_data_type)(PABR)
                    (state(read_data(pm_phy, segment_reg_data_type)(CS)
                            (s'))))'base_addr,
                    addr' + delta', ac',
                    segment_to_priv(data(read_data(pm_phy, seg-
segment_reg_data_type)(CS)(s'))))
                    (state(read_data(pm_phy, pabr_data_type)(PABR)
                            (state(read_data(pm_phy, segment_reg_data_type)(CS)
                                    (s'))))))))

```

Hiding formulas: (2 3),

Using lemma `translate_memory_address`,

Using lemma `translate_memory_address2`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `linear_resolve_same_page_address_ok.2.1.7`.

linear_resolve_same_page_address_ok.2.2:

```

{-1} 0 ≤ (addr' + delta')'offset
{-2} rem(expt(2, min_page))(offset(addr')) + delta' < expt(2, min_page) ∧
      OK?(translate(0,
                    data(read_data(pm_phy, pdb_data_type)(PDBR)
                        (state(read_data(pm_phy, seg-
ment_reg_data_type)(CS)
                            (s'))))'base_addr,
                    addr', ac',
                    segment_to_priv(data(read_data(pm_phy, seg-
ment_reg_data_type)(CS)
                            (s'))))
                    (state(read_data(pm_phy, pdb_data_type)(PDBR)
                        (state(read_data(pm_phy, seg-
ment_reg_data_type)(CS)
                            (s'))))))))
      ⊃
      OK?(translate(0,
                    data(read_data(pm_phy, pdb_data_type)(PDBR)
                        (state(read_data(pm_phy, seg-
ment_reg_data_type)(CS)
                            (s'))))'base_addr,
                    addr' + delta', ac',
                    segment_to_priv(data(read_data(pm_phy, seg-
ment_reg_data_type)(CS)
                            (s'))))
                    (state(read_data(pm_phy, pdb_data_type)(PDBR)
                        (state(read_data(pm_phy, seg-
ment_reg_data_type)(CS)
                            (s'))))))))
      ⊃
      OK?(translate(0,
                    data(read_data(pm_phy, pdb_data_type)(PDBR)
                        (state(read_data(pm_phy, seg-
ment_reg_data_type)(CS)
                            (s'))))'base_addr,
                    addr', ac',
                    segment_to_priv(data(read_data(pm_phy, seg-
ment_reg_data_type)(CS)
                            (s'))))
                    (state(read_data(pm_phy, pdb_data_type)(PDBR)
                        (state(read_data(pm_phy, seg-
ment_reg_data_type)(CS)
                            (s'))))))))
      ⊃
      translate(0,
                data(read_data(pm_phy, pdb_data_type)(PDBR)
                    (state(read_data(pm_phy, segment_reg_data_type)(CS)
                            (s'))))'base_addr,
                addr', ac',
                segment_to_priv(data(read_data(pm_phy, seg-
ment_reg_data_type)(CS)(s'))))
                (state(read_data(pm_phy, pdb_data_type)(PDBR)
                    (state(read_data(pm_phy, segment_reg_data_type)(CS)
                            (s'))))))))
      =
2076 translate(0,
          data(read_data(pm_phy, pdb_data_type)(PDBR)
              (state(read_data(pm_phy, segment_reg_data_type)(CS)
                      (s'))))'base_addr,
          addr' + delta', ac',
          segment_to_priv(data(read_data(pm_phy, seg-
ment_reg_data_type)(CS)(s'))))
          (state(read_data(pm_phy, pdb_data_type)(PDBR)
              (state(read_data(pm_phy, segment_reg_data_type)(CS)
                      (s'))))))))

```

C.117 Proofs for Linear_Memory_Properties (challenge-linear.pvs)

Keeping (-4 -5 -6 -8 1) and hiding *,

Rewriting using delta_memory_address, matching in *,

This completes the proof of `linear_resolve_same_page_address_ok.2.2`.

linear_resolve_same_page_address_ok.2.3:

```

{-1}  rem(expt(2, min_page))(offset(addr')) + delta' < expt(2, min_page) ∧
      OK?(translate(0,
                    data(read_data(pm_phy, pdb_data_type)(PDBR)
                          (state(read_data(pm_phy, seg-
ment_reg_data_type)(CS)
                                (s'))))'base_addr,
                    addr', ac',
                    segment_to_priv(data(read_data(pm_phy, seg-
ment_reg_data_type)(CS)
                                (s'))))
                    (state(read_data(pm_phy, pdb_data_type)(PDBR)
                          (state(read_data(pm_phy, seg-
ment_reg_data_type)(CS)
                                (s'))))))))
      ⊃
      OK?(translate(0,
                    data(read_data(pm_phy, pdb_data_type)(PDBR)
                          (state(read_data(pm_phy, seg-
ment_reg_data_type)(CS)
                                (s'))))'base_addr,
                    addr' + delta', ac',
                    segment_to_priv(data(read_data(pm_phy, seg-
ment_reg_data_type)(CS)
                                (s'))))
                    (state(read_data(pm_phy, pdb_data_type)(PDBR)
                          (state(read_data(pm_phy, seg-
ment_reg_data_type)(CS)
                                (s'))))))))
      (s'))))
{-2}  rem(expt(2, min_page))(offset(addr')) + delta' < expt(2, min_page) ∧
      OK?(translate(0,
                    data(read_data(pm_phy, pdb_data_type)(PDBR)
                          (state(read_data(pm_phy, seg-
ment_reg_data_type)(CS)
                                (s'))))'base_addr,
                    addr', ac',
                    segment_to_priv(data(read_data(pm_phy, seg-
ment_reg_data_type)(CS)
                                (s'))))
                    (state(read_data(pm_phy, pdb_data_type)(PDBR)
                          (state(read_data(pm_phy, seg-
ment_reg_data_type)(CS)
                                (s'))))))))
      ⊃
      translate(0,
                data(read_data(pm_phy, pdb_data_type)(PDBR)
                      (state(read_data(pm_phy, segment_reg_data_type)(CS)
                                (s'))))'base_addr,
                addr', ac',
                segment_to_priv(data(read_data(pm_phy, seg-
ment_reg_data_type)(CS)(s'))))
                (state(read_data(pm_phy, pdb_data_type)(PDBR)
                      (state(read_data(pm_phy, segment_reg_data_type)(CS)
                                (s'))))))))
      =
      translate(0,
                data(read_data(pm_phy, pdb_data_type)(PDBR)
                      (state(read_data(pm_phy, segment_reg_data_type)(CS)
                                (s'))))'base_addr,
                addr' + delta', ac',
                segment_to_priv(data(read_data(pm_phy, seg-
ment_reg_data_type)(CS)(s'))))
                (state(read_data(pm_phy, pdb_data_type)(PDBR)
                      (state(read_data(pm_phy, segment_reg_data_type)(CS)
                                (s'))))))))

```

Keeping (-3 -4 -6 1) and hiding *,
 Expanding the definition of +,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 This completes the proof of linear_resolve_same_page_address_ok.2.3.
 Q.E.D.

C.117.67

Linear_Memory_Properties.linear_resolve_same_page_address_TCC1

Terse proof for linear_resolve_same_page_address_TCC1.

linear_resolve_same_page_address_TCC1:

$\{1\} \quad \forall (s: \text{Linear_memory}[\text{Physical_memory}, \text{pm_phy}], \text{addr}: \text{Memory_Address_4G}, \delta: \text{nat}, \text{ac}: \text{Memory_access}):$ $\text{rem}(\text{expt}(2, \text{min_page}))(\text{offset}(\text{addr})) + \delta < \text{expt}(2, \text{min_page}) \wedge$ $\text{OK}?(\text{linear_resolve}(\text{addr}, \text{ac})(s))$ \supset $\text{OK}?([\text{Physical_memory}, \text{Address}]$ $(\text{linear_resolve}[\text{Physical_memory}, \text{pm_phy}](\text{addr} + \delta, \text{ac})(s))$
--

Repeatedly Skolemizing and flattening,
 Rewriting using linear_resolve_same_page_address_ok, matching in *,
 This completes the proof of linear_resolve_same_page_address_TCC1.
 Q.E.D.

C.117.68

Linear_Memory_Properties.linear_resolve_same_page_address

The L^AT_EX code for this proof is broken.

C.117.69 Linear_Memory_Properties.address_block_split_type

Terse proof for address_block_split_type.

address_block_split_type:

$\{1\} \quad \forall (\text{pm}: \text{Plain_Memory}[\text{Linear_memory}], a: \text{Memory_Address_4G}, \text{bl}: \text{list}[\text{Byte}]):$ $\text{is_linear_plain_memory}?(\text{pm}) \wedge$ $(\text{address_block}(a, \text{length}(\text{bl})) \subseteq (\text{pm}'\text{ro_addr} \cup \text{pm}'\text{rw_addr}))$ \supset $\text{every}(\lambda (t: [\text{Address}, \text{list}[\text{Byte}]]):$ $\text{Mem}?(t'1'\text{type_of}) \wedge 0 \leq t'1'\text{offset} \wedge t'1'\text{offset} < \text{max_linear_offset}$ $(\text{split}(\text{min_page}, a, \text{bl}))$

Repeatedly Skolemizing and flattening,
 Using lemma split_type,
 Using lemma address_block_in_memory,
 Using lemma address_block_in_memory2,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 we get 3 subgoals:

C Proof scripts

address_block_split_type.1:

{-1}	is_linear_plain_memory?(pm')
{-2}	(address_block(a', length(bl')) \subseteq (pm'ro_addr \cup pm'rw_addr))
{-3}	$0 \leq a'$ 'offset
{-4}	$a' + \text{length}(bl') \leq \text{reg_size}(\text{max_linear})(\text{type_of}(a'))$
{-5}	every(λ (t: [Address, list[Byte]]): type_of(t'1) = type_of(a') \wedge reg_base(min_linear)(type_of(a')) \leq t'1 \wedge t'1 < reg_size(max_linear)(type_of(a')) (split(min_page, a', bl'))
{-6}	Mem?(a' type_of)
{-7}	a' 'offset < max_linear_offset
{-8}	every(λ (x: number): number_field_pred(x) \wedge real_pred(x) \wedge rational_pred(x) \wedge integer_pred(x) \wedge $x \geq 0 \wedge x < \text{max_byte}$ (bl'))
{1}	every(λ (t: [Address, list[Byte]]): Mem?(t'1 type_of) \wedge $0 \leq t'1$ 'offset \wedge $t'1$ 'offset < max_linear_offset (split(min_page, a', bl'))

Hiding formulas: (-8),

Using lemma every_implied,

we get 2 subgoals:

address_block_split_type.1.1:

{-1}	$(\forall (t_1: [\text{Address}, \text{list}[\text{Byte}]]):$ $\text{type_of}(t_1) = \text{type_of}(a') \wedge$ $\text{reg_base}(\text{min_linear})(\text{type_of}(a')) \leq t_1 \wedge$ $t_1 < \text{reg_size}(\text{max_linear})(\text{type_of}(a'))$ $\supset \text{Mem?}(t_1) \wedge 0 \leq \text{offset} \wedge \text{offset} < \text{max_linear_offset})$ \wedge $\text{every}(\lambda (t: [\text{Address}, \text{list}[\text{Byte}]]):$ $\text{type_of}(t) = \text{type_of}(a') \wedge$ $\text{reg_base}(\text{min_linear})(\text{type_of}(a')) \leq t \wedge$ $t < \text{reg_size}(\text{max_linear})(\text{type_of}(a'))$ $(\text{split}(\text{min_page}, a', \text{bl}'))$ \supset $\text{every}(\lambda (t: [\text{Address}, \text{list}[\text{Byte}]]):$ $\text{Mem?}(t) \wedge 0 \leq \text{offset} \wedge \text{offset} < \text{max_linear_offset})$ $(\text{split}(\text{min_page}, a', \text{bl}'))$
{-2}	$\text{is_linear_plain_memory?}(\text{pm}')$
{-3}	$(\text{address_block}(a', \text{length}(\text{bl}')) \subseteq (\text{pm}'\text{ro_addr} \cup \text{pm}'\text{rw_addr}))$
{-4}	$0 \leq a'\text{offset}$
{-5}	$a' + \text{length}(\text{bl}') \leq \text{reg_size}(\text{max_linear})(\text{type_of}(a'))$
{-6}	$\text{every}(\lambda (t: [\text{Address}, \text{list}[\text{Byte}]]):$ $\text{type_of}(t) = \text{type_of}(a') \wedge$ $\text{reg_base}(\text{min_linear})(\text{type_of}(a')) \leq t \wedge$ $t < \text{reg_size}(\text{max_linear})(\text{type_of}(a'))$ $(\text{split}(\text{min_page}, a', \text{bl}'))$
{-7}	$\text{Mem?}(a')$
{-8}	$a'\text{offset} < \text{max_linear_offset}$
{1}	$\text{every}(\lambda (t: [\text{Address}, \text{list}[\text{Byte}]]):$ $\text{Mem?}(t) \wedge 0 \leq \text{offset} \wedge \text{offset} < \text{max_linear_offset})$ $(\text{split}(\text{min_page}, a', \text{bl}'))$

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Repeatedly Skolemizing and flattening,

Keeping (-1 -2 -3 1) and hiding *,

Installing automatic rewrites from: (max_linear! Mem!)

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of address_block_split_type.1.1.

C Proof scripts

address_block_split_type.1.2:

<pre> {-1} is_linear_plain_memory?(pm') {-2} (address_block(a', length(bl')) ⊆ (pm'ro_addr ∪ pm'rw_addr)) {-3} 0 ≤ a'offset {-4} a' + length(bl') ≤ reg_size(max_linear)(type_of(a')) {-5} every(λ (t: [Address, list[Byte]]): type_of(t'1) = type_of(a') ∧ reg_base(min_linear)(type_of(a')) ≤ t'1 ∧ t'1 < reg_size(max_linear)(type_of(a')) (split(min_page, a', bl')) {-6} Mem?(a'type_of) {-7} a'offset < max_linear_offset </pre>	<pre> {1} ∀ (t: [Address, list[Byte]]): type_of(t'1) = type_of(a') ⊃ Mem?(min_linear'type_of) {2} every(λ (t: [Address, list[Byte]]): Mem?(t'1'type_of) ∧ 0 ≤ t'1'offset ∧ t'1'offset < max_linear_offset) (split(min_page, a', bl')) </pre>
--	--

Expanding the definition of min_linear,

Expanding the definition of Mem,

which is trivially true.

This completes the proof of address_block_split_type.1.2.

address_block_split_type.2:

<pre> {-1} is_linear_plain_memory?(pm') {-2} (address_block(a', length(bl')) ⊆ (pm'ro_addr ∪ pm'rw_addr)) {-3} 0 ≤ a'offset {-4} a' + length(bl') ≤ reg_size(max_linear)(type_of(a')) {-5} Mem?(a'type_of) {-6} a'offset < max_linear_offset {-7} every(λ (x: number): number_field_pred(x) ∧ real_pred(x) ∧ rational_pred(x) ∧ integer_pred(x) ∧ x ≥ 0 ∧ x < max_byte) (bl') </pre>	<pre> {1} reg_base(min_linear)(type_of(a')) ≤ a' {2} every(λ (t: [Address, list[Byte]]): Mem?(t'1'type_of) ∧ 0 ≤ t'1'offset ∧ t'1'offset < max_linear_offset) (split(min_page, a', bl')) </pre>
--	--

Keeping (-3 -5 1) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of address_block_split_type.2.

address_block_split_type.3:

<pre> {-1} is_linear_plain_memory?(pm') {-2} (address_block(a', length(bl')) ⊆ (pm'ro_addr ∪ pm'rw_addr)) {-3} 0 ≤ a'offset {-4} Mem?(a'type_of) {-5} a'offset < max_linear_offset {-6} every(λ (x: number): number_field_pred(x) ∧ real_pred(x) ∧ rational_pred(x) ∧ integer_pred(x) ∧ x ≥ 0 ∧ x < max_byte) (bl') </pre>	<pre> {1} cons?(bl') {2} a' + length(bl') ≤ reg_size(max_linear)(type_of(a')) {3} every(λ (t: [Address, list[Byte]]): Mem?(t'type_of) ∧ 0 ≤ t'offset ∧ t'offset < max_linear_offset) (split(min_page, a', bl')) </pre>
--	---

Using lemma split_null,
 Keeping (-1 1 3) and hiding *,
 Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of address_block_split_type.3.
 Q.E.D.

C.117.70

Linear_Memory_Properties.same_page_address_block_read_TCC1

Terse proof for same_page_address_block_read_TCC1.

same_page_address_block_read_TCC1:

<pre> {1} ∃ (pm: Plain_Memory [Linear_memory], s: Linear_memory [Physical_memory, pm_phy], a1, a2: Memory_Address_4G, s1, s2: posnat): is_linear_plain_memory?(pm) ∧ pm'states(s) ∧ union(pm'ro_addr, pm'rw_addr)(a2) ∧ a1 ≤ a2 ∧ a2 + s2 ≤ a1 + s1 ∧ rem(expt(2, min_page))(offset(a2)) + s2 ≤ expt(2, min_page) ∧ (address_block(a1, s1) ⊆ (pm'ro_addr ∪ pm'rw_addr)) ⊃ OK?[Physical_memory, Address](linear_resolve[Physical_memory, pm_phy](a2, Read)(s)) </pre>
--

Repeatedly Skolemizing and flattening,
 Using lemma pm_linear_resolve_read_ok,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 This completes the proof of same_page_address_block_read_TCC1.
 Q.E.D.

C.117.71 Linear_Memory_Properties.same_page_address_block_read

Terse proof for same_page_address_block_read.

same_page_address_block_read:

<pre> {1} ∃ (pm: Plain_Memory [Linear_memory], s: Linear_memory [Physical_memory, pm_phy], a₁, a₂: Memory_Address_4G, s₁, s₂: posnat): is_linear_plain_memory?(pm) ∧ pm'states(s) ∧ union(pm'ro_addr, pm'rw_addr)(a₂) ∧ a₁ ≤ a₂ ∧ a₂ + s₂ ≤ a₁ + s₁ ∧ rem(expt(2, min_page))(offset(a₂)) + s₂ ≤ expt(2, min_page) ∧ (address_block(a₁, s₁) ⊆ (pm'ro_addr ∪ pm'rw_addr)) ⊃ (address_block(data(linear_resolve(a₂, Read)(s)), s₂) ⊆ (pm_phy'ro_addr ∪ pm_phy'rw_addr)) </pre>

Installing automatic rewrites from: singleton member subset?

Repeatedly Skolemizing and flattening,

Using lemma pm_linear_blessed,

Expanding the definition of linear_blessed?,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Hiding formulas: (-4 -6 -7 -8 -9),

Instantiating the top quantifier in -3 with the terms: (a2!1),

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 2 subgoals:

same_page_address_block_read.1:

<pre> {-1} is_linear_plain_memory?(pm') {-2} pm'states(s') {-3} OK?(linear_resolve(a'₂, Read)(s')) {-4} OK?(linear_resolve(a'₂, Execute)(s')) {-5} ∃ (x: Memory_Address_4G): virt_to_phys_range(s', (restrict [Address, Memory_Address_4G, boolean] (pm'ro_addr) ∪ restrict (singleton(Read) ∪ singleton(Execute))) (x)) ⇒ restrict [Address, Memory_Address_4G, boolean] ((pm_phy'ro_addr ∪ pm_phy'rw_addr))(x) {-6} Mem?(a'₁'type_of) {-7} 0 ≤ a'₁'offset {-8} a'₁'offset < max_linear_offset {-9} Mem?(a'₂'type_of) {-10} 0 ≤ a'₂'offset {-11} a'₂'offset < max_linear_offset {-12} s'₁ > 0 {-13} s'₂ > 0 {-14} union(pm'ro_addr, pm'rw_addr)(a'₂) {-15} a'₁ ≤ a'₂ {-16} a'₂ + s'₂ ≤ a'₁ + s'₁ {-17} rem(expt(2, min_page))(offset(a'₂)) + s'₂ ≤ expt(2, min_page) {-18} ∃ (x: Address): address_block(a'₁, s'₁)(x) ⇒ union(pm'ro_addr, pm'rw_addr)(x) {1} ∃ (x: Address): address_block(data(linear_resolve(a'₂, Read)(s')), s'₂)(x) ⇒ union(pm_phy'ro_addr, pm_phy'rw_addr)(x) </pre>

Repeatedly Skolemizing and flattening,

Case splitting on $\text{offset}(x!1) - \text{offset}(\text{data}[\text{Physical_memory}, \text{Address}](\text{linear_resolve}[\text{Physical_memory}, \text{pm_phy}](a2!1, \text{Read})(s!1))) \geq 0$,

we get 2 subgoals:

same_page_address_block_read.1.1:

{-1}	$\text{offset}(x') - \text{offset}(\text{data}[\text{Physical_memory}, \text{Address}](\text{linear_resolve}[\text{Physical_memory}, \text{pm_phy}](a'_2, \text{Read})(s'_1))) \geq 0$
{-2}	$\text{is_linear_plain_memory?}(\text{pm}')$
{-3}	$\text{pm}' \text{'states}(s')$
{-4}	$\text{OK?}(\text{linear_resolve}(a'_2, \text{Read})(s'))$
{-5}	$\text{OK?}(\text{linear_resolve}(a'_2, \text{Execute})(s'))$
{-6}	$\forall (x: \text{Memory_Address_4G}):$ $\text{virt_to_phys_range}(s',$ $\quad (\text{restrict}[\text{Address}, \text{Memory_Address_4G}, \text{boolean}](\text{pm}'\text{'ro_addr}) \cup \text{restrict}[\text{Address}, \text{Memory_Address_4G}, \text{boolean}](\text{pm}'\text{'rw_addr}))$ $\quad (\text{singleton}(\text{Read}) \cup \text{singleton}(\text{Execute})))$ $\quad (x)$ \Rightarrow $\text{restrict}[\text{Address}, \text{Memory_Address_4G}, \text{boolean}]((\text{pm_phy}'\text{'ro_addr} \cup \text{pm_phy}'\text{'rw_addr}))(x)$
{-7}	$\text{Mem?}(a'_1 \text{'type_of})$
{-8}	$0 \leq a'_1 \text{'offset}$
{-9}	$a'_1 \text{'offset} < \text{max_linear_offset}$
{-10}	$\text{Mem?}(a'_2 \text{'type_of})$
{-11}	$0 \leq a'_2 \text{'offset}$
{-12}	$a'_2 \text{'offset} < \text{max_linear_offset}$
{-13}	$s'_1 > 0$
{-14}	$s'_2 > 0$
{-15}	$\text{union}(\text{pm}'\text{'ro_addr}, \text{pm}'\text{'rw_addr})(a'_2)$
{-16}	$a'_1 \leq a'_2$
{-17}	$a'_2 + s'_2 \leq a'_1 + s'_1$
{-18}	$\text{rem}(\text{expt}(2, \text{min_page}))(\text{offset}(a'_2)) + s'_2 \leq \text{expt}(2, \text{min_page})$
{-19}	$\forall (x: \text{Address}): \text{address_block}(a'_1, s'_1)(x) \Rightarrow \text{union}(\text{pm}'\text{'ro_addr}, \text{pm}'\text{'rw_addr})(x)$
{-20}	$\text{address_block}(\text{data}(\text{linear_resolve}(a'_2, \text{Read})(s')), s'_2)(x')$
{1}	$\text{union}(\text{pm_phy}'\text{'ro_addr}, \text{pm_phy}'\text{'rw_addr})(x')$

Using lemma linear_resolve_same_page_address,

Case splitting on $\text{rem}(\text{expt}(2, \text{min_page}))(\text{offset}(a2!1)) + \text{offset}(x!1) - \text{offset}(\text{data}(\text{linear_resolve}(a2!1, \text{Read})(s!1))) < \text{expt}(2, \text{min_page})$,

we get 2 subgoals:

same_page_address_block_read.1.1.1:

{-1}	$\text{rem}(\text{expt}(2, \text{min_page}))(\text{offset}(a'_2)) + \text{offset}(x') - \text{offset}(\text{data}(\text{linear_resolve}(a'_2, \text{Read})(s')))$
	$< \text{expt}(2, \text{min_page})$
{-2}	$\text{rem}(\text{expt}(2, \text{min_page}))(\text{offset}(a'_2)) + \text{offset}(x') - \text{offset}(\text{data}(\text{linear_resolve}(a'_2, \text{Read})(s')))$
	$< \text{expt}(2, \text{min_page})$
	$\wedge \text{OK}?(\text{linear_resolve}(a'_2, \text{Read})(s'))$
	\supset
	$\text{data}(\text{linear_resolve}(a'_2, \text{Read})(s')) + \text{offset}(x') - \text{offset}(\text{data}(\text{linear_resolve}(a'_2, \text{Read})(s')))$
	$=$
	$\text{data}(\text{linear_resolve}(a'_2 + \text{offset}(x') - \text{offset}(\text{data}(\text{linear_resolve}(a'_2, \text{Read})(s'))),$
	Read
	$(s'))$
{-3}	$\text{offset}(x') - \text{offset}(\text{data} \text{ [Physical_memory, Address]}(\text{linear_resolve} \text{ [Physical_memory, pm_phy]}($
	≥ 0
{-4}	$\text{is_linear_plain_memory}?(\text{pm}')$
{-5}	$\text{pm}' \text{'states}(s')$
{-6}	$\text{OK}?(\text{linear_resolve}(a'_2, \text{Read})(s'))$
{-7}	$\text{OK}?(\text{linear_resolve}(a'_2, \text{Execute})(s'))$
{-8}	$\forall (x: \text{Memory_Address_4G}):$
	$\text{virt_to_phys_range}(s',$
	$(\text{restrict}[\text{Address}, \text{Memory_Address_4G}, \text{boolean}](\text{pm}'\text{'ro_addr}) \cup \text{restrict}[\text{Address}, \text{Memory_Address_4G}, \text{boolean}](\text{pm}'\text{'rw_addr}))$
	$(\text{singleton}(\text{Read}) \cup \text{singleton}(\text{Execute}))$
	(x)
	\Rightarrow
	$\text{restrict}[\text{Address}, \text{Memory_Address_4G}, \text{boolean}]((\text{pm_phy}'\text{'ro_addr} \cup \text{pm_phy}'\text{'rw_addr}))(x)$
{-9}	$\text{Mem}?(a'_1 \text{'type_of})$
{-10}	$0 \leq a'_1 \text{'offset}$
{-11}	$a'_1 \text{'offset} < \text{max_linear_offset}$
{-12}	$\text{Mem}?(a'_2 \text{'type_of})$
{-13}	$0 \leq a'_2 \text{'offset}$
{-14}	$a'_2 \text{'offset} < \text{max_linear_offset}$
{-15}	$s'_1 > 0$
{-16}	$s'_2 > 0$
{-17}	$\text{union}(\text{pm}'\text{'ro_addr}, \text{pm}'\text{'rw_addr})(a'_2)$
{-18}	$a'_1 \leq a'_2$
{-19}	$a'_2 + s'_2 \leq a'_1 + s'_1$
{-20}	$\text{rem}(\text{expt}(2, \text{min_page}))(\text{offset}(a'_2)) + s'_2 \leq \text{expt}(2, \text{min_page})$
{-21}	$\forall (x: \text{Address}): \text{address_block}(a'_1, s'_1)(x) \Rightarrow \text{union}(\text{pm}'\text{'ro_addr}, \text{pm}'\text{'rw_addr})(x)$
{-22}	$\text{address_block}(\text{data}(\text{linear_resolve}(a'_2, \text{Read})(s')), s'_2)(x')$
{1}	$\text{union}(\text{pm_phy}'\text{'ro_addr}, \text{pm_phy}'\text{'rw_addr})(x')$

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Case splitting on $\text{data}(\text{linear_resolve}(a2!1, \text{Read})(s!1)) + (\text{offset}(x!1) - \text{offset}(\text{data}(\text{linear_resolve}(a2!1, \text{Read})(s!1)))) = x!1,$

we get 2 subgoals:

same_page_address_block_read.1.1.1.1:

{-1}	$\text{data}(\text{linear_resolve}(a'_2, \text{Read})(s')) + (\text{offset}(x') - \text{offset}(\text{data}(\text{linear_resolve}(a'_2, \text{Read})(s'))))$
{-2}	$= x'$ $\text{rem}(\text{expt}(2, \text{min_page}))(\text{offset}(a'_2)) + \text{offset}(x') - \text{offset}(\text{data}(\text{linear_resolve}(a'_2, \text{Read})(s')))$ $< \text{expt}(2, \text{min_page})$
{-3}	$\text{OK}?(\text{linear_resolve}(a'_2, \text{Read})(s'))$
{-4}	$\text{data}(\text{linear_resolve}(a'_2, \text{Read})(s')) + \text{offset}(x') - \text{offset}(\text{data}(\text{linear_resolve}(a'_2, \text{Read})(s')))$ $=$ $\text{data}(\text{linear_resolve}(a'_2 + \text{offset}(x') - \text{offset}(\text{data}(\text{linear_resolve}(a'_2, \text{Read})(s'))),$ $\text{Read})$ $(s'))$
{-5}	$\text{offset}(x') - \text{offset}(\text{data} [\text{Physical_memory}, \text{Address}](\text{linear_resolve} [\text{Physical_memory}, \text{pm_phy}](a'_2, \text{Read})(s')))$ ≥ 0
{-6}	$\text{is_linear_plain_memory}?(\text{pm}')$
{-7}	$\text{pm}' \text{'states}(s')$
{-8}	$\text{OK}?(\text{linear_resolve}(a'_2, \text{Execute})(s'))$
{-9}	$\forall (x: \text{Memory_Address_4G}):$ $\text{virt_to_phys_range}(s',$ $\text{restrict}[\text{Address}, \text{Memory_Address_4G}, \text{boolean}](\text{pm}' \text{'ro_addr}) \cup \text{restrict} [\text{Address},$ $\text{singleton}(\text{Read}) \cup \text{singleton}(\text{Execute}))$ (x) \Rightarrow $\text{restrict}[\text{Address}, \text{Memory_Address_4G}, \text{boolean}]((\text{pm_phy}' \text{'ro_addr} \cup \text{pm_phy}' \text{'rw_addr}))(x)$
{-10}	$\text{Mem}?(a'_1 \text{'type_of})$
{-11}	$0 \leq a'_1 \text{'offset}$
{-12}	$a'_1 \text{'offset} < \text{max_linear_offset}$
{-13}	$\text{Mem}?(a'_2 \text{'type_of})$
{-14}	$0 \leq a'_2 \text{'offset}$
{-15}	$a'_2 \text{'offset} < \text{max_linear_offset}$
{-16}	$s'_1 > 0$
{-17}	$s'_2 > 0$
{-18}	$\text{union}(\text{pm}' \text{'ro_addr}, \text{pm}' \text{'rw_addr})(a'_2)$
{-19}	$a'_1 \leq a'_2$
{-20}	$a'_2 + s'_2 \leq a'_1 + s'_1$
{-21}	$\text{rem}(\text{expt}(2, \text{min_page}))(\text{offset}(a'_2)) + s'_2 \leq \text{expt}(2, \text{min_page})$
{-22}	$\forall (x: \text{Address}): \text{address_block}(a'_1, s'_1)(x) \Rightarrow \text{union}(\text{pm}' \text{'ro_addr}, \text{pm}' \text{'rw_addr})(x)$
{-23}	$\text{address_block}(\text{data}(\text{linear_resolve}(a'_2, \text{Read})(s')), s'_2)(x')$
{1}	$\text{union}(\text{pm_phy}' \text{'ro_addr}, \text{pm_phy}' \text{'rw_addr})(x')$

Replacing using formula -1,

Replacing using formula -4,

Case splitting on $\text{OK}?(\text{linear_resolve}(a_2!1 + (\text{offset}(x!1) - \text{offset}(\text{data}(\text{linear_resolve}(a_2!1, \text{Read})(s!1))))), \text{Read} (s!1))$,

we get 2 subgoals:

same_page_address_block_read.1.1.1.1.1:

<p>{-1} OK?(linear_resolve(a'_2 + (offset(x') - offset(data(linear_resolve(a'_2, Read)(s')))), Read (s'))</p> <p>{-2} data(linear_resolve(a'_2, Read)(s')) + (offset(data(linear_resolve(a'_2 + offset(x') - offset(data(linear_resolve(a'_2, Read)(s')))), = data(linear_resolve(a'_2 + offset(x') - offset(data(linear_resolve(a'_2, Read)(s')))), Read (s'))</p> <p>{-3} rem(expt(2, min_page))(offset(a'_2)) + offset(data(linear_resolve(a'_2 + offset(x') - offset(data(linear_resolve(a'_2, Read)(s')))), < expt(2, min_page)</p> <p>{-4} OK?(linear_resolve(a'_2, Read)(s'))</p> <p>{-5} x' = data(linear_resolve(a'_2 + offset(x') - offset(data(linear_resolve(a'_2, Read)(s')))), Read (s'))</p> <p>{-6} offset(data(linear_resolve(a'_2 + offset(x') - offset(data(linear_resolve(a'_2, Read)(s')))), Read)(s')) ≥ 0</p> <p>{-7} is_linear_plain_memory?(pm')</p> <p>{-8} pm' 'states(s')</p> <p>{-9} OK?(linear_resolve(a'_2, Execute)(s'))</p> <p>{-10} ∃ (x: Memory_Address_4G): virt_to_phys_range(s', (restrict[Address, Memory_Address_4G, boolean](pm' 'ro_addr) ∪ restrict[Address, Memory_Address_4G, boolean](pm' 'rw_addr)) (singleton(Read) ∪ singleton(Execute))) (x)</p> <p style="text-align: center;">⇒</p> <p>restrict[Address, Memory_Address_4G, boolean]((pm_phy'ro_addr ∪ pm_phy'rw_addr))(x)</p> <p>{-11} Mem?(a'_1 'type_of)</p> <p>{-12} 0 ≤ a'_1 'offset</p> <p>{-13} a'_1 'offset < max_linear_offset</p> <p>{-14} Mem?(a'_2 'type_of)</p> <p>{-15} 0 ≤ a'_2 'offset</p> <p>{-16} a'_2 'offset < max_linear_offset</p> <p>{-17} s'_1 > 0</p> <p>{-18} s'_2 > 0</p> <p>{-19} union(pm' 'ro_addr, pm' 'rw_addr)(a'_2)</p> <p>{-20} a'_1 ≤ a'_2</p> <p>{-21} a'_2 + s'_2 ≤ a'_1 + s'_1</p> <p>{-22} rem(expt(2, min_page))(offset(a'_2)) + s'_2 ≤ expt(2, min_page)</p> <p>{-23} ∃ (x: Address): address_block(a'_1, s'_1)(x) ⇒ union(pm' 'ro_addr, pm' 'rw_addr)(x)</p> <p>{-24} address_block(data(linear_resolve(a'_2, Read)(s')), s'_2) (data(linear_resolve(a'_2 + offset(x') - offset(data(linear_resolve(a'_2, Read)(s')))), Read (s'))</p>	<p>{1} union(pm_phy'ro_addr, pm_phy'rw_addr) (data(linear_resolve(a'_2 + offset(x') - offset(data(linear_resolve(a'_2, Read)(s')))), Read (s'))</p>
--	---

Instantiating the top quantifier in -10 with the terms: (data(linear_resolve(a2!1 + (offset(x!1) - offset(data(linear_resolve(a2!1, Read)(s!1))))), Read) (s!1))),

we get 2 subgoals:

same_page_address_block_read.1.1.1.1.1.1:

{-1}	OK?(linear_resolve($a'_2 + (\text{offset}(x') - \text{offset}(\text{data}(\text{linear_resolve}(a'_2, \text{Read})(s'))))$), Read) (s'))
{-2}	$\text{data}(\text{linear_resolve}(a'_2, \text{Read})(s')) + (\text{offset}(\text{data}(\text{linear_resolve}(a'_2 + \text{offset}(x') - \text{offset}(\text{data}(\text{linear_resolve}(a'_2, \text{Read})(s'))))$ = $\text{data}(\text{linear_resolve}(a'_2 + \text{offset}(x') - \text{offset}(\text{data}(\text{linear_resolve}(a'_2, \text{Read})(s'))))$, Read) (s'))
{-3}	$\text{rem}(\text{expt}(2, \text{min_page}))(\text{offset}(a'_2)) + \text{offset}(\text{data}(\text{linear_resolve}(a'_2 + \text{offset}(x') - \text{offset}(\text{data}(\text{linear_resolve}(a'_2, \text{Read})(s'))))$ < $\text{expt}(2, \text{min_page})$
{-4}	OK?(linear_resolve($a'_2, \text{Read})(s')$)
{-5}	$x' =$ $\text{data}(\text{linear_resolve}(a'_2 + \text{offset}(x') - \text{offset}(\text{data}(\text{linear_resolve}(a'_2, \text{Read})(s'))))$, Read) (s'))
{-6}	$\text{offset}(\text{data}(\text{linear_resolve}(a'_2 + \text{offset}(x') - \text{offset}(\text{data}(\text{linear_resolve}(a'_2, \text{Read})(s'))))$, Read)(s')) - $\text{offset}(\text{data}(\text{linear_resolve}(a'_2, \text{Read})(s')))$ ≥ 0
{-7}	is_linear_plain_memory?(pm')
{-8}	$\text{pm}' \text{'states}(s')$
{-9}	OK?(linear_resolve($a'_2, \text{Execute})(s')$)
{-10}	$\text{virt_to_phys_range}(s'$, $(\text{restrict}[\text{Address}, \text{Memory_Address_4G}, \text{boolean}](\text{pm}' \text{'ro_addr}) \cup \text{restrict} [\text{Address},$ $(\text{singleton}(\text{Read}) \cup \text{singleton}(\text{Execute})))$ $(\text{data}(\text{linear_resolve}(a'_2 + (\text{offset}(x') - \text{offset}(\text{data}(\text{linear_resolve}(a'_2, \text{Read})(s'))))$, Read) (s')))
	\Rightarrow $\text{restrict}[\text{Address}, \text{Memory_Address_4G}, \text{boolean}]$ $((\text{pm_phy}' \text{'ro_addr} \cup \text{pm_phy}' \text{'rw_addr}))$ $(\text{data}(\text{linear_resolve}(a'_2 + (\text{offset}(x') - \text{offset}(\text{data}(\text{linear_resolve}(a'_2, \text{Read})(s'))))$, Read) (s')))
{-11}	Mem?($a'_1 \text{'type_of}$)
{-12}	$0 \leq a'_1 \text{'offset}$
{-13}	$a'_1 \text{'offset} < \text{max_linear_offset}$
{-14}	Mem?($a'_2 \text{'type_of}$)
{-15}	$0 \leq a'_2 \text{'offset}$
{-16}	$a'_2 \text{'offset} < \text{max_linear_offset}$
{-17}	$s'_1 > 0$
{-18}	$s'_2 > 0$
{-19}	$\text{union}(\text{pm}' \text{'ro_addr}, \text{pm}' \text{'rw_addr})(a'_2)$
{-20}	$a'_1 \leq a'_2$
{-21}	$a'_2 + s'_2 \leq a'_1 + s'_1$
{-22}	$\text{rem}(\text{expt}(2, \text{min_page}))(\text{offset}(a'_2)) + s'_2 \leq \text{expt}(2, \text{min_page})$
{-23}	$\forall (x: \text{Address}): \text{address_block}(a'_1, s'_1)(x) \Rightarrow \text{union}(\text{pm}' \text{'ro_addr}, \text{pm}' \text{'rw_addr})(x)$
{-24}	$\text{address_block}(\text{data}(\text{linear_resolve}(a'_2, \text{Read})(s')), s'_2)$ $(\text{data}(\text{linear_resolve}(a'_2 + \text{offset}(x') - \text{offset}(\text{data}(\text{linear_resolve}(a'_2, \text{Read})(s'))))$, Read) (s'))
{1}	$\text{union}(\text{pm_phy}' \text{'ro_addr}, \text{pm_phy}' \text{'rw_addr})$ $(\text{data}(\text{linear_resolve}(a'_2 + \text{offset}(x') - \text{offset}(\text{data}(\text{linear_resolve}(a'_2, \text{Read})(s'))))$, Read) (s'))

C Proof scripts

Expanding the definition of restrict,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 Expanding the definition of virt_to_phys_range,
 Instantiating the top quantifier in 1 with the terms: (a2!1 + (offset(x!1) - offset(data(linear_resolve(a2!1, Read)(s!1)))) Read),

we get 3 subgoals:

same_page_address_block_read.1.1.1.1.1.1.1.1:

{-1}	OK?(linear_resolve(a'_2 + (offset(x') - offset(data(linear_resolve(a'_2, Read)(s')))), Read) (s'))
{-2}	data(linear_resolve(a'_2, Read)(s')) + (offset(data(linear_resolve(a'_2 + offset(x') - offset(data(linear_resolve(a'_2, Read)(s')))), = data(linear_resolve(a'_2 + offset(x') - offset(data(linear_resolve(a'_2, Read)(s')))), Read) (s'))
{-3}	offset(data(linear_resolve(a'_2 + offset(x') - offset(data(linear_resolve(a'_2, Read)(s')))), Read)(s')) < expt(2, min_page)
{-4}	OK?(linear_resolve(a'_2, Read)(s'))
{-5}	x' = data(linear_resolve(a'_2 + offset(x') - offset(data(linear_resolve(a'_2, Read)(s')))), Read) (s'))
{-6}	offset(data(linear_resolve(a'_2 + offset(x') - offset(data(linear_resolve(a'_2, Read)(s')))), Read)(s')) ≥ 0
{-7}	is_linear_plain_memory?(pm')
{-8}	pm' 'states(s')
{-9}	OK?(linear_resolve(a'_2, Execute)(s'))
{-10}	Mem?(a'_1 'type_of)
{-11}	0 ≤ a'_1 'offset
{-12}	a'_1 'offset < max_linear_offset
{-13}	Mem?(a'_2 'type_of)
{-14}	0 ≤ a'_2 'offset
{-15}	a'_2 'offset < max_linear_offset
{-16}	s'_1 > 0
{-17}	s'_2 > 0
{-18}	union(pm' 'ro_addr, pm' 'rw_addr)(a'_2)
{-19}	a'_1 ≤ a'_2
{-20}	a'_2 + s'_2 ≤ a'_1 + s'_1
{-21}	rem(expt(2, min_page))(offset(a'_2)) + s'_2 ≤ expt(2, min_page)
{-22}	∀ (x: Address): address_block(a'_1, s'_1)(x) ⇒ union(pm' 'ro_addr, pm' 'rw_addr)(x)
{-23}	address_block(data(linear_resolve(a'_2, Read)(s')), s'_2) (data(linear_resolve(a'_2 + offset(x') - offset(data(linear_resolve(a'_2, Read)(s')))), Read) (s'))
{1}	OK?(linear_resolve(a'_2 + (offset(x') - offset(data(linear_resolve(a'_2, Read)(s')))), Read) (s'))
{2}	union(pm_phy 'ro_addr, pm_phy 'rw_addr) (data(linear_resolve(a'_2 + offset(x') - offset(data(linear_resolve(a'_2, Read)(s')))), Read) (s'))

which is trivially true.

This completes the proof of `same_page_address_block_read.1.1.1.1.1.1.1`.

`same_page_address_block_read.1.1.1.1.1.1.1.2`:

{-1}	$\text{OK?}(\text{linear_resolve}(a'_2 + (\text{offset}(x') - \text{offset}(\text{data}(\text{linear_resolve}(a'_2, \text{Read})(s')))), \text{Read})(s'))$,
{-2}	$\text{data}(\text{linear_resolve}(a'_2, \text{Read})(s')) + (\text{offset}(\text{data}(\text{linear_resolve}(a'_2 + \text{offset}(x') - \text{offset}(\text{data}(\text{linear_resolve}(a'_2, \text{Read})(s')))), \text{Read})(s')) = \text{data}(\text{linear_resolve}(a'_2 + \text{offset}(x') - \text{offset}(\text{data}(\text{linear_resolve}(a'_2, \text{Read})(s')))), \text{Read})(s'))$,
{-3}	$\text{offset}(\text{data}(\text{linear_resolve}(a'_2 + \text{offset}(x') - \text{offset}(\text{data}(\text{linear_resolve}(a'_2, \text{Read})(s')))), \text{Read})(s')) + \text{rem}(\text{expt}(2, \text{min_page}) < \text{expt}(2, \text{min_page})$
{-4}	$\text{OK?}(\text{linear_resolve}(a'_2, \text{Read})(s'))$
{-5}	$x' = \text{data}(\text{linear_resolve}(a'_2 + \text{offset}(x') - \text{offset}(\text{data}(\text{linear_resolve}(a'_2, \text{Read})(s')))), \text{Read})(s'))$,
{-6}	$\text{offset}(\text{data}(\text{linear_resolve}(a'_2 + \text{offset}(x') - \text{offset}(\text{data}(\text{linear_resolve}(a'_2, \text{Read})(s')))), \text{Read})(s')) - \text{offset}(\text{data}(\text{linear_resolve}(a'_2, \text{Read})(s')), \text{Read})(s')) \geq 0$
{-7}	$\text{is_linear_plain_memory?}(\text{pm}')$
{-8}	$\text{pm}' \text{'states}(s')$
{-9}	$\text{OK?}(\text{linear_resolve}(a'_2, \text{Execute})(s'))$
{-10}	$\text{Mem?}(a'_1 \text{'type_of})$
{-11}	$0 \leq a'_1 \text{'offset}$
{-12}	$a'_1 \text{'offset} < \text{max_linear_offset}$
{-13}	$\text{Mem?}(a'_2 \text{'type_of})$
{-14}	$0 \leq a'_2 \text{'offset}$
{-15}	$a'_2 \text{'offset} < \text{max_linear_offset}$
{-16}	$s'_1 > 0$
{-17}	$s'_2 > 0$
{-18}	$\text{union}(\text{pm}' \text{'ro_addr}, \text{pm}' \text{'rw_addr})(a'_2)$
{-19}	$a'_1 \leq a'_2$
{-20}	$a'_2 + s'_2 \leq a'_1 + s'_1$
{-21}	$\text{rem}(\text{expt}(2, \text{min_page}))(\text{offset}(a'_2)) + s'_2 \leq \text{expt}(2, \text{min_page})$
{-22}	$\forall (x: \text{Address}): \text{address_block}(a'_1, s'_1)(x) \Rightarrow \text{union}(\text{pm}' \text{'ro_addr}, \text{pm}' \text{'rw_addr})(x)$
{-23}	$\text{address_block}(\text{data}(\text{linear_resolve}(a'_2, \text{Read})(s')), s'_2) (\text{data}(\text{linear_resolve}(a'_2 + \text{offset}(x') - \text{offset}(\text{data}(\text{linear_resolve}(a'_2, \text{Read})(s')))), \text{Read})(s'))$
{1}	$\text{union}[\text{Memory_access}] (\text{singleton}[\text{Memory_access}](\text{Read}), \text{singleton}[\text{Memory_access}](\text{Execute}))(\text{Read})$
{2}	$\text{union}(\text{pm_phy}' \text{'ro_addr}, \text{pm_phy}' \text{'rw_addr}) (\text{data}(\text{linear_resolve}(a'_2 + \text{offset}(x') - \text{offset}(\text{data}(\text{linear_resolve}(a'_2, \text{Read})(s')))), \text{Read})(s'))$

Using lemma `linear_resolve_same_page_address_ok`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Expanding the definition of union,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `same_page_address_block_read.1.1.1.1.1.1.2`.

same_page_address_block_read.1.1.1.1.1.1.3:

{-1}	$\text{OK?}(\text{linear_resolve}(a'_2 + (\text{offset}(x') - \text{offset}(\text{data}(\text{linear_resolve}(a'_2, \text{Read})(s')))), \text{Read})(s'))$
{-2}	$\text{data}(\text{linear_resolve}(a'_2, \text{Read})(s')) + (\text{offset}(\text{data}(\text{linear_resolve}(a'_2 + \text{offset}(x') - \text{offset}(\text{data}(\text{linear_resolve}(a'_2, \text{Read})(s')))), \text{Read})(s')) = \text{data}(\text{linear_resolve}(a'_2 + \text{offset}(x') - \text{offset}(\text{data}(\text{linear_resolve}(a'_2, \text{Read})(s')))), \text{Read})(s'))$
{-3}	$\text{offset}(\text{data}(\text{linear_resolve}(a'_2 + \text{offset}(x') - \text{offset}(\text{data}(\text{linear_resolve}(a'_2, \text{Read})(s')))), \text{Read})(s')) < \text{expt}(2, \text{min_page})$
{-4}	$\text{OK?}(\text{linear_resolve}(a'_2, \text{Read})(s'))$
{-5}	$x' = \text{data}(\text{linear_resolve}(a'_2 + \text{offset}(x') - \text{offset}(\text{data}(\text{linear_resolve}(a'_2, \text{Read})(s')))), \text{Read})(s'))$
{-6}	$\text{offset}(\text{data}(\text{linear_resolve}(a'_2 + \text{offset}(x') - \text{offset}(\text{data}(\text{linear_resolve}(a'_2, \text{Read})(s')))), \text{Read})(s')) \geq 0$
{-7}	$\text{is_linear_plain_memory?}(\text{pm}')$
{-8}	$\text{pm}'\text{'states}(s')$
{-9}	$\text{OK?}(\text{linear_resolve}(a'_2, \text{Execute})(s'))$
{-10}	$\text{Mem?}(a'_1\text{'type_of})$
{-11}	$0 \leq a'_1\text{'offset}$
{-12}	$a'_1\text{'offset} < \text{max_linear_offset}$
{-13}	$\text{Mem?}(a'_2\text{'type_of})$
{-14}	$0 \leq a'_2\text{'offset}$
{-15}	$a'_2\text{'offset} < \text{max_linear_offset}$
{-16}	$s'_1 > 0$
{-17}	$s'_2 > 0$
{-18}	$\text{union}(\text{pm}'\text{'ro_addr}, \text{pm}'\text{'rw_addr})(a'_2)$
{-19}	$a'_1 \leq a'_2$
{-20}	$a_2 + s'_2 \leq a_1 + s'_1$
{-21}	$\text{rem}(\text{expt}(2, \text{min_page}))(\text{offset}(a'_2)) + s'_2 \leq \text{expt}(2, \text{min_page})$
{-22}	$\forall (x: \text{Address}): \text{address_block}(a'_1, s'_1)(x) \Rightarrow \text{union}(\text{pm}'\text{'ro_addr}, \text{pm}'\text{'rw_addr})(x)$
{-23}	$\text{address_block}(\text{data}(\text{linear_resolve}(a'_2, \text{Read})(s')), s'_2) (\text{data}(\text{linear_resolve}(a'_2 + \text{offset}(x') - \text{offset}(\text{data}(\text{linear_resolve}(a'_2, \text{Read})(s')))), \text{Read})(s'))$
{1}	$\text{union}[\text{Memory_Address_4G}] (\lambda (s: \text{Memory_Address_4G}): \text{pm}'\text{'ro_addr}(s), \lambda (s: \text{Memory_Address_4G}): \text{pm}'\text{'rw_addr}(s)) (a'_2 + (\text{offset}(x') - \text{offset}(\text{data} [\text{Physical_memory}, \text{Address}](\text{linear_resolve} [\text{Physical_memory}$
{2}	$\text{union}(\text{pm_phy}'\text{'ro_addr}, \text{pm_phy}'\text{'rw_addr}) (\text{data}(\text{linear_resolve}(a'_2 + \text{offset}(x') - \text{offset}(\text{data}(\text{linear_resolve}(a'_2, \text{Read})(s')))), \text{Read})(s'))$

Instantiating the top quantifier in -22 with the terms: $(a_2!1 + (\text{offset}(x!1) - \text{offset}(\text{data}(\text{linear_resolve}(a_2!1, \text{Read})(s!1))))$,

Hiding formulas: (-3 -4 -6 -9 2),

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of same_page_address_block_read.1.1.1.1.1.1.3.

same_page_address_block_read.1.1.1.1.1.2:

{-1}	$\text{OK?}(\text{linear_resolve}(a'_2 + (\text{offset}(x') - \text{offset}(\text{data}(\text{linear_resolve}(a'_2, \text{Read})(s')))), \text{Read})(s'))$
{-2}	$\text{data}(\text{linear_resolve}(a'_2, \text{Read})(s')) + (\text{offset}(\text{data}(\text{linear_resolve}(a'_2 + \text{offset}(x') - \text{offset}(\text{data}(\text{linear_resolve}(a'_2, \text{Read})(s')))), \text{Read})(s')) = \text{data}(\text{linear_resolve}(a'_2 + \text{offset}(x') - \text{offset}(\text{data}(\text{linear_resolve}(a'_2, \text{Read})(s')))), \text{Read})(s'))$
{-3}	$\text{rem}(\text{expt}(2, \text{min_page}))(\text{offset}(a'_2)) + \text{offset}(\text{data}(\text{linear_resolve}(a'_2 + \text{offset}(x') - \text{offset}(\text{data}(\text{linear_resolve}(a'_2, \text{Read})(s')))), \text{Read})(s')) < \text{expt}(2, \text{min_page})$
{-4}	$\text{OK?}(\text{linear_resolve}(a'_2, \text{Read})(s'))$
{-5}	$x' = \text{data}(\text{linear_resolve}(a'_2 + \text{offset}(x') - \text{offset}(\text{data}(\text{linear_resolve}(a'_2, \text{Read})(s')))), \text{Read})(s'))$
{-6}	$\text{offset}(\text{data}(\text{linear_resolve}(a'_2 + \text{offset}(x') - \text{offset}(\text{data}(\text{linear_resolve}(a'_2, \text{Read})(s')))), \text{Read})(s')) - \text{offset}(\text{data}(\text{linear_resolve}(a'_2, \text{Read})(s')), \text{Read})(s')) \geq 0$
{-7}	$\text{is_linear_plain_memory?}(\text{pm}')$
{-8}	$\text{pm}' \text{'states}(s')$
{-9}	$\text{OK?}(\text{linear_resolve}(a'_2, \text{Execute})(s'))$
{-10}	$\text{Mem?}(a'_1 \text{'type_of})$
{-11}	$0 \leq a'_1 \text{'offset}$
{-12}	$a'_1 \text{'offset} < \text{max_linear_offset}$
{-13}	$\text{Mem?}(a'_2 \text{'type_of})$
{-14}	$0 \leq a'_2 \text{'offset}$
{-15}	$a'_2 \text{'offset} < \text{max_linear_offset}$
{-16}	$s'_1 > 0$
{-17}	$s'_2 > 0$
{-18}	$\text{union}(\text{pm}' \text{'ro_addr}, \text{pm}' \text{'rw_addr})(a'_2)$
{-19}	$a'_1 \leq a'_2$
{-20}	$a'_2 + s'_2 \leq a'_1 + s'_1$
{-21}	$\text{rem}(\text{expt}(2, \text{min_page}))(\text{offset}(a'_2)) + s'_2 \leq \text{expt}(2, \text{min_page})$
{-22}	$\forall (x: \text{Address}): \text{address_block}(a'_1, s'_1)(x) \Rightarrow \text{union}(\text{pm}' \text{'ro_addr}, \text{pm}' \text{'rw_addr})(x)$
{-23}	$\text{address_block}(\text{data}(\text{linear_resolve}(a'_2, \text{Read})(s')), s'_2) (\text{data}(\text{linear_resolve}(a'_2 + \text{offset}(x') - \text{offset}(\text{data}(\text{linear_resolve}(a'_2, \text{Read})(s')))), \text{Read})(s'))$
{1}	$\text{Mem?}(\text{data}[\text{Physical_memory}, \text{Address}] (\text{linear_resolve}[\text{Physical_memory}, \text{pm_phy}] (a'_2 + (\text{offset}(x') - \text{offset}(\text{data} [\text{Physical_memory}, \text{Address}] (\text{linear_resolve} [\text{Physical_memory}, \text{Read}] (s')) \text{'type_of}))$
	\wedge
	$0 \leq$
	$\text{data}[\text{Physical_memory}, \text{Address}] (\text{linear_resolve}[\text{Physical_memory}, \text{pm_phy}] (a'_2 + (\text{offset}(x') - \text{offset}(\text{data} [\text{Physical_memory}, \text{Address}] (\text{linear_resolve} [\text{Physical_memory}, \text{Read}] (s')) \text{'offset}))$
	\wedge
	$\text{data}[\text{Physical_memory}, \text{Address}] (\text{linear_resolve}[\text{Physical_memory}, \text{pm_phy}] (a'_2 + (\text{offset}(x') - \text{offset}(\text{data} [\text{Physical_memory}, \text{Address}] (\text{linear_resolve} [\text{Physical_memory}, \text{Read}] (s')) \text{'offset}))$
	$< \text{max_linear_offset}$
{2}	$\text{union}(\text{pm_phy}' \text{'ro_addr}, \text{pm_phy}' \text{'rw_addr}) (\text{data}(\text{linear_resolve}(a'_2 + \text{offset}(x') - \text{offset}(\text{data}(\text{linear_resolve}(a'_2, \text{Read})(s')))), \text{Read})(s'))$

C Proof scripts

Using lemma `linear_resolve_memory_address`,

Expanding the definition of `every`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `same_page_address_block_read.1.1.1.1.1.2`.

`same_page_address_block_read.1.1.1.1.1.2`:

{-1}	$\text{data}(\text{linear_resolve}(a'_2, \text{Read})(s')) + (\text{offset}(\text{data}(\text{linear_resolve}(a'_2 + \text{offset}(x') - \text{offset}(\text{data}(\text{linear_resolve}(a'_2, \text{Read})(s'))),$
	$=$
	$\text{data}(\text{linear_resolve}(a'_2 + \text{offset}(x') - \text{offset}(\text{data}(\text{linear_resolve}(a'_2, \text{Read})(s'))),$
	Read
	$(s'))$
{-2}	$\text{rem}(\text{expt}(2, \text{min_page}))(\text{offset}(a'_2)) + \text{offset}(\text{data}(\text{linear_resolve}(a'_2 + \text{offset}(x') - \text{offset}(\text{data}(\text{linear_resolve}(a'_2, \text{Read})(s'))),$
	$< \text{expt}(2, \text{min_page})$
{-3}	$\text{OK}?(\text{linear_resolve}(a'_2, \text{Read})(s'))$
{-4}	$x' =$
	$\text{data}(\text{linear_resolve}(a'_2 + \text{offset}(x') - \text{offset}(\text{data}(\text{linear_resolve}(a'_2, \text{Read})(s'))),$
	Read
	$(s'))$
{-5}	$\text{offset}(\text{data}(\text{linear_resolve}(a'_2 + \text{offset}(x') - \text{offset}(\text{data}(\text{linear_resolve}(a'_2, \text{Read})(s'))), \text{Read})(s'))$
	≥ 0
{-6}	$\text{is_linear_plain_memory}?(pm')$
{-7}	$pm' \text{ 'states}(s')$
{-8}	$\text{OK}?(\text{linear_resolve}(a'_2, \text{Execute})(s'))$
{-9}	$\forall (x: \text{Memory_Address_4G}):$
	$\text{virt_to_phys_range}(s',$
	$(\text{restrict}[\text{Address}, \text{Memory_Address_4G}, \text{boolean}](pm' \text{ 'ro_addr}) \cup \text{restrict}[\text{Address}, \text{Memory_Address_4G}, \text{boolean}](pm' \text{ 'rw_addr})) \cup \text{singleton}(\text{Read}) \cup \text{singleton}(\text{Execute}))$
	(x)
	\Rightarrow
	$\text{restrict}[\text{Address}, \text{Memory_Address_4G}, \text{boolean}]((pm_phy' \text{ 'ro_addr}) \cup pm_phy' \text{ 'rw_addr})(x)$
{-10}	$\text{Mem}?(a'_1 \text{ 'type_of})$
{-11}	$0 \leq a'_1 \text{ 'offset}$
{-12}	$a'_1 \text{ 'offset} < \text{max_linear_offset}$
{-13}	$\text{Mem}?(a'_2 \text{ 'type_of})$
{-14}	$0 \leq a'_2 \text{ 'offset}$
{-15}	$a'_2 \text{ 'offset} < \text{max_linear_offset}$
{-16}	$s'_1 > 0$
{-17}	$s'_2 > 0$
{-18}	$\text{union}(pm' \text{ 'ro_addr}, pm' \text{ 'rw_addr})(a'_2)$
{-19}	$a'_1 \leq a'_2$
{-20}	$a'_2 + s'_2 \leq a'_1 + s'_1$
{-21}	$\text{rem}(\text{expt}(2, \text{min_page}))(\text{offset}(a'_2)) + s'_2 \leq \text{expt}(2, \text{min_page})$
{-22}	$\forall (x: \text{Address}): \text{address_block}(a'_1, s'_1)(x) \Rightarrow \text{union}(pm' \text{ 'ro_addr}, pm' \text{ 'rw_addr})(x)$
{-23}	$\text{address_block}(\text{data}(\text{linear_resolve}(a'_2, \text{Read})(s')), s'_2)$
	$(\text{data}(\text{linear_resolve}(a'_2 + \text{offset}(x') - \text{offset}(\text{data}(\text{linear_resolve}(a'_2, \text{Read})(s'))),$
	Read
	$(s'))$
{1}	$\text{OK}?(\text{linear_resolve}(a'_2 + (\text{offset}(x') - \text{offset}(\text{data}(\text{linear_resolve}(a'_2, \text{Read})(s'))),$
	Read
	$(s'))$
{2}	$\text{union}(pm_phy' \text{ 'ro_addr}, pm_phy' \text{ 'rw_addr})$
	$(\text{data}(\text{linear_resolve}(a'_2 + \text{offset}(x') - \text{offset}(\text{data}(\text{linear_resolve}(a'_2, \text{Read})(s'))),$
	Read
	$(s'))$

Using lemma linear_resolve_same_page_address_ok,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of same_page_address_block_read.1.1.1.1.2.

same_page_address_block_read.1.1.1.2:

{-1}	$\text{rem}(\text{expt}(2, \text{min_page}))(\text{offset}(a'_2)) + \text{offset}(x') - \text{offset}(\text{data}(\text{linear_resolve}(a'_2, \text{Read})(s')))$
	$< \text{expt}(2, \text{min_page})$
{-2}	$\text{OK}?(\text{linear_resolve}(a'_2, \text{Read})(s'))$
{-3}	$\text{data}(\text{linear_resolve}(a'_2, \text{Read})(s')) + \text{offset}(x') - \text{offset}(\text{data}(\text{linear_resolve}(a'_2, \text{Read})(s')))$
	$=$
	$\text{data}(\text{linear_resolve}(a'_2 + \text{offset}(x') - \text{offset}(\text{data}(\text{linear_resolve}(a'_2, \text{Read})(s'))),$
	Read
	$(s'))$
{-4}	$\text{offset}(x') - \text{offset}(\text{data}[\text{Physical_memory}, \text{Address}](\text{linear_resolve}[\text{Physical_memory}, \text{pm_phy}](a'_2, \text{Read})(s')))$
	≥ 0
{-5}	$\text{is_linear_plain_memory}?(\text{pm}')$
{-6}	$\text{pm}' \text{ states}(s')$
{-7}	$\text{OK}?(\text{linear_resolve}(a'_2, \text{Execute})(s'))$
{-8}	$\forall (x: \text{Memory_Address_4G}):$
	$\text{virt_to_phys_range}(s',$
	$(\text{restrict}[\text{Address}, \text{Memory_Address_4G}, \text{boolean}](\text{pm}' \text{ ro_addr}) \cup \text{restrict}[\text{Address}, \text{Memory_Address_4G}, \text{boolean}](\text{pm}' \text{ rw_addr}))$
	(x)
	\Rightarrow
	$\text{restrict}[\text{Address}, \text{Memory_Address_4G}, \text{boolean}]((\text{pm_phy}' \text{ ro_addr} \cup \text{pm_phy}' \text{ rw_addr}))(x)$
{-9}	$\text{Mem}?(a'_1 \text{ type_of})$
{-10}	$0 \leq a'_1 \text{ offset}$
{-11}	$a'_1 \text{ offset} < \text{max_linear_offset}$
{-12}	$\text{Mem}?(a'_2 \text{ type_of})$
{-13}	$0 \leq a'_2 \text{ offset}$
{-14}	$a'_2 \text{ offset} < \text{max_linear_offset}$
{-15}	$s'_1 > 0$
{-16}	$s'_2 > 0$
{-17}	$\text{union}(\text{pm}' \text{ ro_addr}, \text{pm}' \text{ rw_addr})(a'_2)$
{-18}	$a'_1 \leq a'_2$
{-19}	$a'_2 + s'_2 \leq a'_1 + s'_1$
{-20}	$\text{rem}(\text{expt}(2, \text{min_page}))(\text{offset}(a'_2)) + s'_2 \leq \text{expt}(2, \text{min_page})$
{-21}	$\forall (x: \text{Address}): \text{address_block}(a'_1, s'_1)(x) \Rightarrow \text{union}(\text{pm}' \text{ ro_addr}, \text{pm}' \text{ rw_addr})(x)$
{-22}	$\text{address_block}(\text{data}(\text{linear_resolve}(a'_2, \text{Read})(s')), s'_2)(x')$
{1}	$\text{data}(\text{linear_resolve}(a'_2, \text{Read})(s')) + (\text{offset}(x') - \text{offset}(\text{data}(\text{linear_resolve}(a'_2, \text{Read})(s'))))$
	$= x'$
{2}	$\text{union}(\text{pm_phy}' \text{ ro_addr}, \text{pm_phy}' \text{ rw_addr})(x')$

Expanding the definition of +,

Applying decompose-equality,

Expanding the definition of address_block,

which is trivially true.

This completes the proof of same_page_address_block_read.1.1.1.2.

same_page_address_block_read.1.1.2:

{-1}	$\begin{aligned} & \text{rem}(\text{expt}(2, \text{min_page}))(\text{offset}(a'_2)) + \text{offset}(x') - \text{offset}(\text{data}(\text{linear_resolve}(a'_2, \text{Read})(s'))) \\ & < \text{expt}(2, \text{min_page}) \\ & \wedge \text{OK}?(\text{linear_resolve}(a'_2, \text{Read})(s')) \\ & \supset \\ & \text{data}(\text{linear_resolve}(a'_2, \text{Read})(s')) + \text{offset}(x') - \text{offset}(\text{data}(\text{linear_resolve}(a'_2, \text{Read})(s'))) \\ & = \\ & \text{data}(\text{linear_resolve}(a'_2 + \text{offset}(x') - \text{offset}(\text{data}(\text{linear_resolve}(a'_2, \text{Read})(s'))), \\ & \quad \text{Read}) \\ & \quad (s') \end{aligned}$
{-2}	$\text{offset}(x') - \text{offset}(\text{data} [\text{Physical_memory}, \text{Address}](\text{linear_resolve} [\text{Physical_memory}, \text{pm_phy}](x))) \geq 0$
{-3}	is_linear_plain_memory?(pm')
{-4}	pm' 'states(s')
{-5}	OK?(linear_resolve(a'_2, Read)(s'))
{-6}	OK?(linear_resolve(a'_2, Execute)(s'))
{-7}	$\begin{aligned} & \forall (x: \text{Memory_Address_4G}): \\ & \quad \text{virt_to_phys_range}(s', \\ & \quad \quad (\text{restrict}[\text{Address}, \text{Memory_Address_4G}, \text{boolean}](\text{pm}'\text{'ro_addr}) \cup \text{restrict} \\ & \quad \quad (\text{singleton}(\text{Read}) \cup \text{singleton}(\text{Execute}))) \\ & \quad \quad (x) \\ & \Rightarrow \\ & \quad \text{restrict}[\text{Address}, \text{Memory_Address_4G}, \text{boolean}]((\text{pm_phy}'\text{'ro_addr} \cup \text{pm_phy}'\text{'rw_addr}))(x) \end{aligned}$
{-8}	Mem?(a'_1 'type_of)
{-9}	0 ≤ a'_1 'offset
{-10}	a'_1 'offset < max_linear_offset
{-11}	Mem?(a'_2 'type_of)
{-12}	0 ≤ a'_2 'offset
{-13}	a'_2 'offset < max_linear_offset
{-14}	s'_1 > 0
{-15}	s'_2 > 0
{-16}	union(pm' 'ro_addr, pm' 'rw_addr)(a'_2)
{-17}	a'_1 ≤ a'_2
{-18}	a'_2 + s'_2 ≤ a'_1 + s'_1
{-19}	rem(expt(2, min_page))(offset(a'_2)) + s'_2 ≤ expt(2, min_page)
{-20}	∀ (x: Address): address_block(a'_1, s'_1)(x) ⇒ union(pm' 'ro_addr, pm' 'rw_addr)(x)
{-21}	address_block(data(linear_resolve(a'_2, Read)(s')), s'_2)(x')
{1}	$\begin{aligned} & \text{rem}(\text{expt}(2, \text{min_page}))(\text{offset}(a'_2)) + \text{offset}(x') - \text{offset}(\text{data}(\text{linear_resolve}(a'_2, \text{Read})(s'))) \\ & < \text{expt}(2, \text{min_page}) \end{aligned}$
{2}	union(pm_phy' 'ro_addr, pm_phy' 'rw_addr)(x')

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of same_page_address_block_read.1.1.2.

same_page_address_block_read.1.2:

{-1}	is_linear_plain_memory?(pm')
{-2}	pm' 'states(s')
{-3}	OK?(linear_resolve(a'_2, Read)(s'))
{-4}	OK?(linear_resolve(a'_2, Execute)(s'))
{-5}	$\forall (x: \text{Memory_Address_4G}):$ $\text{virt_to_phys_range}(s',$ $\quad (\text{restrict}[\text{Address}, \text{Memory_Address_4G}, \text{boolean}](\text{pm}'\text{'ro_addr}) \cup \text{restrict} [\text{Address}$ $\quad (\text{singleton}(\text{Read}) \cup \text{singleton}(\text{Execute})))$ $\quad (x)$ \Rightarrow $\text{restrict}[\text{Address}, \text{Memory_Address_4G}, \text{boolean}]((\text{pm_phy}'\text{'ro_addr} \cup \text{pm_phy}'\text{'rw_addr}))(x)$
{-6}	Mem?(a'_1 'type_of)
{-7}	$0 \leq a'_1 \text{'offset}$
{-8}	$a'_1 \text{'offset} < \text{max_linear_offset}$
{-9}	Mem?(a'_2 'type_of)
{-10}	$0 \leq a'_2 \text{'offset}$
{-11}	$a'_2 \text{'offset} < \text{max_linear_offset}$
{-12}	$s'_1 > 0$
{-13}	$s'_2 > 0$
{-14}	$\text{union}(\text{pm}'\text{'ro_addr}, \text{pm}'\text{'rw_addr})(a'_2)$
{-15}	$a'_1 \leq a'_2$
{-16}	$a'_2 + s'_2 \leq a'_1 + s'_1$
{-17}	$\text{rem}(\text{expt}(2, \text{min_page}))(\text{offset}(a'_2)) + s'_2 \leq \text{expt}(2, \text{min_page})$
{-18}	$\forall (x: \text{Address}): \text{address_block}(a'_1, s'_1)(x) \Rightarrow \text{union}(\text{pm}'\text{'ro_addr}, \text{pm}'\text{'rw_addr})(x)$
{-19}	$\text{address_block}(\text{data}(\text{linear_resolve}(a'_2, \text{Read})(s')), s'_2)(x')$
{1}	$\text{offset}(x') - \text{offset}(\text{data} [\text{Physical_memory}, \text{Address}] (\text{linear_resolve} [\text{Physical_memory}, \text{pm_phy}](a'_2, \text{Read})($ ≥ 0
{2}	$\text{union}(\text{pm_phy}'\text{'ro_addr}, \text{pm_phy}'\text{'rw_addr})(x')$

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of same_page_address_block_read.1.2.

same_page_address_block_read.2:

<div style="display: flex; flex-direction: column; gap: 5px;"> <div style="display: flex; align-items: flex-start;"> {-1} is_linear_plain_memory?(pm') </div> <div style="display: flex; align-items: flex-start;"> {-2} pm' 'states(s') </div> <div style="display: flex; align-items: flex-start;"> {-3} $\forall (x: \text{Memory_Address_4G}):$ $\text{virt_to_phys_range}(s',$ <div style="margin-left: 20px;"> $(\text{restrict}[\text{Address}, \text{Memory_Address_4G}, \text{boolean}](\text{pm}'\text{'ro_addr}) \cup \text{restrict}[\text{Address}, \text{Memory_Address_4G}, \text{boolean}](\text{pm}'\text{'rw_addr}))$ $(\text{singleton}(\text{Read}) \cup \text{singleton}(\text{Execute})))$ (x) </div> </div> <div style="margin-left: 40px; margin-top: 5px;"> \Rightarrow $\text{restrict}[\text{Address}, \text{Memory_Address_4G}, \text{boolean}]((\text{pm_phy}'\text{'ro_addr} \cup \text{pm_phy}'\text{'rw_addr}))(x)$ </div> <div style="display: flex; align-items: flex-start;"> {-4} Mem?(a₁' 'type_of) </div> <div style="display: flex; align-items: flex-start;"> {-5} $0 \leq a_1'\text{'offset}$ </div> <div style="display: flex; align-items: flex-start;"> {-6} $a_1'\text{'offset} < \text{max_linear_offset}$ </div> <div style="display: flex; align-items: flex-start;"> {-7} Mem?(a₂' 'type_of) </div> <div style="display: flex; align-items: flex-start;"> {-8} $0 \leq a_2'\text{'offset}$ </div> <div style="display: flex; align-items: flex-start;"> {-9} $a_2'\text{'offset} < \text{max_linear_offset}$ </div> <div style="display: flex; align-items: flex-start;"> {-10} $s_1' > 0$ </div> <div style="display: flex; align-items: flex-start;"> {-11} $s_2' > 0$ </div> <div style="display: flex; align-items: flex-start;"> {-12} $\text{union}(\text{pm}'\text{'ro_addr}, \text{pm}'\text{'rw_addr})(a_2')$ </div> <div style="display: flex; align-items: flex-start;"> {-13} $a_1' \leq a_2'$ </div> <div style="display: flex; align-items: flex-start;"> {-14} $a_2' + s_2' \leq a_1' + s_1'$ </div> <div style="display: flex; align-items: flex-start;"> {-15} $\text{rem}(\text{expt}(2, \text{min_page}))(\text{offset}(a_2')) + s_2' \leq \text{expt}(2, \text{min_page})$ </div> <div style="display: flex; align-items: flex-start;"> {-16} $\forall (x: \text{Address}): \text{address_block}(a_1', s_1')(x) \Rightarrow \text{union}(\text{pm}'\text{'ro_addr}, \text{pm}'\text{'rw_addr})(x)$ </div> </div>	<div style="display: flex; flex-direction: column; gap: 5px;"> <div style="display: flex; align-items: flex-start;"> {1} $\text{union}(\text{restrict}[\text{Address}, \text{Memory_Address_4G}, \text{boolean}](\text{pm}'\text{'ro_addr}),$ $\text{restrict}[\text{Address}, \text{Memory_Address_4G}, \text{boolean}](\text{pm}'\text{'rw_addr}))$ (a_2') </div> <div style="display: flex; align-items: flex-start;"> {2} $\forall (x: \text{Address}):$ $\text{address_block}(\text{data}(\text{linear_resolve}(a_2', \text{Read})(s')), s_2')(x) \Rightarrow$ $\text{union}(\text{pm_phy}'\text{'ro_addr}, \text{pm_phy}'\text{'rw_addr})(x)$ </div> </div>
--	---

Keeping (-12 1) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of same_page_address_block_read.2.

Q.E.D.

C.117.72

Linear_Memory_Properties.same_page_address_block_write_TCC1

Terse proof for same_page_address_block_write_TCC1.

same_page_address_block_write_TCC1:

$\{1\} \quad \forall (pm: \text{Plain_Memory}[\text{Linear_memory}], s: \text{Linear_memory}[\text{Physical_memory}, pm_phy],$ $a_1, a_2: \text{Memory_Address_4G}, s_1, s_2: \text{posnat}):$ $\text{is_linear_plain_memory?}(pm) \wedge$ $pm' \text{states}(s) \wedge$ $pm' \text{rw_addr}(a_2) \wedge$ $a_1 \leq a_2 \wedge$ $a_2 + s_2 \leq a_1 + s_1 \wedge$ $\text{rem}(\text{expt}(2, \text{min_page}))(\text{offset}(a_2)) + s_2 \leq \text{expt}(2, \text{min_page}) \wedge$ $(\text{address_block}(a_1, s_1) \subseteq pm' \text{rw_addr})$ \supset $\text{OK?}[\text{Physical_memory}, \text{Address}](\text{linear_resolve}[\text{Physical_memory}, pm_phy](a_2, \text{Write})(s))$
--

Repeatedly Skolemizing and flattening,

Using lemma pm_linear_resolve_write_ok,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of same_page_address_block_write_TCC1.

Q.E.D.

C.117.73 Linear_Memory_Properties.same_page_address_block_write

Terse proof for same_page_address_block_write.

same_page_address_block_write:

$\{1\} \quad \forall (pm: \text{Plain_Memory}[\text{Linear_memory}], s: \text{Linear_memory}[\text{Physical_memory}, pm_phy],$ $a_1, a_2: \text{Memory_Address_4G}, s_1, s_2: \text{posnat}):$ $\text{is_linear_plain_memory?}(pm) \wedge$ $pm' \text{states}(s) \wedge$ $pm' \text{rw_addr}(a_2) \wedge$ $a_1 \leq a_2 \wedge$ $a_2 + s_2 \leq a_1 + s_1 \wedge$ $\text{rem}(\text{expt}(2, \text{min_page}))(\text{offset}(a_2)) + s_2 \leq \text{expt}(2, \text{min_page}) \wedge$ $(\text{address_block}(a_1, s_1) \subseteq pm' \text{rw_addr})$ $\supset (\text{address_block}(\text{data}(\text{linear_resolve}(a_2, \text{Write})(s)), s_2) \subseteq pm_phy' \text{rw_addr})$

Installing automatic rewrites from: singleton member subset?

Repeatedly Skolemizing and flattening,

Using lemma pm_linear_blessed,

Expanding the definition of linear_blessed?,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Hiding formulas: (-3 -5 -7 -8 -9),

Instantiating the top quantifier in -3 with the terms: (a2!1),

Repeatedly Skolemizing and flattening,

Case splitting on $\text{offset}(x!1) - \text{offset}(\text{data}(\text{linear_resolve}(a2!1, \text{Write})(s!1))) \geq 0$,

we get 2 subgoals:

same_page_address_block_write.1:

{-1}	$\text{offset}(x') - \text{offset}(\text{data}(\text{linear_resolve}(a'_2, \text{Write})(s'))) \geq 0$
{-2}	$\text{is_linear_plain_memory?}(\text{pm}')$
{-3}	$\text{pm}'\text{'states}(s')$
{-4}	$\text{restrict}[\text{Address}, \text{Memory_Address_4G}, \text{boolean}](\text{pm}'\text{'rw_addr})(a'_2) \supset$ $\text{OK?}(\text{linear_resolve}(a'_2, \text{Write})(s'))$
{-5}	$\forall (x: \text{Memory_Address_4G}):$ $\text{virt_to_phys_range}(s', \text{restrict}[\text{Address}, \text{Memory_Address_4G}, \text{boolean}](\text{pm}'\text{'rw_addr}),$ $\text{singleton}(\text{Write}))$ (x) $\Rightarrow \text{restrict}[\text{Address}, \text{Memory_Address_4G}, \text{boolean}](\text{pm_phy}'\text{'rw_addr})(x)$
{-6}	$\text{Mem?}(a'_1\text{'type_of})$
{-7}	$0 \leq a'_1\text{'offset}$
{-8}	$a'_1\text{'offset} < \text{max_linear_offset}$
{-9}	$\text{Mem?}(a'_2\text{'type_of})$
{-10}	$0 \leq a'_2\text{'offset}$
{-11}	$a'_2\text{'offset} < \text{max_linear_offset}$
{-12}	$s'_1 > 0$
{-13}	$s'_2 > 0$
{-14}	$\text{pm}'\text{'rw_addr}(a'_2)$
{-15}	$a'_1 \leq a'_2$
{-16}	$a'_2 + s'_2 \leq a'_1 + s'_1$
{-17}	$\text{rem}(\text{expt}(2, \text{min_page}))(\text{offset}(a'_2)) + s'_2 \leq \text{expt}(2, \text{min_page})$
{-18}	$\forall (x: \text{Address}): \text{address_block}(a'_1, s'_1)(x) \Rightarrow \text{pm}'\text{'rw_addr}(x)$
{-19}	$\text{address_block}(\text{data}(\text{linear_resolve}(a'_2, \text{Write})(s')), s'_2)(x')$
{1}	$\text{pm_phy}'\text{'rw_addr}(x')$

Using lemma `linear_resolve_same_page_address`,

Case splitting on $\text{rem}(\text{expt}(2, \text{min_page}))(\text{offset}(a'_2)) + (\text{offset}(x') - \text{offset}(\text{data}(\text{linear_resolve}(a'_2, \text{Write})(s')))) < \text{expt}(2, \text{min_page})$,

we get 2 subgoals:

same_page_address_block_write.1.1:

{-1}	$\text{rem}(\text{expt}(2, \text{min_page}))(\text{offset}(a'_2)) + (\text{offset}(x') - \text{offset}(\text{data}(\text{linear_resolve}(a'_2, \text{Write})(s'))))$ $< \text{expt}(2, \text{min_page})$
{-2}	$\text{rem}(\text{expt}(2, \text{min_page}))(\text{offset}(a'_2)) + \text{offset}(x') - \text{offset}(\text{data}(\text{linear_resolve}(a'_2, \text{Write})(s')))$ $< \text{expt}(2, \text{min_page})$ $\wedge \text{OK}?(\text{linear_resolve}(a'_2, \text{Write})(s'))$ \supset $\text{data}(\text{linear_resolve}(a'_2, \text{Write})(s')) + \text{offset}(x') - \text{offset}(\text{data}(\text{linear_resolve}(a'_2, \text{Write})(s')))$ $=$ $\text{data}(\text{linear_resolve}(a'_2 + \text{offset}(x') - \text{offset}(\text{data}(\text{linear_resolve}(a'_2, \text{Write})(s'))),$ $\text{Write})$ (s')
{-3}	$\text{offset}(x') - \text{offset}(\text{data}(\text{linear_resolve}(a'_2, \text{Write})(s')))) \geq 0$
{-4}	$\text{is_linear_plain_memory}?(\text{pm}')$
{-5}	$\text{pm}' \text{'states}(s')$
{-6}	$\text{restrict}[\text{Address}, \text{Memory_Address_4G}, \text{boolean}](\text{pm}' \text{'rw_addr}(a'_2)) \supset$ $\text{OK}?(\text{linear_resolve}(a'_2, \text{Write})(s'))$
{-7}	$\forall (x: \text{Memory_Address_4G}):$ $\text{virt_to_phys_range}(s', \text{restrict}[\text{Address}, \text{Memory_Address_4G}, \text{boolean}](\text{pm}' \text{'rw_addr},$ $\text{singleton}(\text{Write}))$ (x) $\Rightarrow \text{restrict}[\text{Address}, \text{Memory_Address_4G}, \text{boolean}](\text{pm_phy}' \text{'rw_addr}(x)$
{-8}	$\text{Mem}?(a'_1 \text{'type_of})$
{-9}	$0 \leq a'_1 \text{'offset}$
{-10}	$a'_1 \text{'offset} < \text{max_linear_offset}$
{-11}	$\text{Mem}?(a'_2 \text{'type_of})$
{-12}	$0 \leq a'_2 \text{'offset}$
{-13}	$a'_2 \text{'offset} < \text{max_linear_offset}$
{-14}	$s'_1 > 0$
{-15}	$s'_2 > 0$
{-16}	$\text{pm}' \text{'rw_addr}(a'_2)$
{-17}	$a'_1 \leq a'_2$
{-18}	$a'_2 + s'_2 \leq a'_1 + s'_1$
{-19}	$\text{rem}(\text{expt}(2, \text{min_page}))(\text{offset}(a'_2)) + s'_2 \leq \text{expt}(2, \text{min_page})$
{-20}	$\forall (x: \text{Address}): \text{address_block}(a'_1, s'_1)(x) \Rightarrow \text{pm}' \text{'rw_addr}(x)$
{-21}	$\text{address_block}(\text{data}(\text{linear_resolve}(a'_2, \text{Write})(s')), s'_2)(x')$
{1}	$\text{pm_phy}' \text{'rw_addr}(x')$

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 2 subgoals:

same_page_address_block_write.1.1.1.1:

{-1}	$\text{rem}(\text{expt}(2, \text{min_page}))(\text{offset}(a'_2)) + \text{offset}(x') - \text{offset}(\text{data}(\text{linear_resolve}(a'_2, \text{Write})(s')))$ $< \text{expt}(2, \text{min_page})$
{-2}	$\text{OK}?(\text{linear_resolve}(a'_2, \text{Write})(s'))$
{-3}	$\text{data}(\text{linear_resolve}(a'_2, \text{Write})(s')) + \text{offset}(x') - \text{offset}(\text{data}(\text{linear_resolve}(a'_2, \text{Write})(s')))$ $=$ $\text{data}(\text{linear_resolve}(a'_2 + \text{offset}(x') - \text{offset}(\text{data}(\text{linear_resolve}(a'_2, \text{Write})(s'))),$ $\text{Write})$ $(s'))$
{-4}	$\text{offset}(x') - \text{offset}(\text{data}(\text{linear_resolve}(a'_2, \text{Write})(s')))$ ≥ 0
{-5}	$\text{is_linear_plain_memory}?(pm')$
{-6}	$pm' \text{'states}(s')$
{-7}	$\forall (x: \text{Memory_Address_4G}):$ $\text{virt_to_phys_range}(s', \text{restrict}[\text{Address}, \text{Memory_Address_4G}, \text{boolean}](pm' \text{'rw_addr}),$ $\text{singleton}(\text{Write}))$ (x) $\Rightarrow \text{restrict}[\text{Address}, \text{Memory_Address_4G}, \text{boolean}](pm_phy \text{'rw_addr})(x)$
{-8}	$\text{Mem}?(a'_1 \text{'type_of})$
{-9}	$0 \leq a'_1 \text{'offset}$
{-10}	$a'_1 \text{'offset} < \text{max_linear_offset}$
{-11}	$\text{Mem}?(a'_2 \text{'type_of})$
{-12}	$0 \leq a'_2 \text{'offset}$
{-13}	$a'_2 \text{'offset} < \text{max_linear_offset}$
{-14}	$s'_1 > 0$
{-15}	$s'_2 > 0$
{-16}	$pm' \text{'rw_addr}(a'_2)$
{-17}	$a'_1 \leq a'_2$
{-18}	$a'_2 + s'_2 \leq a'_1 + s'_1$
{-19}	$\text{rem}(\text{expt}(2, \text{min_page}))(\text{offset}(a'_2)) + s'_2 \leq \text{expt}(2, \text{min_page})$
{-20}	$\forall (x: \text{Address}): \text{address_block}(a'_1, s'_1)(x) \Rightarrow pm' \text{'rw_addr}(x)$
{-21}	$\text{address_block}(\text{data}(\text{linear_resolve}(a'_2, \text{Write})(s')), s'_2)(x')$
{1}	$pm_phy \text{'rw_addr}(x')$

Case splitting on $\text{data}(\text{linear_resolve}(a_2!1, \text{Write})(s!1)) + (\text{offset}(x!1) - \text{offset}(\text{data}(\text{linear_resolve}(a_2!1, \text{Write})(s!1)))) = x!1,$

we get 2 subgoals:

same_page_address_block_write.1.1.1.1.1:

{-1}	$\text{data}(\text{linear_resolve}(a'_2, \text{Write})(s')) + (\text{offset}(x') - \text{offset}(\text{data}(\text{linear_resolve}(a'_2, \text{Write})(s'))))$
	$= x'$
{-2}	$\text{rem}(\text{expt}(2, \text{min_page}))(\text{offset}(a'_2)) + \text{offset}(x') - \text{offset}(\text{data}(\text{linear_resolve}(a'_2, \text{Write})(s')))$
	$< \text{expt}(2, \text{min_page})$
{-3}	$\text{OK}?(\text{linear_resolve}(a'_2, \text{Write})(s'))$
{-4}	$\text{data}(\text{linear_resolve}(a'_2, \text{Write})(s')) + \text{offset}(x') - \text{offset}(\text{data}(\text{linear_resolve}(a'_2, \text{Write})(s')))$
	$=$
	$\text{data}(\text{linear_resolve}(a'_2 + \text{offset}(x') - \text{offset}(\text{data}(\text{linear_resolve}(a'_2, \text{Write})(s'))),$
	Write
	$(s'))$
{-5}	$\text{offset}(x') - \text{offset}(\text{data}(\text{linear_resolve}(a'_2, \text{Write})(s')))) \geq 0$
{-6}	$\text{is_linear_plain_memory}?(pm')$
{-7}	$pm' \text{'states}(s')$
{-8}	$\forall (x: \text{Memory_Address_4G}):$
	$\text{virt_to_phys_range}(s', \text{restrict}[\text{Address}, \text{Memory_Address_4G}, \text{boolean}](pm' \text{'rw_addr}),$
	$\text{singleton}(\text{Write}))$
	(x)
	$\Rightarrow \text{restrict}[\text{Address}, \text{Memory_Address_4G}, \text{boolean}](pm_phy \text{'rw_addr})(x)$
{-9}	$\text{Mem}?(a'_1 \text{'type_of})$
{-10}	$0 \leq a'_1 \text{'offset}$
{-11}	$a'_1 \text{'offset} < \text{max_linear_offset}$
{-12}	$\text{Mem}?(a'_2 \text{'type_of})$
{-13}	$0 \leq a'_2 \text{'offset}$
{-14}	$a'_2 \text{'offset} < \text{max_linear_offset}$
{-15}	$s'_1 > 0$
{-16}	$s'_2 > 0$
{-17}	$pm' \text{'rw_addr}(a'_2)$
{-18}	$a'_1 \leq a'_2$
{-19}	$a'_2 + s'_2 \leq a'_1 + s'_1$
{-20}	$\text{rem}(\text{expt}(2, \text{min_page}))(\text{offset}(a'_2)) + s'_2 \leq \text{expt}(2, \text{min_page})$
{-21}	$\forall (x: \text{Address}): \text{address_block}(a'_1, s'_1)(x) \Rightarrow pm' \text{'rw_addr}(x)$
{-22}	$\text{address_block}(\text{data}(\text{linear_resolve}(a'_2, \text{Write})(s')), s'_2)(x')$
{1}	$pm_phy \text{'rw_addr}(x')$

Replacing using formula -1,

Replacing using formula -4,

Case splitting on $\text{OK}?(\text{linear_resolve}(a2!1 + (\text{offset}(x!1) - \text{offset}(\text{data}(\text{linear_resolve}(a2!1, \text{Write})(s!1))))), \text{Write}(s!1))$,

we get 2 subgoals:

same_page_address_block_write.1.1.1.1.1:

{-1}	OK?(linear_resolve($a'_2 + (\text{offset}(x') - \text{offset}(\text{data}(\text{linear_resolve}(a'_2, \text{Write})(s'))))$), Write) (s'))
{-2}	$\text{data}(\text{linear_resolve}(a'_2, \text{Write})(s')) + (\text{offset}(\text{data}(\text{linear_resolve}(a'_2 + \text{offset}(x') - \text{offset}(\text{data}(\text{linear_resolve}(a'_2, \text{Write})(s'))))$) = $\text{data}(\text{linear_resolve}(a'_2 + \text{offset}(x') - \text{offset}(\text{data}(\text{linear_resolve}(a'_2, \text{Write})(s'))))$, Write) (s'))
{-3}	$\text{rem}(\text{expt}(2, \text{min_page}))(\text{offset}(a'_2)) + \text{offset}(\text{data}(\text{linear_resolve}(a'_2 + \text{offset}(x') - \text{offset}(\text{data}(\text{linear_resolve}(a'_2, \text{Write})(s'))))$ < $\text{expt}(2, \text{min_page})$
{-4}	OK?(linear_resolve($a'_2, \text{Write})(s')$)
{-5}	$x' =$ $\text{data}(\text{linear_resolve}(a'_2 + \text{offset}(x') - \text{offset}(\text{data}(\text{linear_resolve}(a'_2, \text{Write})(s'))))$, Write) (s'))
{-6}	$\text{offset}(\text{data}(\text{linear_resolve}(a'_2 + \text{offset}(x') - \text{offset}(\text{data}(\text{linear_resolve}(a'_2, \text{Write})(s'))))$, Write)(s')) ≥ 0
{-7}	is_linear_plain_memory?(pm')
{-8}	$\text{pm}' \text{'states}(s')$
{-9}	$\forall (x: \text{Memory_Address_4G}):$ $\text{virt_to_phys_range}(s', \text{restrict}[\text{Address}, \text{Memory_Address_4G}, \text{boolean}](\text{pm}' \text{'rw_addr}$), singleton(Write)) (x) $\Rightarrow \text{restrict}[\text{Address}, \text{Memory_Address_4G}, \text{boolean}](\text{pm_phy}' \text{'rw_addr})(x)$
{-10}	Mem?($a'_1 \text{'type_of}$)
{-11}	$0 \leq a'_1 \text{'offset}$
{-12}	$a'_1 \text{'offset} < \text{max_linear_offset}$
{-13}	Mem?($a'_2 \text{'type_of}$)
{-14}	$0 \leq a'_2 \text{'offset}$
{-15}	$a'_2 \text{'offset} < \text{max_linear_offset}$
{-16}	$s'_1 > 0$
{-17}	$s'_2 > 0$
{-18}	$\text{pm}' \text{'rw_addr}(a'_2)$
{-19}	$a'_1 \leq a'_2$
{-20}	$a'_2 + s'_2 \leq a'_1 + s'_1$
{-21}	$\text{rem}(\text{expt}(2, \text{min_page}))(\text{offset}(a'_2)) + s'_2 \leq \text{expt}(2, \text{min_page})$
{-22}	$\forall (x: \text{Address}): \text{address_block}(a'_1, s'_1)(x) \Rightarrow \text{pm}' \text{'rw_addr}(x)$
{-23}	$\text{address_block}(\text{data}(\text{linear_resolve}(a'_2, \text{Write})(s')), s'_2)$ $(\text{data}(\text{linear_resolve}(a'_2 + \text{offset}(x') - \text{offset}(\text{data}(\text{linear_resolve}(a'_2, \text{Write})(s'))))$, Write) (s'))
{1}	$\text{pm_phy}' \text{'rw_addr}$ $(\text{data}(\text{linear_resolve}(a'_2 + \text{offset}(x') - \text{offset}(\text{data}(\text{linear_resolve}(a'_2, \text{Write})(s'))))$, Write) (s'))

Instantiating the top quantifier in -9 with the terms: $(\text{data}(\text{linear_resolve}(a2!1 + (\text{offset}(x!1) - \text{offset}(\text{data}(\text{linear_resolve}(a2!1, \text{Write})(s!1))))$), Write) (s!1))),

we get 2 subgoals:

same_page_address_block_write.1.1.1.1.1.1:

```

{-1} OK?(linear_resolve(a'_2 + (offset(x') - offset(data(linear_resolve(a'_2, Write)(s')))),
                        Write)
      (s'))
{-2} data(linear_resolve(a'_2, Write)(s')) + (offset(data(linear_resolve(a'_2 + offset(x') - offset(data(linear_resolve(a'_2,
=
      data(linear_resolve(a'_2 + offset(x') - offset(data(linear_resolve(a'_2, Write)(s'))),
                        Write)
      (s'))
{-3} rem(expt(2, min_page))(offset(a'_2)) + offset(data(linear_resolve(a'_2 + offset(x') - offset(data(linear_resolve(a'_2,
< expt(2, min_page)
{-4} OK?(linear_resolve(a'_2, Write)(s'))
{-5} x' =
      data(linear_resolve(a'_2 + offset(x') - offset(data(linear_resolve(a'_2, Write)(s'))),
                        Write)
      (s'))
{-6} offset(data(linear_resolve(a'_2 + offset(x') - offset(data(linear_resolve(a'_2, Write)(s'))), Write)(s')) - offset(da
≥ 0
{-7} is_linear_plain_memory?(pm')
{-8} pm' 'states(s')
{-9} virt_to_phys_range(s', restrict[Address, Memory_Address_4G, boolean](pm' 'rw_addr),
      singleton(Write)
      (data(linear_resolve(a'_2 + (offset(x') - offset(data(linear_resolve(a'_2, Write)(s')))),
                        Write)
      (s'))))
⇒
restrict[Address, Memory_Address_4G, boolean]
  (pm_phy 'rw_addr)
  (data(linear_resolve(a'_2 + (offset(x') - offset(data(linear_resolve(a'_2, Write)(s')))),
                        Write)
  (s'))))
{-10} Mem?(a'_1 'type_of)
{-11} 0 ≤ a'_1 'offset
{-12} a'_1 'offset < max_linear_offset
{-13} Mem?(a'_2 'type_of)
{-14} 0 ≤ a'_2 'offset
{-15} a'_2 'offset < max_linear_offset
{-16} s'_1 > 0
{-17} s'_2 > 0
{-18} pm' 'rw_addr(a'_2)
{-19} a'_1 ≤ a'_2
{-20} a'_2 + s'_2 ≤ a'_1 + s'_1
{-21} rem(expt(2, min_page))(offset(a'_2)) + s'_2 ≤ expt(2, min_page)
{-22} ∀ (x: Address): address_block(a'_1, s'_1)(x) ⇒ pm' 'rw_addr(x)
{-23} address_block(data(linear_resolve(a'_2, Write)(s')), s'_2)
      (data(linear_resolve(a'_2 + offset(x') - offset(data(linear_resolve(a'_2, Write)(s'))),
                        Write)
      (s'))))
-----
{1} pm_phy 'rw_addr
      (data(linear_resolve(a'_2 + offset(x') - offset(data(linear_resolve(a'_2, Write)(s'))),
                        Write)
      (s'))))

```

C Proof scripts

Expanding the definition of restrict,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 Expanding the definition of virt_to_phys_range,
 Instantiating the top quantifier in 1 with the terms: (a2!1 + (offset(x!1) - offset(data(linear_resolve(a2!1, Write)(s!1)))) Write),
 we get 2 subgoals:

same_page_address_block_write.1.1.1.1.1.1.1.1:

{-1}	OK?(linear_resolve(a ₂ + (offset(x') - offset(data(linear_resolve(a ₂ , Write)(s')))), Write) (s'))
{-2}	data(linear_resolve(a ₂ , Write)(s')) + (offset(data(linear_resolve(a ₂ + offset(x') - offset(data(linear_resolve(a ₂ , Write)(s')))), = data(linear_resolve(a ₂ + offset(x') - offset(data(linear_resolve(a ₂ , Write)(s')))), Write) (s'))
{-3}	offset(data(linear_resolve(a ₂ + offset(x') - offset(data(linear_resolve(a ₂ , Write)(s')))), Write)(s')) < expt(2, min_page)
{-4}	OK?(linear_resolve(a ₂ , Write)(s'))
{-5}	x' = data(linear_resolve(a ₂ + offset(x') - offset(data(linear_resolve(a ₂ , Write)(s')))), Write) (s'))
{-6}	offset(data(linear_resolve(a ₂ + offset(x') - offset(data(linear_resolve(a ₂ , Write)(s')))), Write)(s')) ≥ 0
{-7}	is_linear_plain_memory?(pm')
{-8}	pm' 'states(s')
{-9}	Mem?(a ₁ 'type_of)
{-10}	0 ≤ a ₁ 'offset
{-11}	a ₁ 'offset < max_linear_offset
{-12}	Mem?(a ₂ 'type_of)
{-13}	0 ≤ a ₂ 'offset
{-14}	a ₂ 'offset < max_linear_offset
{-15}	s ₁ > 0
{-16}	s ₂ > 0
{-17}	pm' 'rw_addr(a ₂)
{-18}	a ₁ ≤ a ₂
{-19}	a ₂ + s ₂ ≤ a ₁ + s ₁
{-20}	rem(expt(2, min_page))(offset(a ₂)) + s ₂ ≤ expt(2, min_page)
{-21}	∀ (x: Address): address_block(a ₁ , s ₁)(x) ⇒ pm' 'rw_addr(x)
{-22}	address_block(data(linear_resolve(a ₂ , Write)(s')), s ₂) (data(linear_resolve(a ₂ + offset(x') - offset(data(linear_resolve(a ₂ , Write)(s')))), Write) (s'))
{1}	OK?(linear_resolve(a ₂ + (offset(x') - offset(data(linear_resolve(a ₂ , Write)(s')))), Write) (s'))
{2}	pm_phy 'rw_addr (data(linear_resolve(a ₂ + offset(x') - offset(data(linear_resolve(a ₂ , Write)(s')))), Write) (s'))

which is trivially true.

This completes the proof of same_page_address_block_write.1.1.1.1.1.1.1.1.

same_page_address_block_write.1.1.1.1.1.2:

{-1}	$\text{OK?}(\text{linear_resolve}(a'_2 + (\text{offset}(x') - \text{offset}(\text{data}(\text{linear_resolve}(a'_2, \text{Write})(s')))), \text{Write})(s'))$,
{-2}	$\text{data}(\text{linear_resolve}(a'_2, \text{Write})(s')) + (\text{offset}(\text{data}(\text{linear_resolve}(a'_2 + \text{offset}(x') - \text{offset}(\text{data}(\text{linear_resolve}(a'_2, \text{Write})(s')))), \text{Write})(s'))$ =
{-3}	$\text{offset}(\text{data}(\text{linear_resolve}(a'_2 + \text{offset}(x') - \text{offset}(\text{data}(\text{linear_resolve}(a'_2, \text{Write})(s'))), \text{Write})(s')) + \text{rem}(\text{expt}(2, \text{min_page})$ < $\text{expt}(2, \text{min_page})$
{-4}	$\text{OK?}(\text{linear_resolve}(a'_2, \text{Write})(s'))$
{-5}	$x' =$ $\text{data}(\text{linear_resolve}(a'_2 + \text{offset}(x') - \text{offset}(\text{data}(\text{linear_resolve}(a'_2, \text{Write})(s'))), \text{Write})(s'))$
{-6}	$\text{offset}(\text{data}(\text{linear_resolve}(a'_2 + \text{offset}(x') - \text{offset}(\text{data}(\text{linear_resolve}(a'_2, \text{Write})(s'))), \text{Write})(s')) - \text{offset}(\text{data}(\text{linear_resolve}(a'_2, \text{Write})(s')))$ ≥ 0
{-7}	$\text{is_linear_plain_memory?}(\text{pm}')$
{-8}	$\text{pm}' \text{'states}(s')$
{-9}	$\text{Mem?}(a'_1 \text{'type_of})$
{-10}	$0 \leq a'_1 \text{'offset}$
{-11}	$a'_1 \text{'offset} < \text{max_linear_offset}$
{-12}	$\text{Mem?}(a'_2 \text{'type_of})$
{-13}	$0 \leq a'_2 \text{'offset}$
{-14}	$a'_2 \text{'offset} < \text{max_linear_offset}$
{-15}	$s'_1 > 0$
{-16}	$s'_2 > 0$
{-17}	$\text{pm}' \text{'rw_addr}(a'_2)$
{-18}	$a'_1 \leq a'_2$
{-19}	$a'_2 + s'_2 \leq a'_1 + s'_1$
{-20}	$\text{rem}(\text{expt}(2, \text{min_page}))(\text{offset}(a'_2)) + s'_2 \leq \text{expt}(2, \text{min_page})$
{-21}	$\forall (x: \text{Address}): \text{address_block}(a'_1, s'_1)(x) \Rightarrow \text{pm}' \text{'rw_addr}(x)$
{-22}	$\text{address_block}(\text{data}(\text{linear_resolve}(a'_2, \text{Write})(s')), s'_2)$ $(\text{data}(\text{linear_resolve}(a'_2 + \text{offset}(x') - \text{offset}(\text{data}(\text{linear_resolve}(a'_2, \text{Write})(s'))), \text{Write})(s'))$
{1}	$\text{pm}' \text{'rw_addr}$ $(a'_2 + (\text{offset}(x') - \text{offset}(\text{data}(\text{linear_resolve}(a'_2, \text{Write})(s'))), \text{Write})(s'))$
{2}	$\text{pm_phy}' \text{'rw_addr}$ $(\text{data}(\text{linear_resolve}(a'_2 + \text{offset}(x') - \text{offset}(\text{data}(\text{linear_resolve}(a'_2, \text{Write})(s'))), \text{Write})(s'))$

Instantiating the top quantifier in -21 with the terms: $(a2!1 + (\text{offset}(x!1) - \text{offset}(\text{data}(\text{linear_resolve}(a2!1, \text{Write})(s!1))))$,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of same_page_address_block_write.1.1.1.1.1.2.

same_page_address_block_write.1.1.1.1.1.2:

{-1}	OK?(linear_resolve($a'_2 + (\text{offset}(x') - \text{offset}(\text{data}(\text{linear_resolve}(a'_2, \text{Write})(s'))))$), Write) (s'))
{-2}	$\text{data}(\text{linear_resolve}(a'_2, \text{Write})(s')) + (\text{offset}(\text{data}(\text{linear_resolve}(a'_2 + \text{offset}(x') - \text{offset}(\text{data}(\text{linear_resolve}(a'_2, \text{Write})(s'))))$ = $\text{data}(\text{linear_resolve}(a'_2 + \text{offset}(x') - \text{offset}(\text{data}(\text{linear_resolve}(a'_2, \text{Write})(s'))))$, Write) (s'))
{-3}	$\text{rem}(\text{expt}(2, \text{min_page}))(\text{offset}(a'_2)) + \text{offset}(\text{data}(\text{linear_resolve}(a'_2 + \text{offset}(x') - \text{offset}(\text{data}(\text{linear_resolve}(a'_2, \text{Write})(s'))))$ < $\text{expt}(2, \text{min_page})$
{-4}	OK?(linear_resolve($a'_2, \text{Write})(s')$)
{-5}	$x' =$ $\text{data}(\text{linear_resolve}(a'_2 + \text{offset}(x') - \text{offset}(\text{data}(\text{linear_resolve}(a'_2, \text{Write})(s'))))$, Write) (s'))
{-6}	$\text{offset}(\text{data}(\text{linear_resolve}(a'_2 + \text{offset}(x') - \text{offset}(\text{data}(\text{linear_resolve}(a'_2, \text{Write})(s'))))$, Write)(s')) ≥ 0
{-7}	is_linear_plain_memory?(pm')
{-8}	pm' 'states(s')
{-9}	Mem?(a'_1 'type_of)
{-10}	$0 \leq a'_1$ 'offset
{-11}	a'_1 'offset < max_linear_offset
{-12}	Mem?(a'_2 'type_of)
{-13}	$0 \leq a'_2$ 'offset
{-14}	a'_2 'offset < max_linear_offset
{-15}	$s'_1 > 0$
{-16}	$s'_2 > 0$
{-17}	pm' 'rw_addr(a'_2)
{-18}	$a'_1 \leq a'_2$
{-19}	$a'_2 + s'_2 \leq a'_1 + s'_1$
{-20}	$\text{rem}(\text{expt}(2, \text{min_page}))(\text{offset}(a'_2)) + s'_2 \leq \text{expt}(2, \text{min_page})$
{-21}	$\forall (x: \text{Address}): \text{address_block}(a'_1, s'_1)(x) \Rightarrow \text{pm}'$ 'rw_addr(x)
{-22}	$\text{address_block}(\text{data}(\text{linear_resolve}(a'_2, \text{Write})(s'))$, s'_2) ($\text{data}(\text{linear_resolve}(a'_2 + \text{offset}(x') - \text{offset}(\text{data}(\text{linear_resolve}(a'_2, \text{Write})(s'))))$, Write) (s'))
{1}	Mem?(data[Physical_memory, Address] (linear_resolve[Physical_memory, pm_phy] ($a'_2 + (\text{offset}(x') - \text{offset}(\text{data} [\text{Physical_memory}, \text{Address}]$ (linear_resolve [Phy Write) (s')'type_of)) \wedge $0 \leq$ data[Physical_memory, Address] (linear_resolve[Physical_memory, pm_phy] ($a'_2 + (\text{offset}(x') - \text{offset}(\text{data} [\text{Physical_memory}, \text{Address}]$ (linear_resolve [Phy Write) (s')'offset)) \wedge data[Physical_memory, Address] (linear_resolve[Physical_memory, pm_phy] ($a'_2 + (\text{offset}(x') - \text{offset}(\text{data} [\text{Physical_memory}, \text{Address}]$ (linear_resolve [Phy Write) (s')'offset)) < max_linear_offset pm_phy'rw_addr ($\text{data}(\text{linear_resolve}(a'_2 + \text{offset}(x') - \text{offset}(\text{data}(\text{linear_resolve}(a'_2, \text{Write})(s'))))$, Write) (s')))
{2}	$\text{pm_phy}'$ rw_addr ($\text{data}(\text{linear_resolve}(a'_2 + \text{offset}(x') - \text{offset}(\text{data}(\text{linear_resolve}(a'_2, \text{Write})(s'))))$, Write) (s'))

Using lemma linear_resolve_memory_address,

Expanding the definition of every,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of same_page_address_block_write.1.1.1.1.1.2.

same_page_address_block_write.1.1.1.1.1.2:

{-1}	$\text{data}(\text{linear_resolve}(a'_2, \text{Write})(s')) + (\text{offset}(\text{data}(\text{linear_resolve}(a'_2 + \text{offset}(x') - \text{offset}(\text{data}(\text{linear_resolve}(a'_2, \text{Write})(s')), \text{Write})(s'))))$ $=$ $\text{data}(\text{linear_resolve}(a'_2 + \text{offset}(x') - \text{offset}(\text{data}(\text{linear_resolve}(a'_2, \text{Write})(s')), \text{Write})(s'))),$
{-2}	$\text{rem}(\text{expt}(2, \text{min_page}))(\text{offset}(a'_2)) + \text{offset}(\text{data}(\text{linear_resolve}(a'_2 + \text{offset}(x') - \text{offset}(\text{data}(\text{linear_resolve}(a'_2, \text{Write})(s')), \text{Write})(s'))))$ $< \text{expt}(2, \text{min_page})$
{-3}	OK?(linear_resolve(a'_2, Write)(s'))
{-4}	$x' =$ $\text{data}(\text{linear_resolve}(a'_2 + \text{offset}(x') - \text{offset}(\text{data}(\text{linear_resolve}(a'_2, \text{Write})(s')), \text{Write})(s'))),$
{-5}	$\text{offset}(\text{data}(\text{linear_resolve}(a'_2 + \text{offset}(x') - \text{offset}(\text{data}(\text{linear_resolve}(a'_2, \text{Write})(s')), \text{Write})(s'))), \text{Write})(s')) - \text{offset}(\text{data}(\text{linear_resolve}(a'_2, \text{Write})(s')), \text{Write})(s'))$ ≥ 0
{-6}	is_linear_plain_memory?(pm')
{-7}	pm' 'states(s')
{-8}	$\forall (x: \text{Memory_Address_4G}):$ $\text{virt_to_phys_range}(s', \text{restrict}[\text{Address}, \text{Memory_Address_4G}, \text{boolean}] (\text{pm}' \text{ 'rw_addr}), \text{singleton}(\text{Write}))(x)$ $\Rightarrow \text{restrict}[\text{Address}, \text{Memory_Address_4G}, \text{boolean}] (\text{pm_phy}' \text{ 'rw_addr})(x)$
{-9}	Mem?(a'_1 'type_of)
{-10}	$0 \leq a'_1 \text{ 'offset}$
{-11}	$a'_1 \text{ 'offset} < \text{max_linear_offset}$
{-12}	Mem?(a'_2 'type_of)
{-13}	$0 \leq a'_2 \text{ 'offset}$
{-14}	$a'_2 \text{ 'offset} < \text{max_linear_offset}$
{-15}	$s'_1 > 0$
{-16}	$s'_2 > 0$
{-17}	pm' 'rw_addr(a'_2)
{-18}	$a'_1 \leq a'_2$
{-19}	$a'_2 + s'_2 \leq a'_1 + s'_1$
{-20}	$\text{rem}(\text{expt}(2, \text{min_page}))(\text{offset}(a'_2)) + s'_2 \leq \text{expt}(2, \text{min_page})$
{-21}	$\forall (x: \text{Address}): \text{address_block}(a'_1, s'_1)(x) \Rightarrow \text{pm}' \text{ 'rw_addr}(x)$
{-22}	$\text{address_block}(\text{data}(\text{linear_resolve}(a'_2, \text{Write})(s')), s'_2)$ $(\text{data}(\text{linear_resolve}(a'_2 + \text{offset}(x') - \text{offset}(\text{data}(\text{linear_resolve}(a'_2, \text{Write})(s')), \text{Write})(s'))), \text{Write})(s'))$
{1}	<hr style="border: 0.5px solid black;"/> $\text{OK?}(\text{linear_resolve}(a'_2 + (\text{offset}(x') - \text{offset}(\text{data}(\text{linear_resolve}(a'_2, \text{Write})(s')), \text{Write})(s'))), \text{Write})(s'))$
{2}	$\text{pm_phy}' \text{ 'rw_addr}$ $(\text{data}(\text{linear_resolve}(a'_2 + \text{offset}(x') - \text{offset}(\text{data}(\text{linear_resolve}(a'_2, \text{Write})(s')), \text{Write})(s'))), \text{Write})(s'))$

Using lemma linear_resolve_same_page_address_ok,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

C Proof scripts

This completes the proof of `same_page_address_block_write.1.1.1.1.2`.

`same_page_address_block_write.1.1.1.2`:

{-1}	$\text{rem}(\text{expt}(2, \text{min_page}))(\text{offset}(a'_2)) + \text{offset}(x') - \text{offset}(\text{data}(\text{linear_resolve}(a'_2, \text{Write})(s')))$ $< \text{expt}(2, \text{min_page})$
{-2}	$\text{OK}?(\text{linear_resolve}(a'_2, \text{Write})(s'))$
{-3}	$\text{data}(\text{linear_resolve}(a'_2, \text{Write})(s')) + \text{offset}(x') - \text{offset}(\text{data}(\text{linear_resolve}(a'_2, \text{Write})(s')))$ $=$ $\text{data}(\text{linear_resolve}(a'_2 + \text{offset}(x') - \text{offset}(\text{data}(\text{linear_resolve}(a'_2, \text{Write})(s'))),$ $\text{Write})$ (s')
{-4}	$\text{offset}(x') - \text{offset}(\text{data}(\text{linear_resolve}(a'_2, \text{Write})(s')))) \geq 0$
{-5}	$\text{is_linear_plain_memory}?(\text{pm}')$
{-6}	$\text{pm}' \text{'states}(s')$
{-7}	$\forall (x: \text{Memory_Address_4G}):$ $\text{virt_to_phys_range}(s', \text{restrict}[\text{Address}, \text{Memory_Address_4G}, \text{boolean}](\text{pm}' \text{'rw_addr}),$ $\text{singleton}(\text{Write}))$ (x) $\Rightarrow \text{restrict}[\text{Address}, \text{Memory_Address_4G}, \text{boolean}](\text{pm_phy}' \text{'rw_addr})(x)$
{-8}	$\text{Mem}?(a'_1 \text{'type_of})$
{-9}	$0 \leq a'_1 \text{'offset}$
{-10}	$a'_1 \text{'offset} < \text{max_linear_offset}$
{-11}	$\text{Mem}?(a'_2 \text{'type_of})$
{-12}	$0 \leq a'_2 \text{'offset}$
{-13}	$a'_2 \text{'offset} < \text{max_linear_offset}$
{-14}	$s'_1 > 0$
{-15}	$s'_2 > 0$
{-16}	$\text{pm}' \text{'rw_addr}(a'_2)$
{-17}	$a'_1 \leq a'_2$
{-18}	$a'_2 + s'_2 \leq a'_1 + s'_1$
{-19}	$\text{rem}(\text{expt}(2, \text{min_page}))(\text{offset}(a'_2)) + s'_2 \leq \text{expt}(2, \text{min_page})$
{-20}	$\forall (x: \text{Address}): \text{address_block}(a'_1, s'_1)(x) \Rightarrow \text{pm}' \text{'rw_addr}(x)$
{-21}	$\text{address_block}(\text{data}(\text{linear_resolve}(a'_2, \text{Write})(s')), s'_2)(x')$
{1}	$\text{data}(\text{linear_resolve}(a'_2, \text{Write})(s')) + (\text{offset}(x') - \text{offset}(\text{data}(\text{linear_resolve}(a'_2, \text{Write})(s'))))$ $= x'$
{2}	$\text{pm_phy}' \text{'rw_addr}(x')$

Expanding the definition of +,

Applying decompose-equality,

Expanding the definition of address_block,

which is trivially true.

This completes the proof of `same_page_address_block_write.1.1.1.2`.

same_page_address_block_write.1.1.2:

{-1}	$\text{rem}(\text{expt}(2, \text{min_page}))(\text{offset}(a'_2)) + \text{offset}(x') - \text{offset}(\text{data}(\text{linear_resolve}(a'_2, \text{Write})(s')))$ $< \text{expt}(2, \text{min_page})$
{-2}	$\text{offset}(x') - \text{offset}(\text{data}(\text{linear_resolve}(a'_2, \text{Write})(s')))) \geq 0$
{-3}	$\text{is_linear_plain_memory?}(\text{pm}')$
{-4}	$\text{pm}' \text{'states}(s')$
{-5}	$\forall (x: \text{Memory_Address_4G}):$ $\text{virt_to_phys_range}(s', \text{restrict}[\text{Address}, \text{Memory_Address_4G}, \text{boolean}](\text{pm}' \text{'rw_addr}),$ $\text{singleton}(\text{Write}))$ (x) $\Rightarrow \text{restrict}[\text{Address}, \text{Memory_Address_4G}, \text{boolean}](\text{pm_phy}' \text{'rw_addr})(x)$
{-6}	$\text{Mem?}(a'_1 \text{'type_of})$
{-7}	$0 \leq a'_1 \text{'offset}$
{-8}	$a'_1 \text{'offset} < \text{max_linear_offset}$
{-9}	$\text{Mem?}(a'_2 \text{'type_of})$
{-10}	$0 \leq a'_2 \text{'offset}$
{-11}	$a'_2 \text{'offset} < \text{max_linear_offset}$
{-12}	$s'_1 > 0$
{-13}	$s'_2 > 0$
{-14}	$\text{pm}' \text{'rw_addr}(a'_2)$
{-15}	$a'_1 \leq a'_2$
{-16}	$a'_2 + s'_2 \leq a'_1 + s'_1$
{-17}	$\text{rem}(\text{expt}(2, \text{min_page}))(\text{offset}(a'_2)) + s'_2 \leq \text{expt}(2, \text{min_page})$
{-18}	$\forall (x: \text{Address}): \text{address_block}(a'_1, s'_1)(x) \Rightarrow \text{pm}' \text{'rw_addr}(x)$
{-19}	$\text{address_block}(\text{data}(\text{linear_resolve}(a'_2, \text{Write})(s')), s'_2)(x')$
{1}	$\text{OK?}(\text{linear_resolve}(a'_2, \text{Write})(s'))$
{2}	$\text{restrict}[\text{Address}, \text{Memory_Address_4G}, \text{boolean}](\text{pm}' \text{'rw_addr})(a'_2)$
{3}	$\text{pm_phy}' \text{'rw_addr}(x')$

Expanding the definition of restrict,

which is trivially true.

This completes the proof of same_page_address_block_write.1.1.2.

same_page_address_block_write.1.2:

{-1}	$\begin{aligned} & \text{rem}(\text{expt}(2, \text{min_page}))(\text{offset}(a'_2)) + \text{offset}(x') - \text{offset}(\text{data}(\text{linear_resolve}(a'_2, \text{Write})(s'))) \\ & < \text{expt}(2, \text{min_page}) \\ & \wedge \text{OK}?(\text{linear_resolve}(a'_2, \text{Write})(s')) \\ & \supset \\ & \text{data}(\text{linear_resolve}(a'_2, \text{Write})(s')) + \text{offset}(x') - \text{offset}(\text{data}(\text{linear_resolve}(a'_2, \text{Write})(s'))) \\ & = \\ & \text{data}(\text{linear_resolve}(a'_2 + \text{offset}(x') - \text{offset}(\text{data}(\text{linear_resolve}(a'_2, \text{Write})(s'))), \\ & \quad \text{Write} \\ & \quad (s')) \end{aligned}$
{-2}	$\text{offset}(x') - \text{offset}(\text{data}(\text{linear_resolve}(a'_2, \text{Write})(s'))) \geq 0$
{-3}	$\text{is_linear_plain_memory}?(pm')$
{-4}	$pm' \text{'states}(s')$
{-5}	$\text{restrict}[\text{Address}, \text{Memory_Address_4G}, \text{boolean}](pm' \text{'rw_addr})(a'_2) \supset$ $\text{OK}?(\text{linear_resolve}(a'_2, \text{Write})(s'))$
{-6}	$\forall (x: \text{Memory_Address_4G}):$ $\text{virt_to_phys_range}(s', \text{restrict}[\text{Address}, \text{Memory_Address_4G}, \text{boolean}](pm' \text{'rw_addr}),$ $\quad \text{singleton}(\text{Write}))$ (x) $\Rightarrow \text{restrict}[\text{Address}, \text{Memory_Address_4G}, \text{boolean}](pm_phy \text{'rw_addr})(x)$
{-7}	$\text{Mem}?(a'_1 \text{'type_of})$
{-8}	$0 \leq a'_1 \text{'offset}$
{-9}	$a'_1 \text{'offset} < \text{max_linear_offset}$
{-10}	$\text{Mem}?(a'_2 \text{'type_of})$
{-11}	$0 \leq a'_2 \text{'offset}$
{-12}	$a'_2 \text{'offset} < \text{max_linear_offset}$
{-13}	$s'_1 > 0$
{-14}	$s'_2 > 0$
{-15}	$pm' \text{'rw_addr}(a'_2)$
{-16}	$a'_1 \leq a'_2$
{-17}	$a'_2 + s'_2 \leq a'_1 + s'_1$
{-18}	$\text{rem}(\text{expt}(2, \text{min_page}))(\text{offset}(a'_2)) + s'_2 \leq \text{expt}(2, \text{min_page})$
{-19}	$\forall (x: \text{Address}): \text{address_block}(a'_1, s'_1)(x) \Rightarrow pm' \text{'rw_addr}(x)$
{-20}	$\text{address_block}(\text{data}(\text{linear_resolve}(a'_2, \text{Write})(s')), s'_2)(x')$
{1}	$\text{rem}(\text{expt}(2, \text{min_page}))(\text{offset}(a'_2)) + (\text{offset}(x') - \text{offset}(\text{data}(\text{linear_resolve}(a'_2, \text{Write})(s'))))$ $< \text{expt}(2, \text{min_page})$
{2}	$pm_phy \text{'rw_addr}(x')$

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of same_page_address_block_write.1.2.

same_page_address_block_write.2:

{-1}	is_linear_plain_memory?(pm')
{-2}	pm' 'states(s')
{-3}	restrict[Address, Memory_Address_4G, boolean](pm' 'rw_addr)(a'_2) \supset OK?(linear_resolve(a'_2, Write)(s'))
{-4}	$\forall (x: \text{Memory_Address_4G}):$ virt_to_phys_range(s', restrict[Address, Memory_Address_4G, boolean](pm' 'rw_addr), singleton(Write)) (x) \Rightarrow restrict[Address, Memory_Address_4G, boolean](pm_phy' 'rw_addr)(x)
{-5}	Mem?(a'_1 'type_of)
{-6}	$0 \leq a'_1$ 'offset
{-7}	a'_1 'offset < max_linear_offset
{-8}	Mem?(a'_2 'type_of)
{-9}	$0 \leq a'_2$ 'offset
{-10}	a'_2 'offset < max_linear_offset
{-11}	$s'_1 > 0$
{-12}	$s'_2 > 0$
{-13}	pm' 'rw_addr(a'_2)
{-14}	$a'_1 \leq a'_2$
{-15}	$a'_2 + s'_2 \leq a'_1 + s'_1$
{-16}	rem(expt(2, min_page))(offset(a'_2)) + $s'_2 \leq$ expt(2, min_page)
{-17}	$\forall (x: \text{Address}):$ address_block(a'_1, s'_1)(x) \Rightarrow pm' 'rw_addr(x)
{-18}	address_block(data(linear_resolve(a'_2, Write)(s')), s'_2)(x')
{1}	offset(x') - offset(data(linear_resolve(a'_2, Write)(s'))) ≥ 0
{2}	pm_phy' 'rw_addr(x')

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of same_page_address_block_write.2.
Q.E.D.

C.117.74 Linear_Memory_Properties.split_linear_read_side_effects_states_TCC1

Terse proof for split_linear_read_side_effects_states_TCC1.

split_linear_read_side_effects_states_TCC1:

{1}	$\forall (\text{pm}: \text{Plain_Memory}[\text{Linear_memory}], a: \text{Memory_Address_4G}, \text{bl}: (\text{cons?}[\text{Byte}])):$ is_linear_plain_memory?(pm) \wedge (address_block(a, length(bl)) \subseteq (pm'ro_addr \cup pm'rw_addr)) \supset every[[Address, list[Byte]]] ($\lambda (t: [\text{Address}, \text{list}[\text{Byte}]]):$ Mem?(t'1'type_of) $\wedge 0 \leq t'1$ 'offset $\wedge t'1$ 'offset < max_linear_offset) (split(min_page, a, bl))
-----	--

Repeatedly Skolemizing and flattening,
Using lemma address_block_split_type,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of split_linear_read_side_effects_states_TCC1.
Q.E.D.

C.117.75**Linear_Memory_Properties.split_linear_read_side_effects_states**Terse proof for `split_linear_read_side_effects_states`.`split_linear_read_side_effects_states:`

<pre> {1} ∃ (pm: Plain_Memory[Linear_memory], a: Memory_Address_4G, bl: (cons?[Byte]]): is_linear_plain_memory?(pm) ∧ (address_block(a, length(bl)) ⊆ (pm'ro_addr ∪ pm'rw_addr)) ⊃ every(λ (e: [Memory_Address_4G, list[Byte]]): transformer_invariant?(pm'states, singleton(expr_2_super(linear_read_side_effect_in_page (e)))))) (split(min_page, a, bl)) </pre>

Repeatedly Skolemizing and flattening,

Using lemma `split_range`,Using lemma `split_no_null`,Using lemma `split_type`,Using lemma `split_pair_cross_size`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 3 subgoals:

`split_linear_read_side_effects_states.1:`

<pre> {-1} every(λ (a2: Address, bl2: list[Byte]): rem(expt(2, min_page))(offset(a2)) + length(bl2) ≤ expt(2, min_page)) (split(min_page, a', bl')) {-2} every(λ (t: [Address, list[Byte]]): type_of(t'1) = type_of(a') ∧ reg_base(min_linear)(type_of(a')) ≤ t'1 ∧ t'1 < reg_size(max_linear)(type_of(a')) (split(min_page, a', bl')) {-3} every(λ (a2: Address, bl2: list[Byte]): cons?(bl2))(split(min_page, a', bl')) {-4} every(λ (a2: Address, bl2: list[Byte]): a' ≤ a2 ∧ a2 + length(bl2) ≤ a' + length(bl')) (split(min_page, a', bl')) {-5} Mem?(a''type_of) {-6} 0 ≤ a''offset {-7} a''offset < max_linear_offset {-8} every(λ (x: number): number_field_pred(x) ∧ real_pred(x) ∧ rational_pred(x) ∧ integer_pred(x) ∧ x ≥ 0 ∧ x < max_byte) (bl') {-9} cons?[Byte](bl') {-10} is_linear_plain_memory?(pm') {-11} (address_block(a', length(bl')) ⊆ (pm'ro_addr ∪ pm'rw_addr)) {1} every(λ (e: [Memory_Address_4G, list[Byte]]): transformer_invariant?(pm'states, singleton(expr_2_super(linear_read_side_effect_in_page(e)))) (split(min_page, a', bl')) </pre>
--

Using lemma `every_conjunct_left`,

C.117 Proofs for Linear_Memory_Properties (challenge-linear.pvs)

Using lemma `every_conjunct_left`,

we get 3 subgoals:

split_linear_read_side_effects_states.1.1:

```

{-1} every(λ (t: [Address, list[Byte]]):
      cons?(t'2) ∧
      a' ≤ t'1 ∧ t'1 + length(t'2) ≤ a' + length(bl')
      (split(min_page, a', bl'))
    ∧
    every(λ (t: [Address, list[Byte]]):
      type_of(t'1) = type_of(a') ∧
      reg_base(min_linear)(type_of(a')) ≤ t'1 ∧
      t'1 < reg_size(max_linear)(type_of(a'))
      (split(min_page, a', bl'))
    ⊃
    every(λ (t_1: [Address, list[Byte]]):
      (cons?(t_1'2) ∧
      a' ≤ t_1'1 ∧ t_1'1 + length(t_1'2) ≤ a' + length(bl'))
      ∧
      type_of(t_1'1) = type_of(a') ∧
      reg_base(min_linear)(type_of(a')) ≤ t_1'1 ∧
      t_1'1 < reg_size(max_linear)(type_of(a'))
      (split(min_page, a', bl'))
    {-2} every(λ (a_2: Address, bl2: list[Byte]): cons?(bl2))(split(min_page, a', bl')) ∧
    every(λ (a_2: Address, bl2: list[Byte]):
      a' ≤ a_2 ∧ a_2 + length(bl2) ≤ a' + length(bl')
      (split(min_page, a', bl'))
    ⊃
    every(λ (t: [Address, list[Byte]]):
      cons?(t'2) ∧
      a' ≤ t'1 ∧ t'1 + length(t'2) ≤ a' + length(bl')
      (split(min_page, a', bl'))
    {-3} every(λ (a_2: Address, bl2: list[Byte]):
      rem(expt(2, min_page))(offset(a_2)) + length(bl2) ≤ expt(2, min_page))
      (split(min_page, a', bl'))
    {-4} every(λ (t: [Address, list[Byte]]):
      type_of(t'1) = type_of(a') ∧
      reg_base(min_linear)(type_of(a')) ≤ t'1 ∧
      t'1 < reg_size(max_linear)(type_of(a'))
      (split(min_page, a', bl'))
    {-5} every(λ (a_2: Address, bl2: list[Byte]): cons?(bl2))(split(min_page, a', bl'))
    {-6} every(λ (a_2: Address, bl2: list[Byte]):
      a' ≤ a_2 ∧ a_2 + length(bl2) ≤ a' + length(bl')
      (split(min_page, a', bl'))
    {-7} Mem?(a' type_of)
    {-8} 0 ≤ a' offset
    {-9} a' offset < max_linear_offset
    {-10} every(λ (x: number):
      number_field_pred(x) ∧
      real_pred(x) ∧
      rational_pred(x) ∧ integer_pred(x) ∧ x ≥ 0 ∧ x < max_byte)
      (bl')
    {-11} cons?[Byte](bl')
    {-12} is_linear_plain_memory?(pm')
    {-13} (address_block(a', length(bl')) ⊆ (pm'ro_addr ∪ pm'rw_addr))
  }
  {1} every(λ (e: [Memory_Address_4G, list[Byte]]):
    transformer_invariant?(pm' states,
      singleton(expr_2_super(linear_read_side_effect_in_page(e)))
    (split(min_page, a', bl'))

```

C.117 Proofs for Linear_Memory_Properties (challenge-linear.pvs)

Using lemma `every_conjunct_left`,

we get 3 subgoals:

split_linear_read_side_effects_states.1.1.1:

```

{-1} every(λ (t_1: [Address, list[Byte]]):
      cons?(t_1'2) ∧
      a' ≤ t_1'1 ∧
      t_1'1 + length(t_1'2) ≤ a' + length(bl') ∧
      type_of(t_1'1) = type_of(a') ∧
      reg_base(min_linear)(type_of(a')) ≤ t_1'1 ∧
      t_1'1 < reg_size(max_linear)(type_of(a')))
      (split(min_page, a', bl'))
  ∧
  every(λ (a_2: Address, bl2: list[Byte]):
      rem(expt(2, min_page))(offset(a_2)) + length(bl2) ≤
      expt(2, min_page))
      (split(min_page, a', bl'))
  ⊃
  every(λ (t: [Address, list[Byte]]):
      (cons?(t'2) ∧
      a' ≤ t'1 ∧
      t'1 + length(t'2) ≤ a' + length(bl') ∧
      type_of(t'1) = type_of(a') ∧
      reg_base(min_linear)(type_of(a')) ≤ t'1 ∧
      t'1 < reg_size(max_linear)(type_of(a')))
      ∧
      rem(expt(2, min_page))(offset(t'1)) + length(t'2) ≤
      expt(2, min_page))
      (split(min_page, a', bl'))
{-2} every(λ (t: [Address, list[Byte]]):
      cons?(t'2) ∧
      a' ≤ t'1 ∧ t'1 + length(t'2) ≤ a' + length(bl'))
      (split(min_page, a', bl'))
  ∧
  every(λ (t: [Address, list[Byte]]):
      type_of(t'1) = type_of(a') ∧
      reg_base(min_linear)(type_of(a')) ≤ t'1 ∧
      t'1 < reg_size(max_linear)(type_of(a')))
      (split(min_page, a', bl'))
  ⊃
  every(λ (t_1: [Address, list[Byte]]):
      (cons?(t_1'2) ∧
      a' ≤ t_1'1 ∧ t_1'1 + length(t_1'2) ≤ a' + length(bl'))
      ∧
      type_of(t_1'1) = type_of(a') ∧
      reg_base(min_linear)(type_of(a')) ≤ t_1'1 ∧
      t_1'1 < reg_size(max_linear)(type_of(a')))
      (split(min_page, a', bl'))
{-3} every(λ (a_2: Address, bl2: list[Byte]): cons?(bl2))(split(min_page, a', bl')) ∧
  every(λ (a_2: Address, bl2: list[Byte]):
      a' ≤ a_2 ∧ a_2 + length(bl2) ≤ a' + length(bl'))
      (split(min_page, a', bl'))
  ⊃
  every(λ (t: [Address, list[Byte]]):
      cons?(t'2) ∧
      a' ≤ t'1 ∧ t'1 + length(t'2) ≤ a' + length(bl'))
      (split(min_page, a', bl'))
{-4} every(λ (a_2: Address, bl2: list[Byte]):
      rem(expt(2, min_page))(offset(a_2)) + length(bl2) ≤ expt(2, min_page))
      (split(min_page, a', bl'))
{-5} every(λ (t: [Address, list[Byte]]):
      type_of(t'1) = type_of(a') ∧
      reg_base(min_linear)(type_of(a')) ≤ t'1 ∧
      t'1 < reg_size(max_linear)(type_of(a')))
      (split(min_page, a', bl'))
{-6} every(λ (a_2: Address, bl2: list[Byte]): cons?(bl2))(split(min_page, a', bl'))
{-7} every(λ (a_2: Address, bl2: list[Byte]):

```

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Hiding formulas: (-1 -3 -4 -5 -6 -7 -11),
Installing automatic rewrites from: max_linear Mem min_linear
Using lemma every_extend[[Address, list[Byte]], [Memory_Address_4G, list[Byte]]],
Using lemma every_implied,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Hiding formulas: (-1 -2 2),
Expanding the definition of transformer_invariant?,
Expanding the definition of linear_read_side_effect_in_page,
Repeatedly Skolemizing and flattening,
Expanding the definition of singleton,
Replacing using formula -9,
Hiding formulas: -9,
Installing automatic rewrites from: (result_pred! has_next_state! ##! expr_2_super!
expr_2_super_res!)

Case splitting on union(pm!1'ro_addr, pm!1'rw_addr)(t!1'1),

we get 2 subgoals:

split_linear_read_side_effects_states.1.1.1.1:

{-1}	union(pm'ro_addr, pm'rw_addr)(t'1)
{-2}	cons?(t'2)
{-3}	$a' \leq t'1$
{-4}	$t'1 + \text{length}(t'2) \leq a' + \text{length}(bl')$
{-5}	$\text{type_of}(t'1) = \text{type_of}(a')$
{-6}	$\text{reg_base}(\text{\#type_of} := \text{Mem_}, \text{offset} := 0\#)(\text{type_of}(a')) \leq t'1$
{-7}	$t'1 < \text{reg_size}(\text{\#type_of} := \text{Mem_}, \text{offset} := \text{max_linear_offset}\#)(\text{type_of}(a'))$
{-8}	$\text{rem}(\text{expt}(2, \text{min_page}))(\text{offset}(t'1)) + \text{length}(t'2) \leq \text{expt}(2, \text{min_page})$
{-9}	pm'states(s')
{-10}	Mem?(a'type_of)
{-11}	$0 \leq a'\text{offset}$
{-12}	$a'\text{offset} < \text{max_linear_offset}$
{-13}	cons?[Byte](bl')
{-14}	is_linear_plain_memory?(pm')
{-15}	$(\text{address_block}(a', \text{length}(bl'))) \subseteq (\text{pm}'\text{ro_addr} \cup \text{pm}'\text{rw_addr})$
{1}	$\text{result_pred}(\text{pm}'\text{states})$ $(\text{expr_2_super}(\lambda (s: \text{Linear_memory}[\text{Physical_memory}, \text{pm_phy}]):$ $(\text{linear_resolve}(t'1, \text{Read}) \#\#$ $(\lambda (\text{pa}: \text{Memory_Address_4G})$ $(\text{ns}: \text{Linear_memory}[\text{Physical_memory}, \text{pm_phy}]):$ $\text{memory_read_side_effect}(\text{pm_phy}'\text{mem})$ $(\text{pa}, t'2, \text{TRUE})(s)))$ $(s))$

Using lemma pm_linear_resolve_read_ok,

Case splitting on subset?(address_block(data(linear_resolve(t!1'1, Read)(s!1)), length(t!1'2)), union(pm_phy'ro_addr, pm_phy'rw_addr)),

we get 3 subgoals:

C Proof scripts

split_linear_read_side_effects_states.1.1.1.1.1:

<pre> {-1} {-2} {-3} {-4} {-5} {-6} {-7} {-8} {-9} {-10} {-11} {-12} {-13} {-14} {-15} {-16} {-17} </pre>	<pre> (address_block(data(linear_resolve(t'1, Read)(s')), length(t'2)) ⊆ (pm_phy'ro_addr ∪ pm_phy'rw is_linear_plain_memory?(pm') ⊃ OK?(linear_resolve(t'1, Read)(s')) union(pm'ro_addr, pm'rw_addr)(t'1) cons?(t'2) a' ≤ t'1 t'1 + length(t'2) ≤ a' + length(bl') type_of(t'1) = type_of(a') reg_base((#type_of := Mem_, offset := 0#))(type_of(a')) ≤ t'1 t'1 < reg_size((#type_of := Mem_, offset := max_linear_offset#))(type_of(a')) rem(expt(2, min_page))(offset(t'1)) + length(t'2) ≤ expt(2, min_page) pm'states(s') Mem?(a' type_of) 0 ≤ a' offset a' offset < max_linear_offset cons?[Byte](bl') is_linear_plain_memory?(pm') (address_block(a', length(bl')) ⊆ (pm'ro_addr ∪ pm'rw_addr)) </pre> <hr style="border: 0.5px solid black;"/> <pre> {1} result_pred(pm'states) (expr_2_super(λ (s: Linear_memory[Physical_memory, pm_phy]): (linear_resolve(t'1, Read) ## (λ (pa: Memory_Address_4G) (ns: Linear_memory[Physical_memory, pm_phy]): memory_read_side_effect(pm_phy' mem) (pa, t'2, TRUE)(s))) (s)) (s')) </pre>
---	--

Using lemma linear_resolve_states,

Using lemma pm_states,

Using lemma pm_plain_phy,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 2 subgoals:

split_linear_read_side_effects_states.1.1.1.1.1.1:

{-1}	is_linear_plain_memory?(pm')
{-2}	plain_memory?(pm_phy)
{-3}	pm' states = pm_phy states
{-4}	union(pm' ro_addr, pm' rw_addr)(t'1)
{-5}	pm' states(s')
{-6}	OK?(linear_resolve(t'1, Read)(s'))
{-7}	pm_phy states(state(linear_resolve(t'1, Read)(s')))
{-8}	(address_block(data(linear_resolve(t'1, Read)(s')), length(t'2)) \subseteq (pm_phy ro_addr \cup pm_phy rw_addr))
{-9}	cons?(t'2)
{-10}	$a' \leq t'1$
{-11}	$t'1 + \text{length}(t'2) \leq a' + \text{length}(bl')$
{-12}	type_of(t'1) = type_of(a')
{-13}	reg_base((#type_of := Mem_, offset := 0#))(type_of(a')) $\leq t'1$
{-14}	$t'1 < \text{reg_size}((\#type_of := \text{Mem_}, \text{offset} := \text{max_linear_offset\#}))(type_of(a'))$
{-15}	$\text{rem}(\text{expt}(2, \text{min_page}))(\text{offset}(t'1) + \text{length}(t'2)) \leq \text{expt}(2, \text{min_page})$
{-16}	OK?(memory_read_side_effect(pm_phy mem (data(linear_resolve(t'1, Read)(s')), t'2, TRUE)(s')))
{-17}	Mem?(a' type_of)
{-18}	$0 \leq a' \text{ offset}$
{-19}	$a' \text{ offset} < \text{max_linear_offset}$
{-20}	cons?[Byte](bl')
{-21}	(address_block(a', length(bl')) \subseteq (pm' ro_addr \cup pm' rw_addr))
{1}	pm' states (state(memory_read_side_effect(pm_phy mem (data(linear_resolve(t'1, Read)(s')), t'2, TRUE)(s'))))

Using lemma plain_memory_transformer_invariant_read_side_effects,

Using lemma super_transformer_invariant_next_ok,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Expanding the definition of memory_read_side_effect_super_transformers,

Instantiating the top quantifier in 1 with the terms: (data(linear_resolve(t!1, Read)(s!1)) t!1'2 TRUE),

Rewriting using subset_reflexive, matching in *,

This completes the proof of split_linear_read_side_effects_states.1.1.1.1.1.1.

C Proof scripts

`split_linear_read_side_effects_states.1.1.1.1.1.2:`

{-1}	<code>is_linear_plain_memory?(pm')</code>
{-2}	<code>plain_memory?(pm_phy)</code>
{-3}	<code>pm' 'states = pm_phy' 'states</code>
{-4}	<code>union(pm' 'ro_addr, pm' 'rw_addr)(t' '1)</code>
{-5}	<code>pm' 'states(s')</code>
{-6}	<code>OK?(linear_resolve(t' '1, Read)(s'))</code>
{-7}	<code>pm_phy' 'states(state(linear_resolve(t' '1, Read)(s')))</code>
{-8}	<code>(address_block(data(linear_resolve(t' '1, Read)(s')), length(t' '2)) \subseteq (pm_phy' 'ro_addr \cup pm_phy' 'rw_addr))</code>
{-9}	<code>cons?(t' '2)</code>
{-10}	<code>a' \leq t' '1</code>
{-11}	<code>t' '1 + length(t' '2) \leq a' + length(bl')</code>
{-12}	<code>type_of(t' '1) = type_of(a')</code>
{-13}	<code>reg_base((#type_of := Mem_, offset := 0#))(type_of(a')) \leq t' '1</code>
{-14}	<code>t' '1 < reg_size((#type_of := Mem_, offset := max_linear_offset#))(type_of(a'))</code>
{-15}	<code>rem(expt(2, min_page))(offset(t' '1)) + length(t' '2) \leq expt(2, min_page)</code>
{-16}	<code>Exception?(memory_read_side_effect(pm_phy' 'mem (data(linear_resolve(t' '1, Read)(s')), t' '2, TRUE)(s'))</code>
{-17}	<code>Mem?(a' 'type_of)</code>
{-18}	<code>0 \leq a' 'offset</code>
{-19}	<code>a' 'offset < max_linear_offset</code>
{-20}	<code>cons?[Byte](bl')</code>
{-21}	<code>(address_block(a', length(bl')) \subseteq (pm' 'ro_addr \cup pm' 'rw_addr))</code>
{1}	<code>pm' 'states (state(memory_read_side_effect(pm_phy' 'mem (data(linear_resolve(t' '1, Read)(s')), t' '2, TRUE)(s')))</code>

Using lemma `plain_memory_transformers_ok_read_side_effects_block`,

Using lemma `super_transformers_ok_ok`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Expanding the definition of `memory_read_side_effect_super_transformers`,

Instantiating the top quantifier in 1 with the terms: `(data(linear_resolve(t!1'1, Read)(s!1)) t!1'2 TRUE)`,

Rewriting using `subset_reflexive`, matching in `*`,

This completes the proof of `split_linear_read_side_effects_states.1.1.1.1.1.2`.

split_linear_read_side_effects_states.1.1.1.1.2:

{-1}	is_linear_plain_memory?(pm') \supset OK?(linear_resolve(t'1, Read)(s'))
{-2}	union(pm'ro_addr, pm'rw_addr)(t'1)
{-3}	cons?(t'2)
{-4}	$a' \leq t'1$
{-5}	$t'1 + \text{length}(t'2) \leq a' + \text{length}(bl')$
{-6}	$\text{type_of}(t'1) = \text{type_of}(a')$
{-7}	$\text{reg_base}(\text{\#type_of} := \text{Mem_}, \text{offset} := 0\#)(\text{type_of}(a')) \leq t'1$
{-8}	$t'1 < \text{reg_size}(\text{\#type_of} := \text{Mem_}, \text{offset} := \text{max_linear_offset}\#)(\text{type_of}(a'))$
{-9}	$\text{rem}(\text{expt}(2, \text{min_page}))(\text{offset}(t'1)) + \text{length}(t'2) \leq \text{expt}(2, \text{min_page})$
{-10}	pm'states(s')
{-11}	Mem?(a'type_of)
{-12}	$0 \leq a'\text{'offset}$
{-13}	$a'\text{'offset} < \text{max_linear_offset}$
{-14}	cons?[Byte](bl')
{-15}	is_linear_plain_memory?(pm')
{-16}	$(\text{address_block}(a', \text{length}(bl'))) \subseteq (\text{pm}'\text{ro_addr} \cup \text{pm}'\text{rw_addr})$
{1}	$(\text{address_block}(\text{data}(\text{linear_resolve}(t'1, \text{Read})(s')), \text{length}(t'2))) \subseteq (\text{pm_phy}'\text{ro_addr} \cup \text{pm_phy}'\text{rw_addr})$
{2}	$\text{result_pred}(\text{pm}'\text{states})$ $(\text{expr_2_super}(\lambda (s: \text{Linear_memory}[\text{Physical_memory}, \text{pm_phy}]):$ $\quad (\text{linear_resolve}(t'1, \text{Read}) \#\#$ $\quad (\lambda (\text{pa}: \text{Memory_Address_4G})$ $\quad \quad (\text{ns}: \text{Linear_memory}[\text{Physical_memory}, \text{pm_phy}]):$ $\quad \quad \text{memory_read_side_effect}(\text{pm_phy}'\text{mem})$ $\quad \quad (\text{pa}, t'2, \text{TRUE})(s)))$ $\quad (s))$ $(s'))$

Hiding formulas: 2,

Using lemma same_page_address_block_read,

we get 3 subgoals:

split_linear_read_side_effects_states.1.1.1.1.2.1:

{-1}	is_linear_plain_memory?(pm') \wedge pm' states(s') \wedge union(pm' ro_addr, pm' rw_addr)(t'1) \wedge $a' \leq t'1 \wedge$ $t'1 + \text{length}(t'2) \leq a' + \text{length}(bl') \wedge$ $\text{rem}(\text{expt}(2, \text{min_page}))(\text{offset}(t'1)) + \text{length}(t'2) \leq \text{expt}(2, \text{min_page})$ $\wedge (\text{address_block}(a', \text{length}(bl')) \subseteq (\text{pm}'\text{ro_addr} \cup \text{pm}'\text{rw_addr}))$ \supset $(\text{address_block}(\text{data}(\text{linear_resolve}(t'1, \text{Read})(s')), \text{length}(t'2)) \subseteq (\text{pm_phy}'\text{ro_addr} \cup \text{pm_phy}'\text{rw_addr}))$
{-2}	is_linear_plain_memory?(pm') \supset OK?(linear_resolve(t'1, Read)(s'))
{-3}	union(pm' ro_addr, pm' rw_addr)(t'1)
{-4}	cons?(t'2)
{-5}	$a' \leq t'1$
{-6}	$t'1 + \text{length}(t'2) \leq a' + \text{length}(bl')$
{-7}	type_of(t'1) = type_of(a')
{-8}	reg_base((#type_of := Mem_, offset := 0#))(type_of(a')) $\leq t'1$
{-9}	$t'1 < \text{reg_size}((\#type_of := \text{Mem_}, \text{offset} := \text{max_linear_offset\#})(\text{type_of}(a')))$
{-10}	$\text{rem}(\text{expt}(2, \text{min_page}))(\text{offset}(t'1)) + \text{length}(t'2) \leq \text{expt}(2, \text{min_page})$
{-11}	pm' states(s')
{-12}	Mem?(a' type_of)
{-13}	$0 \leq a' \text{ offset}$
{-14}	$a' \text{ offset} < \text{max_linear_offset}$
{-15}	cons?[Byte](bl')
{-16}	is_linear_plain_memory?(pm')
{-17}	$(\text{address_block}(a', \text{length}(bl')) \subseteq (\text{pm}'\text{ro_addr} \cup \text{pm}'\text{rw_addr}))$
{1}	$(\text{address_block}(\text{data}(\text{linear_resolve}(t'1, \text{Read})(s')), \text{length}(t'2)) \subseteq (\text{pm_phy}'\text{ro_addr} \cup \text{pm_phy}'\text{rw_addr}))$

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of split_linear_read_side_effects_states.1.1.1.1.2.1.

split_linear_read_side_effects_states.1.1.1.1.2.2:

{-1}	is_linear_plain_memory?(pm') \supset OK?(linear_resolve(t'1, Read)(s'))
{-2}	union(pm' ro_addr, pm' rw_addr)(t'1)
{-3}	cons?(t'2)
{-4}	$a' \leq t'1$
{-5}	$t'1 + \text{length}(t'2) \leq a' + \text{length}(bl')$
{-6}	type_of(t'1) = type_of(a')
{-7}	reg_base((#type_of := Mem_, offset := 0#))(type_of(a')) $\leq t'1$
{-8}	$t'1 < \text{reg_size}((\#type_of := \text{Mem_}, \text{offset} := \text{max_linear_offset\#})(\text{type_of}(a')))$
{-9}	$\text{rem}(\text{expt}(2, \text{min_page}))(\text{offset}(t'1)) + \text{length}(t'2) \leq \text{expt}(2, \text{min_page})$
{-10}	pm' states(s')
{-11}	Mem?(a' type_of)
{-12}	$0 \leq a' \text{ offset}$
{-13}	$a' \text{ offset} < \text{max_linear_offset}$
{-14}	cons?[Byte](bl')
{-15}	is_linear_plain_memory?(pm')
{-16}	$(\text{address_block}(a', \text{length}(bl')) \subseteq (\text{pm}'\text{ro_addr} \cup \text{pm}'\text{rw_addr}))$
{1}	$\text{length}[Byte](t'2) > 0$
{2}	$(\text{address_block}(\text{data}(\text{linear_resolve}(t'1, \text{Read})(s')), \text{length}(t'2)) \subseteq (\text{pm_phy}'\text{ro_addr} \cup \text{pm_phy}'\text{rw_addr}))$

Expanding the definition of length,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `split_linear_read_side_effects_states.1.1.1.1.2.2`.

`split_linear_read_side_effects_states.1.1.1.1.2.3`:

{-1}	$\text{is_linear_plain_memory?}(pm') \supset \text{OK?}(\text{linear_resolve}(t'1, \text{Read})(s'))$
{-2}	$\text{union}(pm' \text{ro_addr}, pm' \text{rw_addr})(t'1)$
{-3}	$\text{cons?}(t'2)$
{-4}	$a' \leq t'1$
{-5}	$t'1 + \text{length}(t'2) \leq a' + \text{length}(bl')$
{-6}	$\text{type_of}(t'1) = \text{type_of}(a')$
{-7}	$\text{reg_base}(\text{\#type_of := Mem_}, \text{offset := 0\#})(\text{type_of}(a')) \leq t'1$
{-8}	$t'1 < \text{reg_size}(\text{\#type_of := Mem_}, \text{offset := max_linear_offset\#})(\text{type_of}(a'))$
{-9}	$\text{rem}(\text{expt}(2, \text{min_page}))(\text{offset}(t'1)) + \text{length}(t'2) \leq \text{expt}(2, \text{min_page})$
{-10}	$pm' \text{states}(s')$
{-11}	$\text{Mem?}(a' \text{type_of})$
{-12}	$0 \leq a' \text{offset}$
{-13}	$a' \text{offset} < \text{max_linear_offset}$
{-14}	$\text{cons?}[\text{Byte}](bl')$
{-15}	$\text{is_linear_plain_memory?}(pm')$
{-16}	$(\text{address_block}(a', \text{length}(bl'))) \subseteq (pm' \text{ro_addr} \cup pm' \text{rw_addr})$
{1}	$\text{length}[\text{Byte}](bl') > 0$
{2}	$(\text{address_block}(\text{data}(\text{linear_resolve}(t'1, \text{Read})(s')), \text{length}(t'2))) \subseteq (pm_phy \text{ro_addr} \cup pm_phy \text{rw_addr})$

Expanding the definition of `length`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `split_linear_read_side_effects_states.1.1.1.1.2.3`.

`split_linear_read_side_effects_states.1.1.1.1.3`:

{-1}	$\text{is_linear_plain_memory?}(pm') \supset \text{OK?}(\text{linear_resolve}(t'1, \text{Read})(s'))$
{-2}	$\text{union}(pm' \text{ro_addr}, pm' \text{rw_addr})(t'1)$
{-3}	$\text{cons?}(t'2)$
{-4}	$a' \leq t'1$
{-5}	$t'1 + \text{length}(t'2) \leq a' + \text{length}(bl')$
{-6}	$\text{type_of}(t'1) = \text{type_of}(a')$
{-7}	$\text{reg_base}(\text{\#type_of := Mem_}, \text{offset := 0\#})(\text{type_of}(a')) \leq t'1$
{-8}	$t'1 < \text{reg_size}(\text{\#type_of := Mem_}, \text{offset := max_linear_offset\#})(\text{type_of}(a'))$
{-9}	$\text{rem}(\text{expt}(2, \text{min_page}))(\text{offset}(t'1)) + \text{length}(t'2) \leq \text{expt}(2, \text{min_page})$
{-10}	$pm' \text{states}(s')$
{-11}	$\text{Mem?}(a' \text{type_of})$
{-12}	$0 \leq a' \text{offset}$
{-13}	$a' \text{offset} < \text{max_linear_offset}$
{-14}	$\text{cons?}[\text{Byte}](bl')$
{-15}	$\text{is_linear_plain_memory?}(pm')$
{-16}	$(\text{address_block}(a', \text{length}(bl'))) \subseteq (pm' \text{ro_addr} \cup pm' \text{rw_addr})$
{1}	$\text{OK?}[\text{Physical_memory}, \text{Address}](\text{linear_resolve}[\text{Physical_memory}, pm_phy](t'1, \text{Read})(s'))$
{2}	$\text{result_pred}(pm' \text{states})$
	$(\text{expr_2_super}(\lambda (s: \text{Linear_memory}[\text{Physical_memory}, pm_phy]):$
	$(\text{linear_resolve}(t'1, \text{Read}) \text{\#\#}$
	$(\lambda (pa: \text{Memory_Address_4G})$
	$(\text{ns: Linear_memory}[\text{Physical_memory}, pm_phy]):$
	$\text{memory_read_side_effect}(pm_phy \text{mem}$
	$(pa, t'2, \text{TRUE})(s))$
	$(s))$
	$(s'))$

C Proof scripts

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `split_linear_read_side_effects_states.1.1.1.1.3`.

`split_linear_read_side_effects_states.1.1.1.2`:

<pre> {-1} cons?(t'2) {-2} a' ≤ t'1 {-3} t'1 + length(t'2) ≤ a' + length(bl') {-4} type_of(t'1) = type_of(a') {-5} reg_base((#type_of := Mem_, offset := 0#))(type_of(a')) ≤ t'1 {-6} t'1 < reg_size((#type_of := Mem_, offset := max_linear_offset#))(type_of(a')) {-7} rem(expt(2, min_page))(offset(t'1)) + length(t'2) ≤ expt(2, min_page) {-8} pm' states(s') {-9} Mem?(a' type_of) {-10} 0 ≤ a' offset {-11} a' offset < max_linear_offset {-12} cons?[Byte](bl') {-13} is_linear_plain_memory?(pm') {-14} (address_block(a', length(bl')) ⊆ (pm'ro_addr ∪ pm'rw_addr)) </pre>	<pre> {1} union(pm'ro_addr, pm'rw_addr)(t'1) {2} result_pred(pm' states) (expr_2_super(λ (s: Linear_memory[Physical_memory, pm_phy]): (linear_resolve(t'1, Read) ## (λ (pa: Memory_Address_4G) (ns: Linear_memory[Physical_memory, pm_phy]): memory_read_side_effect(pm_phy mem) (pa, t'2, TRUE)(s))) (s)) (s')) </pre>
--	--

Hiding formulas: (-7 -13 2),

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `split_linear_read_side_effects_states.1.1.1.2`.

split_linear_read_side_effects_states.1.1.2:

```

{-1} every(λ (t: [Address, list[Byte]]):
      cons?(t'2) ∧
      a' ≤ t'1 ∧ t'1 + length(t'2) ≤ a' + length(bl')
      (split(min_page, a', bl'))
      ∧
      every(λ (t: [Address, list[Byte]]):
            type_of(t'1) = type_of(a') ∧
            reg_base(min_linear)(type_of(a')) ≤ t'1 ∧
            t'1 < reg_size(max_linear)(type_of(a'))
            (split(min_page, a', bl'))
            ⊃
            every(λ (t_1: [Address, list[Byte]]):
                  (cons?(t_1'2) ∧
                   a' ≤ t_1'1 ∧ t_1'1 + length(t_1'2) ≤ a' + length(bl'))
                  ∧
                  type_of(t_1'1) = type_of(a') ∧
                  reg_base(min_linear)(type_of(a')) ≤ t_1'1 ∧
                  t_1'1 < reg_size(max_linear)(type_of(a'))
                  (split(min_page, a', bl'))
            )
      )
      ⊃
      every(λ (a_2: Address, bl2: list[Byte]): cons?(bl2))(split(min_page, a', bl')) ∧
      every(λ (a_2: Address, bl2: list[Byte]):
            a' ≤ a_2 ∧ a_2 + length(bl2) ≤ a' + length(bl')
            (split(min_page, a', bl'))
            ⊃
            every(λ (t: [Address, list[Byte]]):
                  cons?(t'2) ∧
                  a' ≤ t'1 ∧ t'1 + length(t'2) ≤ a' + length(bl')
                  (split(min_page, a', bl'))
            )
      )
      )
{-2} every(λ (a_2: Address, bl2: list[Byte]): cons?(bl2))(split(min_page, a', bl')) ∧
      every(λ (a_2: Address, bl2: list[Byte]):
            a' ≤ a_2 ∧ a_2 + length(bl2) ≤ a' + length(bl')
            (split(min_page, a', bl'))
            ⊃
            every(λ (t: [Address, list[Byte]]):
                  cons?(t'2) ∧
                  a' ≤ t'1 ∧ t'1 + length(t'2) ≤ a' + length(bl')
                  (split(min_page, a', bl'))
            )
      )
      )
{-3} every(λ (a_2: Address, bl2: list[Byte]):
      rem(expt(2, min_page))(offset(a_2)) + length(bl2) ≤ expt(2, min_page))
      (split(min_page, a', bl'))
{-4} every(λ (t: [Address, list[Byte]]):
      type_of(t'1) = type_of(a') ∧
      reg_base(min_linear)(type_of(a')) ≤ t'1 ∧
      t'1 < reg_size(max_linear)(type_of(a'))
      (split(min_page, a', bl'))
      )
{-5} every(λ (a_2: Address, bl2: list[Byte]): cons?(bl2))(split(min_page, a', bl'))
{-6} every(λ (a_2: Address, bl2: list[Byte]):
      a' ≤ a_2 ∧ a_2 + length(bl2) ≤ a' + length(bl')
      (split(min_page, a', bl'))
      )
{-7} Mem?(a' type_of)
{-8} 0 ≤ a' offset
{-9} a' offset < max_linear_offset
{-10} every(λ (x: number):
      number_field_pred(x) ∧
      real_pred(x) ∧
      rational_pred(x) ∧ integer_pred(x) ∧ x ≥ 0 ∧ x < max_byte)
      (bl')
{-11} cons?[Byte](bl')
{-12} is_linear_plain_memory?(pm')
{-13} (address_block(a', length(bl')) ⊆ (pm'ro_addr ∪ pm'rw_addr))
-----
{1} ∀ (t_1: [Address, list[Byte]]):
      reg_base(min_linear)(type_of(a')) ≤ t_1'1 ∧
      type_of(t_1'1) = type_of(a') ∧
      t_1'1 + length(t_1'2) ≤ a' + length(bl') ∧
      a' ≤ t_1'1 ∧ cons?(t_1'2)
      ⊃ Mem?(max_linear type_of)
{2} every(λ (e: [Memory_Address_4G, list[Byte]]):
      transformer_invariant?(pm' states,
                              singleton(expr_2_super(linear_read_side_effect_in_page(e))))
      (split(min_page, a', bl'))

```


C Proof scripts

Expanding the definition of `max_linear`,

Expanding the definition of `Mem`,

which is trivially true.

This completes the proof of `split_linear_read_side_effects_states.1.1.2`.

split_linear_read_side_effects_states.1.1.3:

```

{-1} every(λ (t: [Address, list[Byte]]):
      cons?(t'2) ∧
      a' ≤ t'1 ∧ t'1 + length(t'2) ≤ a' + length(bl')
      (split(min_page, a', bl'))
    ∧
    every(λ (t: [Address, list[Byte]]):
      type_of(t'1) = type_of(a') ∧
      reg_base(min_linear)(type_of(a')) ≤ t'1 ∧
      t'1 < reg_size(max_linear)(type_of(a'))
      (split(min_page, a', bl'))
    ⊃
    every(λ (t_1: [Address, list[Byte]]):
      (cons?(t_1'2) ∧
      a' ≤ t_1'1 ∧ t_1'1 + length(t_1'2) ≤ a' + length(bl'))
      ∧
      type_of(t_1'1) = type_of(a') ∧
      reg_base(min_linear)(type_of(a')) ≤ t_1'1 ∧
      t_1'1 < reg_size(max_linear)(type_of(a'))
      (split(min_page, a', bl'))
    {-2} every(λ (a2: Address, bl2: list[Byte]): cons?(bl2))(split(min_page, a', bl')) ∧
    every(λ (a2: Address, bl2: list[Byte]):
      a' ≤ a2 ∧ a2 + length(bl2) ≤ a' + length(bl')
      (split(min_page, a', bl'))
    ⊃
    every(λ (t: [Address, list[Byte]]):
      cons?(t'2) ∧
      a' ≤ t'1 ∧ t'1 + length(t'2) ≤ a' + length(bl')
      (split(min_page, a', bl'))
    {-3} every(λ (a2: Address, bl2: list[Byte]):
      rem(expt(2, min_page))(offset(a2)) + length(bl2) ≤ expt(2, min_page))
      (split(min_page, a', bl'))
    {-4} every(λ (t: [Address, list[Byte]]):
      type_of(t'1) = type_of(a') ∧
      reg_base(min_linear)(type_of(a')) ≤ t'1 ∧
      t'1 < reg_size(max_linear)(type_of(a'))
      (split(min_page, a', bl'))
    {-5} every(λ (a2: Address, bl2: list[Byte]): cons?(bl2))(split(min_page, a', bl'))
    {-6} every(λ (a2: Address, bl2: list[Byte]):
      a' ≤ a2 ∧ a2 + length(bl2) ≤ a' + length(bl')
      (split(min_page, a', bl'))
    {-7} Mem?(a' type_of)
    {-8} 0 ≤ a' offset
    {-9} a' offset < max_linear_offset
    {-10} every(λ (x: number):
      number_field_pred(x) ∧
      real_pred(x) ∧
      rational_pred(x) ∧ integer_pred(x) ∧ x ≥ 0 ∧ x < max_byte)
      (bl')
    {-11} cons?[Byte](bl')
    {-12} is_linear_plain_memory?(pm')
    {-13} (address_block(a', length(bl')) ⊆ (pm'ro_addr ∪ pm'rw_addr))
  }
  {1} ∀ (t_1: [Address, list[Byte]]):
    type_of(t_1'1) = type_of(a') ∧
    t_1'1 + length(t_1'2) ≤ a' + length(bl') ∧
    a' ≤ t_1'1 ∧ cons?(t_1'2)
    ⊃ Mem?(min_linear type_of)
  {2} every(λ (e: [Memory_Address_4G, list[Byte]]):
    transformer_invariant?(pm' states,
      singleton(expr_2_super(linear_read_side_effect_in_page(e))))
    (split(min_page, a', bl'))

```

C Proof scripts

Expanding the definition of `min_linear`,

Expanding the definition of `Mem`,

which is trivially true.

This completes the proof of `split_linear_read_side_effects_states.1.1.3`.

`split_linear_read_side_effects_states.1.2`:

<div style="display: flex; flex-direction: column; gap: 5px;"> <div style="display: flex; align-items: flex-start;"> {-1} <div style="flex-grow: 1;"> $\text{every}(\lambda (a_2: \text{Address}, \text{bl2}: \text{list}[\text{Byte}]): \text{cons}?(a_2))(\text{split}(\text{min_page}, a', \text{bl}')) \wedge$ $\text{every}(\lambda (a_2: \text{Address}, \text{bl2}: \text{list}[\text{Byte}]):$ $a' \leq a_2 \wedge a_2 + \text{length}(\text{bl2}) \leq a' + \text{length}(\text{bl}'))$ $(\text{split}(\text{min_page}, a', \text{bl}'))$ \supset $\text{every}(\lambda (t: [\text{Address}, \text{list}[\text{Byte}]]):$ $\text{cons}?(t'2) \wedge$ $a' \leq t'1 \wedge t'1 + \text{length}(t'2) \leq a' + \text{length}(\text{bl}'))$ $(\text{split}(\text{min_page}, a', \text{bl}'))$ </div> </div> <div style="display: flex; align-items: flex-start;"> {-2} <div style="flex-grow: 1;"> $\text{every}(\lambda (a_2: \text{Address}, \text{bl2}: \text{list}[\text{Byte}]):$ $\text{rem}(\text{expt}(2, \text{min_page}))(\text{offset}(a_2)) + \text{length}(\text{bl2}) \leq \text{expt}(2, \text{min_page}))$ $(\text{split}(\text{min_page}, a', \text{bl}'))$ </div> </div> <div style="display: flex; align-items: flex-start;"> {-3} <div style="flex-grow: 1;"> $\text{every}(\lambda (t: [\text{Address}, \text{list}[\text{Byte}]]):$ $\text{type_of}(t'1) = \text{type_of}(a') \wedge$ $\text{reg_base}(\text{min_linear})(\text{type_of}(a')) \leq t'1 \wedge$ $t'1 < \text{reg_size}(\text{max_linear})(\text{type_of}(a'))$ $(\text{split}(\text{min_page}, a', \text{bl}'))$ </div> </div> <div style="display: flex; align-items: flex-start;"> {-4} <div style="flex-grow: 1;"> $\text{every}(\lambda (a_2: \text{Address}, \text{bl2}: \text{list}[\text{Byte}]): \text{cons}?(a_2))(\text{split}(\text{min_page}, a', \text{bl}'))$ </div> </div> <div style="display: flex; align-items: flex-start;"> {-5} <div style="flex-grow: 1;"> $\text{every}(\lambda (a_2: \text{Address}, \text{bl2}: \text{list}[\text{Byte}]):$ $a' \leq a_2 \wedge a_2 + \text{length}(\text{bl2}) \leq a' + \text{length}(\text{bl}'))$ $(\text{split}(\text{min_page}, a', \text{bl}'))$ </div> </div> <div style="display: flex; align-items: flex-start;"> {-6} <div style="flex-grow: 1;"> $\text{Mem}?(a' \text{ type_of})$ </div> </div> <div style="display: flex; align-items: flex-start;"> {-7} <div style="flex-grow: 1;"> $0 \leq a' \text{ offset}$ </div> </div> <div style="display: flex; align-items: flex-start;"> {-8} <div style="flex-grow: 1;"> $a' \text{ offset} < \text{max_linear_offset}$ </div> </div> <div style="display: flex; align-items: flex-start;"> {-9} <div style="flex-grow: 1;"> $\text{every}(\lambda (x: \text{number}):$ $\text{number_field_pred}(x) \wedge$ $\text{real_pred}(x) \wedge$ $\text{rational_pred}(x) \wedge \text{integer_pred}(x) \wedge x \geq 0 \wedge x < \text{max_byte})$ (bl') </div> </div> <div style="display: flex; align-items: flex-start;"> {-10} <div style="flex-grow: 1;"> $\text{cons}?[\text{Byte}](\text{bl}')$ </div> </div> <div style="display: flex; align-items: flex-start;"> {-11} <div style="flex-grow: 1;"> $\text{is_linear_plain_memory}?(pm')$ </div> </div> <div style="display: flex; align-items: flex-start;"> {-12} <div style="flex-grow: 1;"> $(\text{address_block}(a', \text{length}(\text{bl}')) \subseteq (pm' \text{ ro_addr} \cup pm' \text{ rw_addr}))$ </div> </div> <hr style="border: 0.5px solid black; margin: 5px 0;"/> <div style="display: flex; align-items: flex-start;"> {1} <div style="flex-grow: 1;"> $\forall (t: [\text{Address}, \text{list}[\text{Byte}]]):$ $\text{reg_base}(\text{min_linear})(\text{type_of}(a')) \leq t'1 \wedge \text{type_of}(t'1) = \text{type_of}(a') \supset$ $\text{Mem}?(a' \text{ type_of})$ </div> </div> <div style="display: flex; align-items: flex-start;"> {2} <div style="flex-grow: 1;"> $\text{every}(\lambda (e: [\text{Memory_Address_4G}, \text{list}[\text{Byte}]]):$ $\text{transformer_invariant}?(pm' \text{ states},$ $\text{singleton}(\text{expr_2_super}(\text{linear_read_side_effect_in_page}(e))))$ $(\text{split}(\text{min_page}, a', \text{bl}'))$ </div> </div> </div>
--

Expanding the definition of `max_linear`,

Expanding the definition of `Mem`,

which is trivially true.

This completes the proof of `split_linear_read_side_effects_states.1.2`.

split_linear_read_side_effects_states.1.3:

{-1}	$\text{every}(\lambda (a_2: \text{Address}, \text{bl2}: \text{list}[\text{Byte}]): \text{cons}?(a_2))(\text{split}(\text{min_page}, a', \text{bl}')) \wedge$ $\text{every}(\lambda (a_2: \text{Address}, \text{bl2}: \text{list}[\text{Byte}]):$ $a' \leq a_2 \wedge a_2 + \text{length}(\text{bl2}) \leq a' + \text{length}(\text{bl}'))$ $(\text{split}(\text{min_page}, a', \text{bl}'))$ \supset $\text{every}(\lambda (t: [\text{Address}, \text{list}[\text{Byte}]]):$ $\text{cons}?(t'2) \wedge$ $a' \leq t'1 \wedge t'1 + \text{length}(t'2) \leq a' + \text{length}(\text{bl}'))$ $(\text{split}(\text{min_page}, a', \text{bl}'))$
{-2}	$\text{every}(\lambda (a_2: \text{Address}, \text{bl2}: \text{list}[\text{Byte}]):$ $\text{rem}(\text{expt}(2, \text{min_page}))(\text{offset}(a_2)) + \text{length}(\text{bl2}) \leq \text{expt}(2, \text{min_page}))$ $(\text{split}(\text{min_page}, a', \text{bl}'))$
{-3}	$\text{every}(\lambda (t: [\text{Address}, \text{list}[\text{Byte}]]):$ $\text{type_of}(t'1) = \text{type_of}(a') \wedge$ $\text{reg_base}(\text{min_linear})(\text{type_of}(a')) \leq t'1 \wedge$ $t'1 < \text{reg_size}(\text{max_linear})(\text{type_of}(a'))))$ $(\text{split}(\text{min_page}, a', \text{bl}'))$
{-4}	$\text{every}(\lambda (a_2: \text{Address}, \text{bl2}: \text{list}[\text{Byte}]): \text{cons}?(a_2))(\text{split}(\text{min_page}, a', \text{bl}'))$
{-5}	$\text{every}(\lambda (a_2: \text{Address}, \text{bl2}: \text{list}[\text{Byte}]):$ $a' \leq a_2 \wedge a_2 + \text{length}(\text{bl2}) \leq a' + \text{length}(\text{bl}'))$ $(\text{split}(\text{min_page}, a', \text{bl}'))$
{-6}	$\text{Mem}?(a' \text{ 'type_of})$
{-7}	$0 \leq a' \text{ 'offset}$
{-8}	$a' \text{ 'offset} < \text{max_linear_offset}$
{-9}	$\text{every}(\lambda (x: \text{number}):$ $\text{number_field_pred}(x) \wedge$ $\text{real_pred}(x) \wedge$ $\text{rational_pred}(x) \wedge \text{integer_pred}(x) \wedge x \geq 0 \wedge x < \text{max_byte})$ (bl')
{-10}	$\text{cons}?[\text{Byte}](\text{bl}')$
{-11}	$\text{is_linear_plain_memory}?(pm')$
{-12}	$(\text{address_block}(a', \text{length}(\text{bl}')) \subseteq (pm' \text{ 'ro_addr} \cup pm' \text{ 'rw_addr}))$
{1}	$\forall (t: [\text{Address}, \text{list}[\text{Byte}]]):$ $\text{type_of}(t'1) = \text{type_of}(a') \supset \text{Mem}?(a' \text{ 'type_of})$
{2}	$\text{every}(\lambda (e: [\text{Memory_Address_4G}, \text{list}[\text{Byte}]]):$ $\text{transformer_invariant}?(pm' \text{ 'states},$ $\text{singleton}(\text{expr_2_super}(\text{linear_read_side_effect_in_page}(e))))))$ $(\text{split}(\text{min_page}, a', \text{bl}'))$

Expanding the definition of min_linear,

Expanding the definition of Mem,

which is trivially true.

This completes the proof of split_linear_read_side_effects_states.1.3.

C Proof scripts

`split_linear_read_side_effects_states.2:`

{-1}	every(λ (a_2 : Address, $bl2$: list[Byte]): $\text{rem}(\text{expt}(2, \text{min_page}))(\text{offset}(a_2)) + \text{length}(bl2) \leq \text{expt}(2, \text{min_page})$ $(\text{split}(\text{min_page}, a', bl'))$)
{-2}	every(λ (a_2 : Address, $bl2$: list[Byte]): $\text{cons?}(bl2)$) $(\text{split}(\text{min_page}, a', bl'))$)
{-3}	every(λ (a_2 : Address, $bl2$: list[Byte]): $a' \leq a_2 \wedge a_2 + \text{length}(bl2) \leq a' + \text{length}(bl')$ $(\text{split}(\text{min_page}, a', bl'))$)
{-4}	Mem? $(a'$, type_of)
{-5}	$0 \leq a'$, offset
{-6}	a' , offset < max_linear_offset
{-7}	every(λ (x : number): $\text{number_field_pred}(x) \wedge$ $\text{real_pred}(x) \wedge$ $\text{rational_pred}(x) \wedge \text{integer_pred}(x) \wedge x \geq 0 \wedge x < \text{max_byte}$ (bl'))
{-8}	$\text{cons?}[\text{Byte}](bl')$
{-9}	$\text{is_linear_plain_memory?}(pm')$
{-10}	$(\text{address_block}(a', \text{length}(bl'))) \subseteq (pm'$, ro_addr $\cup pm'$, rw_addr)
{1}	$a' + \text{length}(bl') \leq \text{reg_size}(\text{max_linear})(\text{type_of}(a'))$
{2}	every(λ (e : [Memory_Address_4G, list[Byte]]): $\text{transformer_invariant?}(pm'$, states, $\text{singleton}(\text{expr_2_super}(\text{linear_read_side_effect_in_page}(e)))$ $(\text{split}(\text{min_page}, a', bl'))$)

Hiding formulas: (-1 -2 -3 -7 2),

Expanding the definition of length,

Using lemma pm_memory_addr,

Simplifying, rewriting, and recording with decision procedures,

Hiding formulas: -6,

Expanding the definition of subset?,

Installing automatic rewrites from: Mem max_linear bus_width min_linear

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `split_linear_read_side_effects_states.2`.

split_linear_read_side_effects_states.3:

{-1}	every(λ (a_2 : Address, bl_2 : list[Byte]): rem(expt(2, min_page))(offset(a_2)) + length(bl_2) \leq expt(2, min_page)) (split(min_page, a' , bl'))
{-2}	every(λ (a_2 : Address, bl_2 : list[Byte]): cons?(bl_2))(split(min_page, a' , bl'))
{-3}	every(λ (a_2 : Address, bl_2 : list[Byte]): $a' \leq a_2 \wedge a_2 + \text{length}(bl_2) \leq a' + \text{length}(bl')$ (split(min_page, a' , bl'))
{-4}	Mem?(a' 'type_of)
{-5}	$0 \leq a'$ 'offset
{-6}	a' 'offset < max_linear_offset
{-7}	every(λ (x : number): number_field_pred(x) \wedge real_pred(x) \wedge rational_pred(x) \wedge integer_pred(x) $\wedge x \geq 0 \wedge x < \text{max_byte}$) (bl')
{-8}	cons?[Byte](bl')
{-9}	is_linear_plain_memory?(pm')
{-10}	(address_block(a' , length(bl'))) \subseteq (pm' 'ro_addr $\cup pm'$ 'rw_addr))
{1}	reg_base(min_linear)(type_of(a')) $\leq a'$
{2}	every(λ (e : [Memory_Address_4G, list[Byte]]): transformer_invariant?(pm' 'states, singleton(expr_2_super(linear_read_side_effect_in_page(e)))) (split(min_page, a' , bl'))

Expanding the definition of reg_base,

Expanding the definition of min_linear,

Expanding the definition of Mem,

Expanding the definition of <=,

which is trivially true.

This completes the proof of split_linear_read_side_effects_states.3.

Q.E.D.

C.117.76 Linear_Memory_Properties.split_linear_write_side_effects_states_TCC1

Terse proof for split_linear_write_side_effects_states_TCC1.

split_linear_write_side_effects_states_TCC1:

{1}	\forall (pm : Plain_Memory[Linear_memory], a : Memory_Address_4G, bl : (cons?[Byte])): is_linear_plain_memory?(pm) \wedge (address_block(a , length(bl))) $\subseteq pm'$ 'rw_addr) \supset every[[Address, list[Byte]]] (λ (t : [Address, list[Byte]]): Mem?(t' 'type_of) $\wedge 0 \leq t'$ 'offset $\wedge t'$ 'offset < max_linear_offset) (split(min_page, a , bl))
-----	---

Repeatedly Skolemizing and flattening,

Using lemma address_block_split_type,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Keeping (-7 1) and hiding *,

Rewriting using subset_transitive, matching in * where a gets address_block($a!1$, length($bl!1$)), b gets $pm!1'$ 'rw_addr, c gets union($pm!1'$ 'ro_addr, $pm!1'$ 'rw_addr),

C Proof scripts

Keeping (1) and hiding *,

Rewriting using union_commutative, matching in *,

Rewriting using union_subset1, matching in *,

This completes the proof of `split_linear_write_side_effects_states_TCC1`.

Q.E.D.

C.117.77

Linear_Memory_Properties.split_linear_write_side_effects_states

Terse proof for `split_linear_write_side_effects_states`.

```
split_linear_write_side_effects_states:
```

<pre>{1} ∃ (pm: Plain_Memory[Linear_memory], a: Memory_Address_4G, bl: (cons?[Byte])): is_linear_plain_memory?(pm) ∧ (address_block(a, length(bl)) ⊆ pm'rw_addr) ⊃ every(λ (e: [Memory_Address_4G, list[Byte]]): transformer_invariant?(pm' states, singleton(expr_2_super(linear_write_side_effect_in_page (e)))))) (split(min_page, a, bl))</pre>
--

Repeatedly Skolemizing and flattening,

Using lemma `split_range`,

Using lemma `split_no_null`,

Using lemma `split_type`,

Using lemma `split_pair_cross_size`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 3 subgoals:

split_linear_write_side_effects_states.1:

{-1}	every(λ (a_2 : Address, $bl2$: list[Byte]): rem(expt(2, min_page))(offset(a_2)) + length($bl2$) \leq expt(2, min_page)) (split(min_page, a' , bl'))
{-2}	every(λ (t : [Address, list[Byte]]): type_of($t'1$) = type_of(a') \wedge reg_base(min_linear)(type_of(a')) \leq $t'1$ \wedge $t'1$ < reg_size(max_linear)(type_of(a'))) (split(min_page, a' , bl'))
{-3}	every(λ (a_2 : Address, $bl2$: list[Byte]): cons?($bl2$))(split(min_page, a' , bl'))
{-4}	every(λ (a_2 : Address, $bl2$: list[Byte]): $a' \leq a_2$ \wedge a_2 + length($bl2$) \leq a' + length(bl')) (split(min_page, a' , bl'))
{-5}	Mem?(a' 'type_of)
{-6}	$0 \leq a'$ 'offset
{-7}	a' 'offset < max_linear_offset
{-8}	every(λ (x : number): number_field_pred(x) \wedge real_pred(x) \wedge rational_pred(x) \wedge integer_pred(x) \wedge $x \geq 0$ \wedge $x < \text{max_byte}$) (bl')
{-9}	cons?[Byte](bl')
{-10}	is_linear_plain_memory?(pm')
{-11}	(address_block(a' , length(bl')) \subseteq pm' 'rw_addr)
{1}	every(λ (e : [Memory_Address_4G, list[Byte]]): transformer_invariant?(pm' 'states, singleton(expr_2_super(linear_write_side_effect_in_page(e)))) (split(min_page, a' , bl'))

Using lemma every_conjunct_left,

Using lemma every_conjunct_left,

we get 3 subgoals:

split_linear_write_side_effects_states.1.1:

```

{-1} every(λ (t: [Address, list[Byte]]):
      cons?(t'2) ∧
      a' ≤ t'1 ∧ t'1 + length(t'2) ≤ a' + length(bl')
      (split(min_page, a', bl'))
    ∧
    every(λ (t: [Address, list[Byte]]):
      type_of(t'1) = type_of(a') ∧
      reg_base(min_linear)(type_of(a')) ≤ t'1 ∧
      t'1 < reg_size(max_linear)(type_of(a'))
      (split(min_page, a', bl'))
    ⊃
    every(λ (t_1: [Address, list[Byte]]):
      (cons?(t_1'2) ∧
      a' ≤ t_1'1 ∧ t_1'1 + length(t_1'2) ≤ a' + length(bl'))
      ∧
      type_of(t_1'1) = type_of(a') ∧
      reg_base(min_linear)(type_of(a')) ≤ t_1'1 ∧
      t_1'1 < reg_size(max_linear)(type_of(a'))
      (split(min_page, a', bl'))
    {-2} every(λ (a_2: Address, bl2: list[Byte]): cons?(bl2))(split(min_page, a', bl')) ∧
    every(λ (a_2: Address, bl2: list[Byte]):
      a' ≤ a_2 ∧ a_2 + length(bl2) ≤ a' + length(bl')
      (split(min_page, a', bl'))
    ⊃
    every(λ (t: [Address, list[Byte]]):
      cons?(t'2) ∧
      a' ≤ t'1 ∧ t'1 + length(t'2) ≤ a' + length(bl')
      (split(min_page, a', bl'))
    {-3} every(λ (a_2: Address, bl2: list[Byte]):
      rem(expt(2, min_page))(offset(a_2)) + length(bl2) ≤ expt(2, min_page))
      (split(min_page, a', bl'))
    {-4} every(λ (t: [Address, list[Byte]]):
      type_of(t'1) = type_of(a') ∧
      reg_base(min_linear)(type_of(a')) ≤ t'1 ∧
      t'1 < reg_size(max_linear)(type_of(a'))
      (split(min_page, a', bl'))
    {-5} every(λ (a_2: Address, bl2: list[Byte]): cons?(bl2))(split(min_page, a', bl'))
    {-6} every(λ (a_2: Address, bl2: list[Byte]):
      a' ≤ a_2 ∧ a_2 + length(bl2) ≤ a' + length(bl')
      (split(min_page, a', bl'))
    {-7} Mem?(a' type_of)
    {-8} 0 ≤ a' offset
    {-9} a' offset < max_linear_offset
    {-10} every(λ (x: number):
      number_field_pred(x) ∧
      real_pred(x) ∧
      rational_pred(x) ∧ integer_pred(x) ∧ x ≥ 0 ∧ x < max_byte)
      (bl')
    {-11} cons?[Byte](bl')
    {-12} is_linear_plain_memory?(pm')
    {-13} (address_block(a', length(bl')) ⊆ pm'rw_addr)
  }
  {1} every(λ (e: [Memory_Address_4G, list[Byte]]):
    transformer_invariant?(pm' states,
      singleton(expr_2_super(linear_write_side_effect_in_page(e)))
    (split(min_page, a', bl'))

```

Using lemma `every_conjunct_left`,

we get 3 subgoals:

split_linear_write_side_effects_states.1.1.1:

```

{-1} every(λ (t_1: [Address, list[Byte]]):
  cons?(t_1'2) ∧
  a' ≤ t_1'1 ∧
  t_1'1 + length(t_1'2) ≤ a' + length(bl') ∧
  type_of(t_1'1) = type_of(a') ∧
  reg_base(min_linear)(type_of(a')) ≤ t_1'1 ∧
  t_1'1 < reg_size(max_linear)(type_of(a')))
  (split(min_page, a', bl'))
  ∧
  every(λ (a_2: Address, bl2: list[Byte]):
    rem(expt(2, min_page))(offset(a_2)) + length(bl2) ≤
    expt(2, min_page))
    (split(min_page, a', bl'))
  ⊃
  every(λ (t: [Address, list[Byte]]):
    (cons?(t'2) ∧
     a' ≤ t'1 ∧
     t'1 + length(t'2) ≤ a' + length(bl') ∧
     type_of(t'1) = type_of(a') ∧
     reg_base(min_linear)(type_of(a')) ≤ t'1 ∧
     t'1 < reg_size(max_linear)(type_of(a')))
    ∧
    rem(expt(2, min_page))(offset(t'1)) + length(t'2) ≤
    expt(2, min_page))
    (split(min_page, a', bl'))
  {-2} every(λ (t: [Address, list[Byte]]):
    cons?(t'2) ∧
    a' ≤ t'1 ∧ t'1 + length(t'2) ≤ a' + length(bl')
    (split(min_page, a', bl'))
    ∧
    every(λ (t: [Address, list[Byte]]):
      type_of(t'1) = type_of(a') ∧
      reg_base(min_linear)(type_of(a')) ≤ t'1 ∧
      t'1 < reg_size(max_linear)(type_of(a')))
      (split(min_page, a', bl'))
    ⊃
    every(λ (t_1: [Address, list[Byte]]):
      (cons?(t_1'2) ∧
       a' ≤ t_1'1 ∧ t_1'1 + length(t_1'2) ≤ a' + length(bl'))
      ∧
      type_of(t_1'1) = type_of(a') ∧
      reg_base(min_linear)(type_of(a')) ≤ t_1'1 ∧
      t_1'1 < reg_size(max_linear)(type_of(a')))
      (split(min_page, a', bl'))
    {-3} every(λ (a_2: Address, bl2: list[Byte]): cons?(bl2))(split(min_page, a', bl')) ∧
    every(λ (a_2: Address, bl2: list[Byte]):
      a' ≤ a_2 ∧ a_2 + length(bl2) ≤ a' + length(bl')
      (split(min_page, a', bl'))
    ⊃
    every(λ (t: [Address, list[Byte]]):
      cons?(t'2) ∧
      a' ≤ t'1 ∧ t'1 + length(t'2) ≤ a' + length(bl')
      (split(min_page, a', bl'))
    {-4} every(λ (a_2: Address, bl2: list[Byte]):
      rem(expt(2, min_page))(offset(a_2)) + length(bl2) ≤ expt(2, min_page))
      (split(min_page, a', bl'))
    {-5} every(λ (t: [Address, list[Byte]]):
      type_of(t'1) = type_of(a') ∧
      reg_base(min_linear)(type_of(a')) ≤ t'1 ∧
      t'1 < reg_size(max_linear)(type_of(a')))
      (split(min_page, a', bl'))
    {-6} every(λ (a_2: Address, bl2: list[Byte]): cons?(bl2))(split(min_page, a', bl'))
    {-7} every(λ (a_2: Address, bl2: list[Byte]):

```

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Hiding formulas: (-1 -3 -4 -5 -6 -7 -11),
Installing automatic rewrites from: (min_linear! max_linear! Mem!)
Using lemma every_extend[[Address, list[Byte]], [Memory_Address_4G, list[Byte]]],
Using lemma every_implied,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Hiding formulas: (-1 -2 2),
Expanding the definition of transformer_invariant?,
Expanding the definition of linear_write_side_effect_in_page,
Repeatedly Skolemizing and flattening,
Expanding the definition of singleton,
Replacing using formula -9,
Hiding formulas: -9,
Installing automatic rewrites from: (result_pred! has_next_state! ##! expr_2_super!
expr_2_super_res!)

Case splitting on pm!1'rw_addr(t!1'1),

we get 2 subgoals:

split_linear_write_side_effects_states.1.1.1.1.1:

{-1}	pm'rw_addr(t'1)
{-2}	cons?(t'2)
{-3}	a' ≤ t'1
{-4}	t'1 + length(t'2) ≤ a' + length(bl')
{-5}	type_of(t'1) = type_of(a')
{-6}	reg_base((#type_of := Mem_, offset := 0#))(type_of(a')) ≤ t'1
{-7}	t'1 < reg_size((#type_of := Mem_, offset := max_linear_offset#))(type_of(a'))
{-8}	rem(expt(2, min_page))(offset(t'1)) + length(t'2) ≤ expt(2, min_page)
{-9}	pm'states(s')
{-10}	Mem?(a'type_of)
{-11}	0 ≤ a'offset
{-12}	a'offset < max_linear_offset
{-13}	cons?[Byte](bl')
{-14}	is_linear_plain_memory?(pm')
{-15}	(address_block(a', length(bl')) ⊆ pm'rw_addr)
{1}	result_pred(pm'states)
	(expr_2_super(λ (s: Linear_memory[Physical_memory, pm_phy]):
	(linear_resolve(t'1, Write) ##
	(λ (pa: Memory_Address_4G)
	(ns: Linear_memory[Physical_memory, pm_phy]):
	memory_write_side_effect(pm_phy'mem)
	(pa, t'2, TRUE)(s)))
	(s))

Using lemma pm_linear_resolve_write_ok,

Case splitting on subset?(address_block(data(linear_resolve(t!1'1, Write)(s!1)), length(t!1'2)), pm_phy'rw_addr),

we get 3 subgoals:

split_linear_write_side_effects_states.1.1.1.1.1:

<pre> {-1} (address_block(data(linear_resolve(t'1, Write)(s')), length(t'2)) ⊆ pm_phy'rw_addr) {-2} is_linear_plain_memory?(pm') ⊃ OK?(linear_resolve(t'1, Write)(s')) {-3} pm'rw_addr(t'1) {-4} cons?(t'2) {-5} a' ≤ t'1 {-6} t'1 + length(t'2) ≤ a' + length(bl') {-7} type_of(t'1) = type_of(a') {-8} reg_base((#type_of := Mem_, offset := 0#))(type_of(a')) ≤ t'1 {-9} t'1 < reg_size((#type_of := Mem_, offset := max_linear_offset#))(type_of(a')) {-10} rem(expt(2, min_page))(offset(t'1)) + length(t'2) ≤ expt(2, min_page) {-11} pm'states(s') {-12} Mem?(a'type_of) {-13} 0 ≤ a'offset {-14} a'offset < max_linear_offset {-15} cons?[Byte](bl') {-16} is_linear_plain_memory?(pm') {-17} (address_block(a', length(bl')) ⊆ pm'rw_addr) </pre>	<pre> {1} result_pred(pm'states) (expr_2_super(λ (s: Linear_memory[Physical_memory, pm_phy]): (linear_resolve(t'1, Write) ## (λ (pa: Memory_Address_4G) (ns: Linear_memory[Physical_memory, pm_phy]): memory_write_side_effect(pm_phy' mem) (pa, t'2, TRUE)(s))) (s)) </pre>
--	--

Using lemma linear_resolve_states,

Using lemma pm_states,

Using lemma pm_plain_phy,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 4 subgoals:

split_linear_write_side_effects_states.1.1.1.1.1.1:

{-1}	is_linear_plain_memory?(pm')
{-2}	plain_memory?(pm_phy)
{-3}	pm' 'states = pm_phy 'states
{-4}	pm' 'states(s')
{-5}	OK?(linear_resolve(t' '1, Write)(s'))
{-6}	pm_phy 'states(state(linear_resolve(t' '1, Write)(s')))
{-7}	(address_block(data(linear_resolve(t' '1, Write)(s')), length(t' '2)) \subseteq pm_phy 'rw_addr)
{-8}	pm' 'rw_addr(t' '1)
{-9}	cons?(t' '2)
{-10}	$a' \leq t' '1$
{-11}	$t' '1 + \text{length}(t' '2) \leq a' + \text{length}(bl')$
{-12}	type_of(t' '1) = type_of(a')
{-13}	reg_base((#type_of := Mem_, offset := 0#))(type_of(a')) $\leq t' '1$
{-14}	$t' '1 < \text{reg_size}((\#type_of := \text{Mem_}, \text{offset} := \text{max_linear_offset\#})(\text{type_of}(a')))$
{-15}	$\text{rem}(\text{expt}(2, \text{min_page}))(\text{offset}(t' '1) + \text{length}(t' '2)) \leq \text{expt}(2, \text{min_page})$
{-16}	OK?(memory_write_side_effect(pm_phy 'mem (data(linear_resolve(t' '1, Write)(s')), t' '2, TRUE)(s'))
{-17}	Mem?(a' 'type_of)
{-18}	$0 \leq a' \text{'offset}$
{-19}	$a' \text{'offset} < \text{max_linear_offset}$
{-20}	cons?[Byte](bl')
{-21}	(address_block(a', length(bl')) \subseteq pm' 'rw_addr)
{1}	pm' 'states (state(memory_write_side_effect(pm_phy 'mem (data(linear_resolve(t' '1, Write)(s')), t' '2, TRUE)(s')))

Using lemma plain_memory_transformer_invariant_write_side_effects,

Using lemma super_transformer_invariant_next_ok,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Expanding the definition of memory_write_side_effect_super_transformers,

Instantiating the top quantifier in 1 with the terms: (data(linear_resolve(t! '1, Write)(s!)) t! '2 TRUE),

Rewriting using subset_reflexive, matching in *,

This completes the proof of split_linear_write_side_effects_states.1.1.1.1.1.1.

C Proof scripts

`split_linear_write_side_effects_states.1.1.1.1.1.2:`

{-1}	<code>is_linear_plain_memory?(pm')</code>
{-2}	<code>plain_memory?(pm_phy)</code>
{-3}	<code>pm' 'states = pm_phy' 'states</code>
{-4}	<code>pm' 'states(s')</code>
{-5}	<code>OK?(linear_resolve(t' '1, Write)(s'))</code>
{-6}	<code>pm_phy' 'states(state(linear_resolve(t' '1, Write)(s')))</code>
{-7}	<code>(address_block(data(linear_resolve(t' '1, Write)(s')), length(t' '2)) \subseteq pm_phy' 'rw_addr)</code>
{-8}	<code>pm' 'rw_addr(t' '1)</code>
{-9}	<code>cons?(t' '2)</code>
{-10}	<code>a' \leq t' '1</code>
{-11}	<code>t' '1 + length(t' '2) \leq a' + length(bl')</code>
{-12}	<code>type_of(t' '1) = type_of(a')</code>
{-13}	<code>reg_base((#type_of := Mem_, offset := 0#))(type_of(a')) \leq t' '1</code>
{-14}	<code>t' '1 < reg_size((#type_of := Mem_, offset := max_linear_offset#))(type_of(a'))</code>
{-15}	<code>rem(expt(2, min_page))(offset(t' '1)) + length(t' '2) \leq expt(2, min_page)</code>
{-16}	<code>Exception?(memory_write_side_effect(pm_phy' 'mem (data(linear_resolve(t' '1, Write)(s')), t' '2, TRUE)(s'))</code>
{-17}	<code>Mem?(a' 'type_of)</code>
{-18}	<code>0 \leq a' 'offset</code>
{-19}	<code>a' 'offset < max_linear_offset</code>
{-20}	<code>cons?[Byte](bl')</code>
{-21}	<code>(address_block(a', length(bl')) \subseteq pm' 'rw_addr)</code>
{1}	<code>pm' 'states (state(memory_write_side_effect(pm_phy' 'mem (data(linear_resolve(t' '1, Write)(s')), t' '2, TRUE)(s')))</code>

Using lemma `plain_memory_transformers_ok_write_side_effects_block`,

Using lemma `super_transformers_ok_ok`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Expanding the definition of `memory_write_side_effect_super_transformers`,

Instantiating the top quantifier in 1 with the terms: `(data(linear_resolve(t!1'1, Write)(s!1)) t!1'2 TRUE)`,

Rewriting using `subset_reflexive`, matching in `*`,

This completes the proof of `split_linear_write_side_effects_states.1.1.1.1.1.2`.

split_linear_write_side_effects_states.1.1.1.1.1.3:

{-1}	is_linear_plain_memory?(pm')
{-2}	plain_memory?(pm_phy)
{-3}	pm' states = pm_phy states
{-4}	pm' states(s')
{-5}	OK?(linear_resolve(t'1, Write)(s'))
{-6}	(address_block(data(linear_resolve(t'1, Write)(s')), length(t'2)) \subseteq pm_phy rw_addr)
{-7}	pm' rw_addr(t'1)
{-8}	cons?(t'2)
{-9}	$a' \leq t'1$
{-10}	$t'1 + \text{length}(t'2) \leq a' + \text{length}(bl')$
{-11}	type_of(t'1) = type_of(a')
{-12}	reg_base((#type_of := Mem_, offset := 0#))(type_of(a')) $\leq t'1$
{-13}	$t'1 < \text{reg_size}((\#type_of := \text{Mem_}, \text{offset} := \text{max_linear_offset\#})(\text{type_of}(a')))$
{-14}	$\text{rem}(\text{expt}(2, \text{min_page}))(\text{offset}(t'1)) + \text{length}(t'2) \leq \text{expt}(2, \text{min_page})$
{-15}	OK?(memory_write_side_effect(pm_phy mem) (data(linear_resolve(t'1, Write)(s')), t'2, TRUE)(s'))
{-16}	Mem?(a' type_of)
{-17}	$0 \leq a' \text{ offset}$
{-18}	$a' \text{ offset} < \text{max_linear_offset}$
{-19}	cons?[Byte](bl')
{-20}	(address_block(a', length(bl')) \subseteq pm' rw_addr)
{1}	union(pm' ro_addr, pm' rw_addr)(t'1)
{2}	pm' states (state(memory_write_side_effect(pm_phy mem) (data(linear_resolve(t'1, Write)(s')), t'2, TRUE)(s')))

Keeping (-7 1) and hiding *,

Rewriting using union_right, matching in *,

This completes the proof of split_linear_write_side_effects_states.1.1.1.1.1.3.

split_linear_write_side_effects_states.1.1.1.1.1.4:

{-1}	is_linear_plain_memory?(pm')
{-2}	plain_memory?(pm_phy)
{-3}	pm' states = pm_phy states
{-4}	pm' states(s')
{-5}	OK?(linear_resolve(t'1, Write)(s'))
{-6}	(address_block(data(linear_resolve(t'1, Write)(s')), length(t'2)) \subseteq pm_phy rw_addr)
{-7}	pm' rw_addr(t'1)
{-8}	cons?(t'2)
{-9}	$a' \leq t'1$
{-10}	$t'1 + \text{length}(t'2) \leq a' + \text{length}(bl')$
{-11}	type_of(t'1) = type_of(a')
{-12}	reg_base((#type_of := Mem_, offset := 0#))(type_of(a')) $\leq t'1$
{-13}	$t'1 < \text{reg_size}((\#type_of := \text{Mem_}, \text{offset} := \text{max_linear_offset\#}))(type_of(a'))$
{-14}	$\text{rem}(\text{expt}(2, \text{min_page}))(\text{offset}(t'1)) + \text{length}(t'2) \leq \text{expt}(2, \text{min_page})$
{-15}	Exception?(memory_write_side_effect(pm_phy mem (data(linear_resolve(t'1, Write)(s')), t'2, TRUE)(s')))
{-16}	Mem?(a' type_of)
{-17}	$0 \leq a' \text{ offset}$
{-18}	$a' \text{ offset} < \text{max_linear_offset}$
{-19}	cons?[Byte](bl')
{-20}	(address_block(a', length(bl')) \subseteq pm' rw_addr)
{1}	union(pm' ro_addr, pm' rw_addr)(t'1)
{2}	pm' states (state(memory_write_side_effect(pm_phy mem (data(linear_resolve(t'1, Write)(s')), t'2, TRUE)(s')))

Rewriting using union_right, matching in *,

This completes the proof of split_linear_write_side_effects_states.1.1.1.1.1.4.

split_linear_write_side_effects_states.1.1.1.1.2:

{-1}	is_linear_plain_memory?(pm') \supset OK?(linear_resolve(t'1, Write)(s'))
{-2}	pm'rw_addr(t'1)
{-3}	cons?(t'2)
{-4}	$a' \leq t'1$
{-5}	$t'1 + \text{length}(t'2) \leq a' + \text{length}(bl')$
{-6}	$\text{type_of}(t'1) = \text{type_of}(a')$
{-7}	$\text{reg_base}(\text{\#type_of} := \text{Mem_}, \text{offset} := 0\#)(\text{type_of}(a')) \leq t'1$
{-8}	$t'1 < \text{reg_size}(\text{\#type_of} := \text{Mem_}, \text{offset} := \text{max_linear_offset}\#)(\text{type_of}(a'))$
{-9}	$\text{rem}(\text{expt}(2, \text{min_page}))(\text{offset}(t'1)) + \text{length}(t'2) \leq \text{expt}(2, \text{min_page})$
{-10}	pm'states(s')
{-11}	Mem?(a'type_of)
{-12}	$0 \leq a'\text{offset}$
{-13}	$a'\text{offset} < \text{max_linear_offset}$
{-14}	cons?[Byte](bl')
{-15}	is_linear_plain_memory?(pm')
{-16}	$(\text{address_block}(a', \text{length}(bl')) \subseteq \text{pm}'\text{rw_addr})$
{1}	$(\text{address_block}(\text{data}(\text{linear_resolve}(t'1, \text{Write})(s')), \text{length}(t'2)) \subseteq \text{pm_phy}'\text{rw_addr})$
{2}	$\text{result_pred}(\text{pm}'\text{states})$ $(\text{expr_2_super}(\lambda (s: \text{Linear_memory}[\text{Physical_memory}, \text{pm_phy}]):$ $(\text{linear_resolve}(t'1, \text{Write}) \#\#$ $(\lambda (\text{pa}: \text{Memory_Address_4G})$ $(\text{ns}: \text{Linear_memory}[\text{Physical_memory}, \text{pm_phy}]):$ $\text{memory_write_side_effect}(\text{pm_phy}'\text{mem})$ $(\text{pa}, t'2, \text{TRUE})(s)))$ $(s))$ (s')

Hiding formulas: 2,

Using lemma same_page_address_block_write,

we get 3 subgoals:

split_linear_write_side_effects_states.1.1.1.1.2.1:

{-1}	is_linear_plain_memory?(pm') \wedge pm'rw_addr(t'1) \wedge a' \leq t'1 \wedge t'1 + length(t'2) \leq a' + length(bl') \wedge rem(expt(2, min_page))(offset(t'1)) + length(t'2) \leq expt(2, min_page) \wedge (address_block(a', length(bl')) \subseteq pm'rw_addr)
	\supset (address_block(data(linear_resolve(t'1, Write)(s')), length(t'2)) \subseteq pm_phy'rw_addr)
{-2}	is_linear_plain_memory?(pm') \supset OK?(linear_resolve(t'1, Write)(s'))
{-3}	pm'rw_addr(t'1)
{-4}	cons?(t'2)
{-5}	a' \leq t'1
{-6}	t'1 + length(t'2) \leq a' + length(bl')
{-7}	type_of(t'1) = type_of(a')
{-8}	reg_base((#type_of := Mem_, offset := 0#))(type_of(a')) \leq t'1
{-9}	t'1 < reg_size((#type_of := Mem_, offset := max_linear_offset#))(type_of(a'))
{-10}	rem(expt(2, min_page))(offset(t'1)) + length(t'2) \leq expt(2, min_page)
{-11}	pm'rw_addr(t'1)
{-12}	Mem?(a' type_of)
{-13}	0 \leq a' offset
{-14}	a' offset < max_linear_offset
{-15}	cons?[Byte](bl')
{-16}	is_linear_plain_memory?(pm')
{-17}	(address_block(a', length(bl')) \subseteq pm'rw_addr)
{1}	(address_block(data(linear_resolve(t'1, Write)(s')), length(t'2)) \subseteq pm_phy'rw_addr)

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of split_linear_write_side_effects_states.1.1.1.1.2.1.

split_linear_write_side_effects_states.1.1.1.1.2.2:

{-1}	is_linear_plain_memory?(pm') \supset OK?(linear_resolve(t'1, Write)(s'))
{-2}	pm'rw_addr(t'1)
{-3}	cons?(t'2)
{-4}	a' \leq t'1
{-5}	t'1 + length(t'2) \leq a' + length(bl')
{-6}	type_of(t'1) = type_of(a')
{-7}	reg_base((#type_of := Mem_, offset := 0#))(type_of(a')) \leq t'1
{-8}	t'1 < reg_size((#type_of := Mem_, offset := max_linear_offset#))(type_of(a'))
{-9}	rem(expt(2, min_page))(offset(t'1)) + length(t'2) \leq expt(2, min_page)
{-10}	pm'rw_addr(t'1)
{-11}	Mem?(a' type_of)
{-12}	0 \leq a' offset
{-13}	a' offset < max_linear_offset
{-14}	cons?[Byte](bl')
{-15}	is_linear_plain_memory?(pm')
{-16}	(address_block(a', length(bl')) \subseteq pm'rw_addr)
{1}	length[Byte](t'2) > 0
{2}	(address_block(data(linear_resolve(t'1, Write)(s')), length(t'2)) \subseteq pm_phy'rw_addr)

Expanding the definition of length,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `split_linear_write_side_effects_states.1.1.1.1.2.2.`

`split_linear_write_side_effects_states.1.1.1.1.2.3:`

{-1}	<code>is_linear_plain_memory?(pm') \supset OK?(linear_resolve(t'1, Write)(s'))</code>
{-2}	<code>pm'rw_addr(t'1)</code>
{-3}	<code>cons?(t'2)</code>
{-4}	<code>a' \leq t'1</code>
{-5}	<code>t'1 + length(t'2) \leq a' + length(bl')</code>
{-6}	<code>type_of(t'1) = type_of(a')</code>
{-7}	<code>reg_base((#type_of := Mem_, offset := 0#))(type_of(a')) \leq t'1</code>
{-8}	<code>t'1 < reg_size((#type_of := Mem_, offset := max_linear_offset#))(type_of(a'))</code>
{-9}	<code>rem(expt(2, min_page))(offset(t'1)) + length(t'2) \leq expt(2, min_page)</code>
{-10}	<code>pm'states(s')</code>
{-11}	<code>Mem?(a'type_of)</code>
{-12}	<code>0 \leq a'offset</code>
{-13}	<code>a'offset < max_linear_offset</code>
{-14}	<code>cons?[Byte](bl')</code>
{-15}	<code>is_linear_plain_memory?(pm')</code>
{-16}	<code>(address_block(a', length(bl'))) \subseteq pm'rw_addr</code>
{1}	<code>length[Byte](bl') > 0</code>
{2}	<code>(address_block(data(linear_resolve(t'1, Write)(s')), length(t'2))) \subseteq pm_phyrw_addr</code>

Expanding the definition of length,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `split_linear_write_side_effects_states.1.1.1.1.2.3.`

`split_linear_write_side_effects_states.1.1.1.1.3:`

{-1}	<code>is_linear_plain_memory?(pm') \supset OK?(linear_resolve(t'1, Write)(s'))</code>
{-2}	<code>pm'rw_addr(t'1)</code>
{-3}	<code>cons?(t'2)</code>
{-4}	<code>a' \leq t'1</code>
{-5}	<code>t'1 + length(t'2) \leq a' + length(bl')</code>
{-6}	<code>type_of(t'1) = type_of(a')</code>
{-7}	<code>reg_base((#type_of := Mem_, offset := 0#))(type_of(a')) \leq t'1</code>
{-8}	<code>t'1 < reg_size((#type_of := Mem_, offset := max_linear_offset#))(type_of(a'))</code>
{-9}	<code>rem(expt(2, min_page))(offset(t'1)) + length(t'2) \leq expt(2, min_page)</code>
{-10}	<code>pm'states(s')</code>
{-11}	<code>Mem?(a'type_of)</code>
{-12}	<code>0 \leq a'offset</code>
{-13}	<code>a'offset < max_linear_offset</code>
{-14}	<code>cons?[Byte](bl')</code>
{-15}	<code>is_linear_plain_memory?(pm')</code>
{-16}	<code>(address_block(a', length(bl'))) \subseteq pm'rw_addr</code>
{1}	<code>OK?[Physical_memory, Address](linear_resolve[Physical_memory, pm_phy](t'1, Write)(s'))</code>
{2}	<code>result_pred(pm'states)</code> $\begin{aligned} & (\text{expr_2_super}(\lambda (s: \text{Linear_memory}[\text{Physical_memory}, \text{pm_phy}]): \\ & \quad (\text{linear_resolve}(t'1, \text{Write}) \#\# \\ & \quad \quad (\lambda (\text{pa}: \text{Memory_Address_4G}) \\ & \quad \quad \quad (\text{ns}: \text{Linear_memory}[\text{Physical_memory}, \text{pm_phy}]): \\ & \quad \quad \quad \text{memory_write_side_effect}(\text{pm_phy}'\text{mem}) \\ & \quad \quad \quad (\text{pa}, t'2, \text{TRUE})(s))) \\ & \quad (s)) \\ & (s')) \end{aligned}$

C Proof scripts

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `split_linear_write_side_effects_states.1.1.1.1.3`.

`split_linear_write_side_effects_states.1.1.1.2`:

{-1}	<code>cons?(t'2)</code>
{-2}	<code>a' ≤ t'1</code>
{-3}	<code>t'1 + length(t'2) ≤ a' + length(bl')</code>
{-4}	<code>type_of(t'1) = type_of(a')</code>
{-5}	<code>reg_base((#type_of := Mem_, offset := 0#))(type_of(a')) ≤ t'1</code>
{-6}	<code>t'1 < reg_size((#type_of := Mem_, offset := max_linear_offset#))(type_of(a'))</code>
{-7}	<code>rem(expt(2, min_page))(offset(t'1)) + length(t'2) ≤ expt(2, min_page)</code>
{-8}	<code>pm' states(s')</code>
{-9}	<code>Mem?(a' type_of)</code>
{-10}	<code>0 ≤ a' offset</code>
{-11}	<code>a' offset < max_linear_offset</code>
{-12}	<code>cons?[Byte](bl')</code>
{-13}	<code>is_linear_plain_memory?(pm')</code>
{-14}	<code>(address_block(a', length(bl')) ⊆ pm'rw_addr)</code>
{1}	<code>pm'rw_addr(t'1)</code>
{2}	<code>result_pred(pm' states</code> <div style="margin-left: 40px;"><code>(expr_2_super(λ (s: Linear_memory[Physical_memory, pm_phy]):</code> <div style="margin-left: 40px;"><code>(linear_resolve(t'1, Write) ##</code> <div style="margin-left: 40px;"><code>(λ (pa: Memory_Address_4G)</code> <div style="margin-left: 40px;"><code>(ns: Linear_memory[Physical_memory, pm_phy]):</code> <div style="margin-left: 40px;"><code>memory_write_side_effect(pm_phy mem)</code> <div style="margin-left: 40px;"><code>(pa, t'2, TRUE)(s)))</code></div></div></div></div></div></div>

`(s))``(s'))`

Hiding formulas: (-7 -13 2),

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `split_linear_write_side_effects_states.1.1.1.2`.

split_linear_write_side_effects_states.1.1.2:

```

{-1} every(λ (t: [Address, list[Byte]]):
      cons?(t'2) ∧
      a' ≤ t'1 ∧ t'1 + length(t'2) ≤ a' + length(bl')
      (split(min_page, a', bl'))
      ∧
      every(λ (t: [Address, list[Byte]]):
            type_of(t'1) = type_of(a') ∧
            reg_base(min_linear)(type_of(a')) ≤ t'1 ∧
            t'1 < reg_size(max_linear)(type_of(a'))
            (split(min_page, a', bl'))
            ⊃
            every(λ (t_1: [Address, list[Byte]]):
                  (cons?(t_1'2) ∧
                   a' ≤ t_1'1 ∧ t_1'1 + length(t_1'2) ≤ a' + length(bl'))
                  ∧
                  type_of(t_1'1) = type_of(a') ∧
                  reg_base(min_linear)(type_of(a')) ≤ t_1'1 ∧
                  t_1'1 < reg_size(max_linear)(type_of(a'))
                  (split(min_page, a', bl'))
            )
      )
      ⊃
      every(λ (a_2: Address, bl2: list[Byte]): cons?(bl2))(split(min_page, a', bl')) ∧
      every(λ (a_2: Address, bl2: list[Byte]):
            a' ≤ a_2 ∧ a_2 + length(bl2) ≤ a' + length(bl')
            (split(min_page, a', bl'))
            ⊃
            every(λ (t: [Address, list[Byte]]):
                  cons?(t'2) ∧
                  a' ≤ t'1 ∧ t'1 + length(t'2) ≤ a' + length(bl')
                  (split(min_page, a', bl'))
            )
      )
      ⊃
      every(λ (a_2: Address, bl2: list[Byte]):
            rem(expt(2, min_page))(offset(a_2)) + length(bl2) ≤ expt(2, min_page)
            (split(min_page, a', bl'))
      )
      ⊃
      every(λ (t: [Address, list[Byte]]):
            type_of(t'1) = type_of(a') ∧
            reg_base(min_linear)(type_of(a')) ≤ t'1 ∧
            t'1 < reg_size(max_linear)(type_of(a'))
            (split(min_page, a', bl'))
      )
      ⊃
      every(λ (a_2: Address, bl2: list[Byte]): cons?(bl2))(split(min_page, a', bl'))
      ⊃
      every(λ (a_2: Address, bl2: list[Byte]):
            a' ≤ a_2 ∧ a_2 + length(bl2) ≤ a' + length(bl')
            (split(min_page, a', bl'))
      )
      ⊃
      Mem?(a' type_of)
      ⊃
      0 ≤ a' offset
      ⊃
      a' offset < max_linear_offset
      ⊃
      every(λ (x: number):
            number_field_pred(x) ∧
            real_pred(x) ∧
            rational_pred(x) ∧ integer_pred(x) ∧ x ≥ 0 ∧ x < max_byte
            (bl')
      )
      ⊃
      cons?[Byte](bl')
      ⊃
      is_linear_plain_memory?(pm')
      ⊃
      (address_block(a', length(bl')) ⊆ pm'rw_addr)
      ⊃
      {1} ∀ (t_1: [Address, list[Byte]]):
            reg_base(min_linear)(type_of(a')) ≤ t_1'1 ∧
            type_of(t_1'1) = type_of(a') ∧
            t_1'1 + length(t_1'2) ≤ a' + length(bl') ∧
            a' ≤ t_1'1 ∧ cons?(t_1'2)
            ⊃
            Mem?(max_linear type_of)
      )
      ⊃
      {2} every(λ (e: [Memory_Address_4G, list[Byte]]):
            transformer_invariant?(pm' states,
            singleton(expr_2_super(linear_write_side_effect_in_page(e))))
            (split(min_page, a', bl'))

```

C Proof scripts

Expanding the definition of `max_linear`,

Expanding the definition of `Mem`,

which is trivially true.

This completes the proof of `split_linear_write_side_effects_states.1.1.2`.

split_linear_write_side_effects_states.1.1.3:

```

{-1} every(λ (t: [Address, list[Byte]]):
      cons?(t'2) ∧
      a' ≤ t'1 ∧ t'1 + length(t'2) ≤ a' + length(bl')
      (split(min_page, a', bl'))
    ∧
    every(λ (t: [Address, list[Byte]]):
      type_of(t'1) = type_of(a') ∧
      reg_base(min_linear)(type_of(a')) ≤ t'1 ∧
      t'1 < reg_size(max_linear)(type_of(a'))
      (split(min_page, a', bl'))
    ⊃
    every(λ (t_1: [Address, list[Byte]]):
      (cons?(t_1'2) ∧
      a' ≤ t_1'1 ∧ t_1'1 + length(t_1'2) ≤ a' + length(bl'))
      ∧
      type_of(t_1'1) = type_of(a') ∧
      reg_base(min_linear)(type_of(a')) ≤ t_1'1 ∧
      t_1'1 < reg_size(max_linear)(type_of(a'))
      (split(min_page, a', bl'))
    {-2} every(λ (a2: Address, bl2: list[Byte]): cons?(bl2))(split(min_page, a', bl')) ∧
    every(λ (a2: Address, bl2: list[Byte]):
      a' ≤ a2 ∧ a2 + length(bl2) ≤ a' + length(bl')
      (split(min_page, a', bl'))
    ⊃
    every(λ (t: [Address, list[Byte]]):
      cons?(t'2) ∧
      a' ≤ t'1 ∧ t'1 + length(t'2) ≤ a' + length(bl')
      (split(min_page, a', bl'))
    {-3} every(λ (a2: Address, bl2: list[Byte]):
      rem(expt(2, min_page))(offset(a2)) + length(bl2) ≤ expt(2, min_page))
      (split(min_page, a', bl'))
    {-4} every(λ (t: [Address, list[Byte]]):
      type_of(t'1) = type_of(a') ∧
      reg_base(min_linear)(type_of(a')) ≤ t'1 ∧
      t'1 < reg_size(max_linear)(type_of(a'))
      (split(min_page, a', bl'))
    {-5} every(λ (a2: Address, bl2: list[Byte]): cons?(bl2))(split(min_page, a', bl'))
    {-6} every(λ (a2: Address, bl2: list[Byte]):
      a' ≤ a2 ∧ a2 + length(bl2) ≤ a' + length(bl')
      (split(min_page, a', bl'))
    {-7} Mem?(a' type_of)
    {-8} 0 ≤ a' offset
    {-9} a' offset < max_linear_offset
    {-10} every(λ (x: number):
      number_field_pred(x) ∧
      real_pred(x) ∧
      rational_pred(x) ∧ integer_pred(x) ∧ x ≥ 0 ∧ x < max_byte)
      (bl')
    {-11} cons?[Byte](bl')
    {-12} is_linear_plain_memory?(pm')
    {-13} (address_block(a', length(bl')) ⊆ pm'rw_addr)
  }
}
{1} ∀ (t_1: [Address, list[Byte]]):
    type_of(t_1'1) = type_of(a') ∧
    t_1'1 + length(t_1'2) ≤ a' + length(bl') ∧
    a' ≤ t_1'1 ∧ cons?(t_1'2)
    ⊃ Mem?(min_linear type_of)
{2} every(λ (e: [Memory_Address_4G, list[Byte]]):
    transformer_invariant?(pm' states,
    singleton(expr_2_super(linear_write_side_effect_in_page(e))))
    (split(min_page, a', bl'))

```


C Proof scripts

Expanding the definition of `min_linear`,

Expanding the definition of `Mem`,

which is trivially true.

This completes the proof of `split_linear_write_side_effects_states.1.1.3`.

`split_linear_write_side_effects_states.1.2`:

<div style="display: flex; flex-direction: column; gap: 5px;"> <div style="display: flex; align-items: flex-start;"> {-1} <div style="flex-grow: 1;"> $\text{every}(\lambda (a_2: \text{Address}, \text{bl2}: \text{list}[\text{Byte}]): \text{cons}?(a_2))(\text{split}(\text{min_page}, a', \text{bl}')) \wedge$ $\text{every}(\lambda (a_2: \text{Address}, \text{bl2}: \text{list}[\text{Byte}]):$ $a' \leq a_2 \wedge a_2 + \text{length}(\text{bl2}) \leq a' + \text{length}(\text{bl}'))$ $(\text{split}(\text{min_page}, a', \text{bl}'))$ \supset $\text{every}(\lambda (t: [\text{Address}, \text{list}[\text{Byte}]]):$ $\text{cons}?(t'2) \wedge$ $a' \leq t'1 \wedge t'1 + \text{length}(t'2) \leq a' + \text{length}(\text{bl}'))$ $(\text{split}(\text{min_page}, a', \text{bl}'))$ </div> </div> <div style="display: flex; align-items: flex-start;"> {-2} <div style="flex-grow: 1;"> $\text{every}(\lambda (a_2: \text{Address}, \text{bl2}: \text{list}[\text{Byte}]):$ $\text{rem}(\text{expt}(2, \text{min_page}))(\text{offset}(a_2)) + \text{length}(\text{bl2}) \leq \text{expt}(2, \text{min_page}))$ $(\text{split}(\text{min_page}, a', \text{bl}'))$ </div> </div> <div style="display: flex; align-items: flex-start;"> {-3} <div style="flex-grow: 1;"> $\text{every}(\lambda (t: [\text{Address}, \text{list}[\text{Byte}]]):$ $\text{type_of}(t'1) = \text{type_of}(a') \wedge$ $\text{reg_base}(\text{min_linear})(\text{type_of}(a')) \leq t'1 \wedge$ $t'1 < \text{reg_size}(\text{max_linear})(\text{type_of}(a'))$ $(\text{split}(\text{min_page}, a', \text{bl}'))$ </div> </div> <div style="display: flex; align-items: flex-start;"> {-4} <div style="flex-grow: 1;"> $\text{every}(\lambda (a_2: \text{Address}, \text{bl2}: \text{list}[\text{Byte}]): \text{cons}?(a_2))(\text{split}(\text{min_page}, a', \text{bl}'))$ </div> </div> <div style="display: flex; align-items: flex-start;"> {-5} <div style="flex-grow: 1;"> $\text{every}(\lambda (a_2: \text{Address}, \text{bl2}: \text{list}[\text{Byte}]):$ $a' \leq a_2 \wedge a_2 + \text{length}(\text{bl2}) \leq a' + \text{length}(\text{bl}'))$ $(\text{split}(\text{min_page}, a', \text{bl}'))$ </div> </div> <div style="display: flex; align-items: flex-start;"> {-6} <div style="flex-grow: 1;"> $\text{Mem}?(a' \text{ 'type_of})$ </div> </div> <div style="display: flex; align-items: flex-start;"> {-7} <div style="flex-grow: 1;"> $0 \leq a' \text{ 'offset}$ </div> </div> <div style="display: flex; align-items: flex-start;"> {-8} <div style="flex-grow: 1;"> $a' \text{ 'offset} < \text{max_linear_offset}$ </div> </div> <div style="display: flex; align-items: flex-start;"> {-9} <div style="flex-grow: 1;"> $\text{every}(\lambda (x: \text{number}):$ $\text{number_field_pred}(x) \wedge$ $\text{real_pred}(x) \wedge$ $\text{rational_pred}(x) \wedge \text{integer_pred}(x) \wedge x \geq 0 \wedge x < \text{max_byte})$ (a') </div> </div> <div style="display: flex; align-items: flex-start;"> {-10} <div style="flex-grow: 1;"> $\text{cons}?[\text{Byte}](a')$ </div> </div> <div style="display: flex; align-items: flex-start;"> {-11} <div style="flex-grow: 1;"> $\text{is_linear_plain_memory}?(a')$ </div> </div> <div style="display: flex; align-items: flex-start;"> {-12} <div style="flex-grow: 1;"> $(\text{address_block}(a', \text{length}(a')) \subseteq a' \text{ 'rw_addr})$ </div> </div> <hr style="border: 0.5px solid black; margin: 5px 0;"/> <div style="display: flex; align-items: flex-start;"> {1} <div style="flex-grow: 1;"> $\forall (t: [\text{Address}, \text{list}[\text{Byte}]]):$ $\text{reg_base}(\text{min_linear})(\text{type_of}(a')) \leq t'1 \wedge \text{type_of}(t'1) = \text{type_of}(a') \supset$ $\text{Mem}?(a' \text{ 'type_of})$ </div> </div> <div style="display: flex; align-items: flex-start;"> {2} <div style="flex-grow: 1;"> $\text{every}(\lambda (e: [\text{Memory_Address_4G}, \text{list}[\text{Byte}]]):$ $\text{transformer_invariant}?(a' \text{ 'states},$ $\text{singleton}(\text{expr_2_super}(\text{linear_write_side_effect_in_page}(e)))$ $(\text{split}(\text{min_page}, a', \text{bl}'))$ </div> </div> </div>

Expanding the definition of `max_linear`,

Expanding the definition of `Mem`,

which is trivially true.

This completes the proof of `split_linear_write_side_effects_states.1.2`.

split_linear_write_side_effects_states.1.3:

{-1}	every(λ (a_2 : Address, $bl2$: list[Byte]): cons?($bl2$))(split(min_page, a' , bl') \wedge every(λ (a_2 : Address, $bl2$: list[Byte]): $a' \leq a_2 \wedge a_2 + \text{length}(bl2) \leq a' + \text{length}(bl')$) (split(min_page, a' , bl')) \supset every(λ (t : [Address, list[Byte]]): cons?($t'2$) \wedge $a' \leq t'1 \wedge t'1 + \text{length}(t'2) \leq a' + \text{length}(bl')$) (split(min_page, a' , bl'))
{-2}	every(λ (a_2 : Address, $bl2$: list[Byte]): rem(expt(2, min_page))(offset(a_2)) + length($bl2$) \leq expt(2, min_page)) (split(min_page, a' , bl'))
{-3}	every(λ (t : [Address, list[Byte]]): type_of($t'1$) = type_of(a') \wedge reg_base(min_linear)(type_of(a')) \leq $t'1 \wedge t'1 <$ reg_size(max_linear)(type_of(a'))) (split(min_page, a' , bl'))
{-4}	every(λ (a_2 : Address, $bl2$: list[Byte]): cons?($bl2$))(split(min_page, a' , bl'))
{-5}	every(λ (a_2 : Address, $bl2$: list[Byte]): $a' \leq a_2 \wedge a_2 + \text{length}(bl2) \leq a' + \text{length}(bl')$) (split(min_page, a' , bl'))
{-6}	Mem?(a' 'type_of)
{-7}	$0 \leq a'$ 'offset
{-8}	a' 'offset < max_linear_offset
{-9}	every(λ (x : number): number_field_pred(x) \wedge real_pred(x) \wedge rational_pred(x) \wedge integer_pred(x) \wedge $x \geq 0 \wedge x <$ max_byte) (bl')
{-10}	cons?[Byte](bl')
{-11}	is_linear_plain_memory?(pm')
{-12}	(address_block(a' , length(bl')) \subseteq pm' 'rw_addr)
{1}	\forall (t : [Address, list[Byte]]): type_of($t'1$) = type_of(a') \supset Mem?(min_linear'type_of)
{2}	every(λ (e : [Memory_Address_4G, list[Byte]]): transformer_invariant?(pm' 'states, singleton(expr_2_super(linear_write_side_effect_in_page(e)))) (split(min_page, a' , bl'))

Expanding the definition of min_linear,

Expanding the definition of Mem,

which is trivially true.

This completes the proof of split_linear_write_side_effects_states.1.3.

C Proof scripts

`split_linear_write_side_effects_states.2:`

{-1}	$\text{every}(\lambda (a_2: \text{Address}, \text{bl2}: \text{list}[\text{Byte}]):$ $\quad \text{rem}(\text{expt}(2, \text{min_page}))(\text{offset}(a_2)) + \text{length}(\text{bl2}) \leq \text{expt}(2, \text{min_page}))$ $\quad (\text{split}(\text{min_page}, a', \text{bl}'))$
{-2}	$\text{every}(\lambda (a_2: \text{Address}, \text{bl2}: \text{list}[\text{Byte}]): \text{cons}?(\text{bl2}))(\text{split}(\text{min_page}, a', \text{bl}'))$
{-3}	$\text{every}(\lambda (a_2: \text{Address}, \text{bl2}: \text{list}[\text{Byte}]):$ $\quad a' \leq a_2 \wedge a_2 + \text{length}(\text{bl2}) \leq a' + \text{length}(\text{bl}'))$ $\quad (\text{split}(\text{min_page}, a', \text{bl}'))$
{-4}	$\text{Mem}?(a', \text{type_of})$
{-5}	$0 \leq a' \text{'offset}$
{-6}	$a' \text{'offset} < \text{max_linear_offset}$
{-7}	$\text{every}(\lambda (x: \text{number}):$ $\quad \text{number_field_pred}(x) \wedge$ $\quad \text{real_pred}(x) \wedge$ $\quad \text{rational_pred}(x) \wedge \text{integer_pred}(x) \wedge x \geq 0 \wedge x < \text{max_byte})$ $\quad (\text{bl}')$
{-8}	$\text{cons}?[\text{Byte}](\text{bl}')$
{-9}	$\text{is_linear_plain_memory}?(\text{pm}')$
{-10}	$(\text{address_block}(a', \text{length}(\text{bl}')) \subseteq \text{pm}' \text{'rw_addr})$
{1}	$a' + \text{length}(\text{bl}') \leq \text{reg_size}(\text{max_linear})(\text{type_of}(a'))$
{2}	$\text{every}(\lambda (e: [\text{Memory_Address_4G}, \text{list}[\text{Byte}]]):$ $\quad \text{transformer_invariant}?(\text{pm}' \text{'states},$ $\quad \quad \text{singleton}(\text{expr_2_super}(\text{linear_write_side_effect_in_page}(e)))$ $\quad (\text{split}(\text{min_page}, a', \text{bl}'))$

Hiding formulas: (-1 -2 -3 -7 2),

Expanding the definition of length,

Using lemma `pm_memory_addr`,

Simplifying, rewriting, and recording with decision procedures,

Hiding formulas: -6,

Expanding the definition of subset?,

Installing automatic rewrites from: `Mem max_linear bus_width min_linear`

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `split_linear_write_side_effects_states.2`.

split_linear_write_side_effects_states.3:

{-1}	every(λ (a_2 : Address, $bl2$: list[Byte]): rem(expt(2, min_page))(offset(a_2)) + length($bl2$) \leq expt(2, min_page)) (split(min_page, a' , bl'))
{-2}	every(λ (a_2 : Address, $bl2$: list[Byte]): cons?($bl2$))(split(min_page, a' , bl'))
{-3}	every(λ (a_2 : Address, $bl2$: list[Byte]): $a' \leq a_2 \wedge a_2 + \text{length}(bl2) \leq a' + \text{length}(bl')$ (split(min_page, a' , bl'))
{-4}	Mem?(a' 'type_of)
{-5}	$0 \leq a'$ 'offset
{-6}	a' 'offset < max_linear_offset
{-7}	every(λ (x : number): number_field_pred(x) \wedge real_pred(x) \wedge rational_pred(x) \wedge integer_pred(x) $\wedge x \geq 0 \wedge x < \text{max_byte}$) (bl')
{-8}	cons?[Byte](bl')
{-9}	is_linear_plain_memory?(pm')
{-10}	(address_block(a' , length(bl'))) $\subseteq pm'$ 'rw_addr
{1}	reg_base(min_linear)(type_of(a')) $\leq a'$
{2}	every(λ (e : [Memory_Address_4G, list[Byte]]): transformer_invariant?(pm' 'states, singleton(expr_2_super(linear_write_side_effect_in_page(e)))) (split(min_page, a' , bl'))

Expanding the definition of reg_base,

Expanding the definition of min_linear,

Expanding the definition of Mem,

Expanding the definition of <=,

which is trivially true.

This completes the proof of split_linear_write_side_effects_states.3.

Q.E.D.

C.117.78 Linear_Memory_Properties.split_linear_read_side_effects_ok

Terse proof for split_linear_read_side_effects_ok.

split_linear_read_side_effects_ok:

{1}	\forall (pm : Plain_Memory[Linear_memory], s : Linear_memory[Physical_memory, pm_phy], a : Memory_Address_4G, bl : (cons?[Byte])): is_linear_plain_memory?(pm) \wedge (address_block(a , length(bl))) \subseteq (pm 'ro_addr $\cup pm$ 'rw_addr) $\wedge pm$ 'states(s) \supset every(λ (e : [Memory_Address_4G, list[Byte]]): OK?(linear_read_side_effect_in_page(e)(s))) (split(min_page, a , bl))
-----	---

Repeatedly Skolemizing and flattening,

Installing automatic rewrites from: Mem!! max_linear!! min_linear!!

Using lemma split_range,

Using lemma split_no_null,

Using lemma split_type,

C Proof scripts

Using lemma `split_pair_cross_size`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 3 subgoals:

`split_linear_read_side_effects_ok.1`:

{-1}	every(λ (a_2 : Address, bl_2 : list[Byte]): rem(expt(2, min_page))(offset(a_2)) + length(bl_2) \leq expt(2, min_page)) (split(min_page, a' , bl'))
{-2}	every(λ (t : [Address, list[Byte]]): type_of($t'1$) = type_of(a') \wedge reg_base((#type_of := Mem_, offset := 0#))(type_of(a')) \leq $t'1$ \wedge $t'1$ < reg_size((#type_of := Mem_, offset := max_linear_offset#))(type_of(a'))) (split(min_page, a' , bl'))
{-3}	every(λ (a_2 : Address, bl_2 : list[Byte]): cons?(bl_2))(split(min_page, a' , bl'))
{-4}	every(λ (a_2 : Address, bl_2 : list[Byte]): $a' \leq a_2$ \wedge a_2 + length(bl_2) \leq a' + length(bl')) (split(min_page, a' , bl'))
{-5}	Mem?(a' 'type_of)
{-6}	$0 \leq a'$ 'offset
{-7}	a' 'offset < max_linear_offset
{-8}	every(λ (x : number): number_field_pred(x) \wedge real_pred(x) \wedge rational_pred(x) \wedge integer_pred(x) \wedge $x \geq 0$ \wedge $x <$ max_byte) (bl')
{-9}	cons?[Byte](bl')
{-10}	is_linear_plain_memory?(pm')
{-11}	(address_block(a' , length(bl'))) \subseteq (pm' 'ro_addr \cup pm' 'rw_addr)
{-12}	pm' 'states(s')
{1}	every(λ (e : [Memory_Address_4G, list[Byte]]): OK?(linear_read_side_effect_in_page(e)(s'))) (split(min_page, a' , bl'))

Using lemma `every_conjunct_left`,

Using lemma `every_conjunct_left`,

we get 3 subgoals:

split_linear_read_side_effects_ok.1.1:

```

{-1} every(λ (t: [Address, list[Byte]]):
      cons?(t'2) ∧
      a' ≤ t'1 ∧ t'1 + length(t'2) ≤ a' + length(bl')
      (split(min_page, a', bl'))
    ∧
    every(λ (t: [Address, list[Byte]]):
      type_of(t'1) = type_of(a') ∧
      reg_base(min_linear)(type_of(a')) ≤ t'1 ∧
      t'1 < reg_size(max_linear)(type_of(a'))
      (split(min_page, a', bl'))
    ⊃
    every(λ (t_1: [Address, list[Byte]]):
      (cons?(t_1'2) ∧
      a' ≤ t_1'1 ∧ t_1'1 + length(t_1'2) ≤ a' + length(bl'))
      ∧
      type_of(t_1'1) = type_of(a') ∧
      reg_base(min_linear)(type_of(a')) ≤ t_1'1 ∧
      t_1'1 < reg_size(max_linear)(type_of(a'))
      (split(min_page, a', bl'))
    {-2} every(λ (a2: Address, bl2: list[Byte]): cons?(bl2))(split(min_page, a', bl')) ∧
    every(λ (a2: Address, bl2: list[Byte]):
      a' ≤ a2 ∧ a2 + length(bl2) ≤ a' + length(bl')
      (split(min_page, a', bl'))
    ⊃
    every(λ (t: [Address, list[Byte]]):
      cons?(t'2) ∧
      a' ≤ t'1 ∧ t'1 + length(t'2) ≤ a' + length(bl')
      (split(min_page, a', bl'))
    {-3} every(λ (a2: Address, bl2: list[Byte]):
      rem(expt(2, min_page))(offset(a2)) + length(bl2) ≤ expt(2, min_page))
      (split(min_page, a', bl'))
    {-4} every(λ (t: [Address, list[Byte]]):
      type_of(t'1) = type_of(a') ∧
      reg_base((#type_of := Mem_, offset := 0#))(type_of(a')) ≤ t'1 ∧
      t'1 < reg_size((#type_of := Mem_, offset := max_linear_offset#))(type_of(a'))
      (split(min_page, a', bl'))
    {-5} every(λ (a2: Address, bl2: list[Byte]): cons?(bl2))(split(min_page, a', bl'))
    {-6} every(λ (a2: Address, bl2: list[Byte]):
      a' ≤ a2 ∧ a2 + length(bl2) ≤ a' + length(bl')
      (split(min_page, a', bl'))
    {-7} Mem?(a' type_of)
    {-8} 0 ≤ a' offset
    {-9} a' offset < max_linear_offset
    {-10} every(λ (x: number):
      number_field_pred(x) ∧
      real_pred(x) ∧
      rational_pred(x) ∧ integer_pred(x) ∧ x ≥ 0 ∧ x < max_byte)
      (bl')
    {-11} cons?[Byte](bl')
    {-12} is_linear_plain_memory?(pm')
    {-13} (address_block(a', length(bl')) ⊆ (pm'ro_addr ∪ pm'rw_addr))
    {-14} pm' states(s')
  }
  {1} every(λ (e: [Memory_Address_4G, list[Byte]]):
    OK?(linear_read_side_effect_in_page(e)(s'))
    (split(min_page, a', bl'))
  }

```

C Proof scripts

Using lemma `every_conjunct_left`,

we get 3 subgoals:

split_linear_read_side_effects_ok.1.1.1:

```

{-1} every(λ (t_1: [Address, list[Byte]]):
    cons?(t_1'2) ∧
    a' ≤ t_1'1 ∧
    t_1'1 + length(t_1'2) ≤ a' + length(bl') ∧
    type_of(t_1'1) = type_of(a') ∧
    reg_base(min_linear)(type_of(a')) ≤ t_1'1 ∧
    t_1'1 < reg_size(max_linear)(type_of(a')))
    (split(min_page, a', bl'))
    ∧
    every(λ (a_2: Address, bl2: list[Byte]):
        rem(expt(2, min_page))(offset(a_2)) + length(bl2) ≤
        expt(2, min_page))
        (split(min_page, a', bl'))
    ⊃
    every(λ (t: [Address, list[Byte]]):
        (cons?(t'2) ∧
        a' ≤ t'1 ∧
        t'1 + length(t'2) ≤ a' + length(bl') ∧
        type_of(t'1) = type_of(a') ∧
        reg_base(min_linear)(type_of(a')) ≤ t'1 ∧
        t'1 < reg_size(max_linear)(type_of(a')))
        ∧
        rem(expt(2, min_page))(offset(t'1)) + length(t'2) ≤
        expt(2, min_page))
        (split(min_page, a', bl'))
{-2} every(λ (t: [Address, list[Byte]]):
    cons?(t'2) ∧
    a' ≤ t'1 ∧ t'1 + length(t'2) ≤ a' + length(bl'))
    (split(min_page, a', bl'))
    ∧
    every(λ (t: [Address, list[Byte]]):
        type_of(t'1) = type_of(a') ∧
        reg_base(min_linear)(type_of(a')) ≤ t'1 ∧
        t'1 < reg_size(max_linear)(type_of(a')))
        (split(min_page, a', bl'))
    ⊃
    every(λ (t_1: [Address, list[Byte]]):
        (cons?(t_1'2) ∧
        a' ≤ t_1'1 ∧ t_1'1 + length(t_1'2) ≤ a' + length(bl'))
        ∧
        type_of(t_1'1) = type_of(a') ∧
        reg_base(min_linear)(type_of(a')) ≤ t_1'1 ∧
        t_1'1 < reg_size(max_linear)(type_of(a')))
        (split(min_page, a', bl'))
{-3} every(λ (a_2: Address, bl2: list[Byte]): cons?(bl2))(split(min_page, a', bl')) ∧
    every(λ (a_2: Address, bl2: list[Byte]):
        a' ≤ a_2 ∧ a_2 + length(bl2) ≤ a' + length(bl'))
        (split(min_page, a', bl'))
    ⊃
    every(λ (t: [Address, list[Byte]]):
        cons?(t'2) ∧
        a' ≤ t'1 ∧ t'1 + length(t'2) ≤ a' + length(bl'))
        (split(min_page, a', bl'))
{-4} every(λ (a_2: Address, bl2: list[Byte]):
    rem(expt(2, min_page))(offset(a_2)) + length(bl2) ≤ expt(2, min_page))
    (split(min_page, a', bl'))
{-5} every(λ (t: [Address, list[Byte]]):
    type_of(t'1) = type_of(a') ∧
    reg_base((#type_of := Mem_, offset := 0#))(type_of(a')) ≤ t'1 ∧
    t'1 < reg_size((#type_of := Mem_, offset := max_linear_offset#))(type_of(a')))
    (split(min_page, a', bl'))
{-6} every(λ (a_2: Address, bl2: list[Byte]): cons?(bl2))(split(min_page, a', bl'))
{-7} every(λ (a_2: Address, bl2: list[Byte]):

```


Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Hiding formulas: (-1 -3 -4 -5 -6 -7 -11),

Installing automatic rewrites from: (### expr_2_super! expr_2_super_res!)

Using lemma every_extend[[Address, list[Byte]], [Memory_Address_4G, list[Byte]]],

we get 3 subgoals:

split_linear_read_side_effects_ok.1.1.1.1:

<pre> {-1} every(λ (t_1: [Address, list[Byte]]): cons?(t_1'2) ∧ a' ≤ t_1'1 ∧ t_1'1 + length(t_1'2) ≤ a' + length(bl') ∧ type_of(t_1'1) = type_of(a') ∧ reg_base(min_linear)(type_of(a')) ≤ t_1'1 ∧ t_1'1 < reg_size(max_linear)(type_of(a')) ∧ rem(expt(2, min_page))(offset(t_1'1)) + length(t_1'2) ≤ expt(2, min_page)) (split(min_page, a', bl')) ⊃ every(λ (s: [Memory_Address_4G, list[Byte]]): cons?(s'2) ∧ a' ≤ s'1 ∧ s'1 + length(s'2) ≤ a' + length(bl') ∧ type_of(s'1) = type_of(a') ∧ reg_base(min_linear)(type_of(a')) ≤ s'1 ∧ s'1 < reg_size(max_linear)(type_of(a')) ∧ rem(expt(2, min_page))(offset(s'1)) + length(s'2) ≤ expt(2, min_page)) (split(min_page, a', bl')) {-2} every(λ (t: [Address, list[Byte]]): (cons?(t'2) ∧ a' ≤ t'1 ∧ t'1 + length(t'2) ≤ a' + length(bl') ∧ type_of(t'1) = type_of(a') ∧ reg_base((#type_of := Mem_, offset := 0#))(type_of(a')) ≤ t'1 ∧ t'1 < reg_size((#type_of := Mem_, offset := max_linear_offset#))(type_of(a'))) ∧ rem(expt(2, min_page))(offset(t'1)) + length(t'2) ≤ expt(2, min_page)) (split(min_page, a', bl')) {-3} Mem?(a' type_of) {-4} 0 ≤ a' offset {-5} a' offset < max_linear_offset {-6} cons?[Byte](bl') {-7} is_linear_plain_memory?(pm') {-8} (address_block(a', length(bl')) ⊆ (pm'ro_addr ∪ pm'rw_addr)) {-9} pm' states(s') </pre>	<pre> {1} every(λ (e: [Memory_Address_4G, list[Byte]]): OK?(linear_read_side_effect_in_page(e)(s'))) (split(min_page, a', bl')) </pre>
--	--

Using lemma every_implied,

we get 3 subgoals:

split_linear_read_side_effects_ok.1.1.1.1.1:

```

{-1}  (∀ (t: [Memory_Address_4G, list[Byte]]):
      cons?(t'2) ∧
      a' ≤ t'1 ∧
      t'1 + length(t'2) ≤ a' + length(bl') ∧
      type_of(t'1) = type_of(a') ∧
      reg_base(min_linear)(type_of(a')) ≤ t'1 ∧
      t'1 < reg_size(max_linear)(type_of(a')) ∧
      rem(expt(2, min_page))(offset(t'1)) + length(t'2) ≤
      expt(2, min_page)
      ⊃ OK?(linear_read_side_effect_in_page(t)(s')))
      ∧
      every(λ (s: [Memory_Address_4G, list[Byte]]):
        cons?(s'2) ∧
        a' ≤ s'1 ∧
        s'1 + length(s'2) ≤ a' + length(bl') ∧
        type_of(s'1) = type_of(a') ∧
        reg_base(min_linear)(type_of(a')) ≤ s'1 ∧
        s'1 < reg_size(max_linear)(type_of(a')) ∧
        rem(expt(2, min_page))(offset(s'1)) + length(s'2) ≤
        expt(2, min_page)
        (split(min_page, a', bl')))
        ⊃
        every(λ (e: [Memory_Address_4G, list[Byte]]):
          OK?(linear_read_side_effect_in_page(e)(s')))
          (split(min_page, a', bl')))
{-2}  every(λ (t_1: [Address, list[Byte]]):
      cons?(t_1'2) ∧
      a' ≤ t_1'1 ∧
      t_1'1 + length(t_1'2) ≤ a' + length(bl') ∧
      type_of(t_1'1) = type_of(a') ∧
      reg_base(min_linear)(type_of(a')) ≤ t_1'1 ∧
      t_1'1 < reg_size(max_linear)(type_of(a')) ∧
      rem(expt(2, min_page))(offset(t_1'1)) + length(t_1'2) ≤
      expt(2, min_page)
      (split(min_page, a', bl')))
      ⊃
      every(λ (s: [Memory_Address_4G, list[Byte]]):
        cons?(s'2) ∧
        a' ≤ s'1 ∧
        s'1 + length(s'2) ≤ a' + length(bl') ∧
        type_of(s'1) = type_of(a') ∧
        reg_base(min_linear)(type_of(a')) ≤ s'1 ∧
        s'1 < reg_size(max_linear)(type_of(a')) ∧
        rem(expt(2, min_page))(offset(s'1)) + length(s'2) ≤
        expt(2, min_page)
        (split(min_page, a', bl')))
        every(λ (t: [Address, list[Byte]]):
          (cons?(t'2) ∧
           a' ≤ t'1 ∧
           t'1 + length(t'2) ≤ a' + length(bl') ∧
           type_of(t'1) = type_of(a') ∧
           reg_base((#type_of := Mem_, offset := 0#))(type_of(a')) ≤ t'1 ∧
           t'1 <
           reg_size((#type_of := Mem_, offset := max_linear_offset#))(type_of(a')))
           ∧
           rem(expt(2, min_page))(offset(t'1)) + length(t'2) ≤
           expt(2, min_page)
           (split(min_page, a', bl')))
{-4}  Mem?(a' type_of)
{-5}  0 ≤ a' offset
{-6}  a' offset < max_linear_offset
{-7}  cons?[Byte](bl')
{-8}  is_linear_plain_memory?(pm')

```

C Proof scripts

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Hiding formulas: (-1 -2 2),

Expanding the definition of `linear_read_side_effect_in_page`,

Repeatedly Skolemizing and flattening,

Case splitting on `union(pm!1'ro_addr, pm!1'rw_addr)(t!1'1)`,

we get 2 subgoals:

`split_linear_read_side_effects_ok.1.1.1.1.1.1:`

{-1}	<code>union(pm'ro_addr, pm'rw_addr)(t'1)</code>
{-2}	<code>cons?(t'2)</code>
{-3}	<code>a' ≤ t'1</code>
{-4}	<code>t'1 + length(t'2) ≤ a' + length(bl')</code>
{-5}	<code>type_of(t'1) = type_of(a')</code>
{-6}	<code>reg_base((#type_of := Mem_, offset := 0#))(type_of(a')) ≤ t'1</code>
{-7}	<code>t'1 < reg_size((#type_of := Mem_, offset := max_linear_offset#))(type_of(a'))</code>
{-8}	<code>rem(expt(2, min_page))(offset(t'1)) + length(t'2) ≤ expt(2, min_page)</code>
{-9}	<code>Mem?(a'type_of)</code>
{-10}	<code>0 ≤ a'offset</code>
{-11}	<code>a'offset < max_linear_offset</code>
{-12}	<code>cons?[Byte](bl')</code>
{-13}	<code>is_linear_plain_memory?(pm')</code>
{-14}	<code>(address_block(a', length(bl'))) ⊆ (pm'ro_addr ∪ pm'rw_addr)</code>
{-15}	<code>pm'states(s')</code>
{1}	<code>OK?((linear_resolve(t'1, Read) ## (λ (pa: Memory_Address_4G)(ns: Linear_memory[Physical_memory, pm_phy]): memory_read_side_effect(pm_phy'mem)(pa, t'2, TRUE)(s')) (s'))</code>

Using lemma `pm_linear_resolve_read_ok`,

Case splitting on `subset?(address_block(data(linear_resolve(t!1'1, Read)(s!1)), length(t!1'2)), union(pm_phy'ro_addr, pm_phy'rw_addr))`,

we get 3 subgoals:

split_linear_read_side_effects_ok.1.1.1.1.1.1.1:

{-1}	$(\text{address_block}(\text{data}(\text{linear_resolve}(t'1, \text{Read})(s')), \text{length}(t'2)) \subseteq (\text{pm_phy}'\text{ro_addr} \cup \text{pm_phy}'\text{rw_addr}))$
{-2}	$\text{is_linear_plain_memory?}(\text{pm}') \supset \text{OK?}(\text{linear_resolve}(t'1, \text{Read})(s'))$
{-3}	$\text{union}(\text{pm}'\text{ro_addr}, \text{pm}'\text{rw_addr})(t'1)$
{-4}	$\text{cons?}(t'2)$
{-5}	$a' \leq t'1$
{-6}	$t'1 + \text{length}(t'2) \leq a' + \text{length}(bl')$
{-7}	$\text{type_of}(t'1) = \text{type_of}(a')$
{-8}	$\text{reg_base}((\# \text{type_of} := \text{Mem_}, \text{offset} := 0\#))(\text{type_of}(a')) \leq t'1$
{-9}	$t'1 < \text{reg_size}((\# \text{type_of} := \text{Mem_}, \text{offset} := \text{max_linear_offset}\#))(\text{type_of}(a'))$
{-10}	$\text{rem}(\text{expt}(2, \text{min_page}))(\text{offset}(t'1)) + \text{length}(t'2) \leq \text{expt}(2, \text{min_page})$
{-11}	$\text{Mem?}(a' \text{ type_of})$
{-12}	$0 \leq a' \text{ offset}$
{-13}	$a' \text{ offset} < \text{max_linear_offset}$
{-14}	$\text{cons?}[\text{Byte}](bl')$
{-15}	$\text{is_linear_plain_memory?}(\text{pm}')$
{-16}	$(\text{address_block}(a', \text{length}(bl'))) \subseteq (\text{pm}'\text{ro_addr} \cup \text{pm}'\text{rw_addr})$
{-17}	$\text{pm}'\text{states}(s')$
{1}	$\text{OK?}((\text{linear_resolve}(t'1, \text{Read}) \#\#$ $(\lambda (\text{pa}: \text{Memory_Address_4G})(\text{ns}: \text{Linear_memory}[\text{Physical_memory}, \text{pm_phy}]):$ $\text{memory_read_side_effect}(\text{pm_phy}'\text{mem})(\text{pa}, t'2, \text{TRUE})(s'))$ $(s'))$

Using lemma linear_resolve_states,

Using lemma pm_states,

Using lemma pm_plain_phy,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Using lemma plain_memory_transformers_ok_read_side_effects_block,

Using lemma super_transformers_ok_ok,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Expanding the definition of memory_read_side_effect_super_transformers,

Instantiating quantified variables,

Rewriting using subset_reflexive, matching in *,

This completes the proof of split_linear_read_side_effects_ok.1.1.1.1.1.1.1.

split_linear_read_side_effects_ok.1.1.1.1.1.1.2:

{-1}	is_linear_plain_memory?(pm') \supset OK?(linear_resolve(t'1, Read)(s'))
{-2}	union(pm'ro_addr, pm'rw_addr)(t'1)
{-3}	cons?(t'2)
{-4}	$a' \leq t'1$
{-5}	$t'1 + \text{length}(t'2) \leq a' + \text{length}(b')$
{-6}	type_of(t'1) = type_of(a')
{-7}	reg_base((#type_of := Mem_, offset := 0#))(type_of(a')) $\leq t'1$
{-8}	$t'1 < \text{reg_size}((\#type_of := \text{Mem_}, \text{offset} := \text{max_linear_offset\#}))(type_of(a'))$
{-9}	$\text{rem}(\text{expt}(2, \text{min_page}))(\text{offset}(t'1)) + \text{length}(t'2) \leq \text{expt}(2, \text{min_page})$
{-10}	Mem?(a'type_of)
{-11}	$0 \leq a'\text{offset}$
{-12}	$a'\text{offset} < \text{max_linear_offset}$
{-13}	cons?[Byte](b')
{-14}	is_linear_plain_memory?(pm')
{-15}	$(\text{address_block}(a', \text{length}(b'))) \subseteq (\text{pm'ro_addr} \cup \text{pm'rw_addr})$
{-16}	pm'states(s')
{1}	$(\text{address_block}(\text{data}(\text{linear_resolve}(t'1, \text{Read})(s')), \text{length}(t'2))) \subseteq (\text{pm_phy'ro_addr} \cup \text{pm_phy'rw_addr})$
{2}	OK?((linear_resolve(t'1, Read) ## $(\lambda (\text{pa}: \text{Memory_Address_4G})(\text{ns}: \text{Linear_memory}[\text{Physical_memory}, \text{pm_phy}]):$ $\text{memory_read_side_effect}(\text{pm_phy'mem})(\text{pa}, t'2, \text{TRUE})(s'))$ $(s'))$

Hiding formulas: 2,

Using lemma same_page_address_block_read,

we get 3 subgoals:

split_linear_read_side_effects_ok.1.1.1.1.1.1.2.1:

{-1}	is_linear_plain_memory?(pm') \wedge pm' states(s') \wedge union(pm' ro_addr, pm' rw_addr)(t'1) \wedge $a' \leq t'1 \wedge$ $t'1 + \text{length}(t'2) \leq a' + \text{length}(bl') \wedge$ $\text{rem}(\text{expt}(2, \text{min_page}))(\text{offset}(t'1)) + \text{length}(t'2) \leq \text{expt}(2, \text{min_page})$ $\wedge (\text{address_block}(a', \text{length}(bl')) \subseteq (\text{pm}'\text{ro_addr} \cup \text{pm}'\text{rw_addr}))$ \supset $(\text{address_block}(\text{data}(\text{linear_resolve}(t'1, \text{Read})(s')), \text{length}(t'2)) \subseteq (\text{pm_phy}'\text{ro_addr} \cup \text{pm_phy}'\text{rw_addr}))$
{-2}	is_linear_plain_memory?(pm') \supset OK?(linear_resolve(t'1, Read)(s'))
{-3}	union(pm' ro_addr, pm' rw_addr)(t'1)
{-4}	cons?(t'2)
{-5}	$a' \leq t'1$
{-6}	$t'1 + \text{length}(t'2) \leq a' + \text{length}(bl')$
{-7}	type_of(t'1) = type_of(a')
{-8}	reg_base((#type_of := Mem_, offset := 0#))(type_of(a')) $\leq t'1$
{-9}	$t'1 < \text{reg_size}((\#type_of := \text{Mem_}, \text{offset} := \text{max_linear_offset\#}))(type_of(a'))$
{-10}	$\text{rem}(\text{expt}(2, \text{min_page}))(\text{offset}(t'1)) + \text{length}(t'2) \leq \text{expt}(2, \text{min_page})$
{-11}	Mem?(a' type_of)
{-12}	$0 \leq a'\text{'offset}$
{-13}	$a'\text{'offset} < \text{max_linear_offset}$
{-14}	cons?[Byte](bl')
{-15}	is_linear_plain_memory?(pm')
{-16}	$(\text{address_block}(a', \text{length}(bl')) \subseteq (\text{pm}'\text{ro_addr} \cup \text{pm}'\text{rw_addr}))$
{-17}	pm' states(s')
{1}	$(\text{address_block}(\text{data}(\text{linear_resolve}(t'1, \text{Read})(s')), \text{length}(t'2)) \subseteq (\text{pm_phy}'\text{ro_addr} \cup \text{pm_phy}'\text{rw_addr}))$

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of split_linear_read_side_effects_ok.1.1.1.1.1.1.2.1.

split_linear_read_side_effects_ok.1.1.1.1.1.1.2.2:

{-1}	is_linear_plain_memory?(pm') \supset OK?(linear_resolve(t'1, Read)(s'))
{-2}	union(pm' ro_addr, pm' rw_addr)(t'1)
{-3}	cons?(t'2)
{-4}	$a' \leq t'1$
{-5}	$t'1 + \text{length}(t'2) \leq a' + \text{length}(bl')$
{-6}	type_of(t'1) = type_of(a')
{-7}	reg_base((#type_of := Mem_, offset := 0#))(type_of(a')) $\leq t'1$
{-8}	$t'1 < \text{reg_size}((\#type_of := \text{Mem_}, \text{offset} := \text{max_linear_offset\#}))(type_of(a'))$
{-9}	$\text{rem}(\text{expt}(2, \text{min_page}))(\text{offset}(t'1)) + \text{length}(t'2) \leq \text{expt}(2, \text{min_page})$
{-10}	Mem?(a' type_of)
{-11}	$0 \leq a'\text{'offset}$
{-12}	$a'\text{'offset} < \text{max_linear_offset}$
{-13}	cons?[Byte](bl')
{-14}	is_linear_plain_memory?(pm')
{-15}	$(\text{address_block}(a', \text{length}(bl')) \subseteq (\text{pm}'\text{ro_addr} \cup \text{pm}'\text{rw_addr}))$
{-16}	pm' states(s')
{1}	$\text{length}[Byte](t'2) > 0$
{2}	$(\text{address_block}(\text{data}(\text{linear_resolve}(t'1, \text{Read})(s')), \text{length}(t'2)) \subseteq (\text{pm_phy}'\text{ro_addr} \cup \text{pm_phy}'\text{rw_addr}))$

Expanding the definition of length,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

C Proof scripts

This completes the proof of `split_linear_read_side_effects_ok.1.1.1.1.1.1.2.2.`

`split_linear_read_side_effects_ok.1.1.1.1.1.1.2.3:`

{-1}	<code>is_linear_plain_memory?(pm') \supset OK?(linear_resolve(t'1, Read)(s'))</code>
{-2}	<code>union(pm'ro_addr, pm'rw_addr)(t'1)</code>
{-3}	<code>cons?(t'2)</code>
{-4}	<code>a' \leq t'1</code>
{-5}	<code>t'1 + length(t'2) \leq a' + length(bl')</code>
{-6}	<code>type_of(t'1) = type_of(a')</code>
{-7}	<code>reg_base((#type_of := Mem_, offset := 0#))(type_of(a')) \leq t'1</code>
{-8}	<code>t'1 < reg_size((#type_of := Mem_, offset := max_linear_offset#))(type_of(a'))</code>
{-9}	<code>rem(expt(2, min_page))(offset(t'1)) + length(t'2) \leq expt(2, min_page)</code>
{-10}	<code>Mem?(a'type_of)</code>
{-11}	<code>0 \leq a'offset</code>
{-12}	<code>a'offset < max_linear_offset</code>
{-13}	<code>cons?[Byte](bl')</code>
{-14}	<code>is_linear_plain_memory?(pm')</code>
{-15}	<code>(address_block(a', length(bl'))) \subseteq (pm'ro_addr \cup pm'rw_addr)</code>
{-16}	<code>pm'states(s')</code>
{1}	<code>length[Byte](bl') > 0</code>
{2}	<code>(address_block(data(linear_resolve(t'1, Read)(s')), length(t'2))) \subseteq (pm_phy'ro_addr \cup pm_phy'rw_addr)</code>

Expanding the definition of length,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `split_linear_read_side_effects_ok.1.1.1.1.1.1.2.3.`

`split_linear_read_side_effects_ok.1.1.1.1.1.1.3:`

{-1}	<code>is_linear_plain_memory?(pm') \supset OK?(linear_resolve(t'1, Read)(s'))</code>
{-2}	<code>union(pm'ro_addr, pm'rw_addr)(t'1)</code>
{-3}	<code>cons?(t'2)</code>
{-4}	<code>a' \leq t'1</code>
{-5}	<code>t'1 + length(t'2) \leq a' + length(bl')</code>
{-6}	<code>type_of(t'1) = type_of(a')</code>
{-7}	<code>reg_base((#type_of := Mem_, offset := 0#))(type_of(a')) \leq t'1</code>
{-8}	<code>t'1 < reg_size((#type_of := Mem_, offset := max_linear_offset#))(type_of(a'))</code>
{-9}	<code>rem(expt(2, min_page))(offset(t'1)) + length(t'2) \leq expt(2, min_page)</code>
{-10}	<code>Mem?(a'type_of)</code>
{-11}	<code>0 \leq a'offset</code>
{-12}	<code>a'offset < max_linear_offset</code>
{-13}	<code>cons?[Byte](bl')</code>
{-14}	<code>is_linear_plain_memory?(pm')</code>
{-15}	<code>(address_block(a', length(bl'))) \subseteq (pm'ro_addr \cup pm'rw_addr)</code>
{-16}	<code>pm'states(s')</code>
{1}	<code>OK?[Physical_memory, Address](linear_resolve[Physical_memory, pm_phy](t'1, Read)(s'))</code>
{2}	<code>OK?((linear_resolve(t'1, Read) ## $(\lambda$ (pa: Memory_Address_4G)(ns: Linear_memory[Physical_memory, pm_phy]): memory_read_side_effect(pm_phy'mem)(pa, t'2, TRUE)(s')) (s'))</code>

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `split_linear_read_side_effects_ok.1.1.1.1.1.1.3.`

split_linear_read_side_effects_ok.1.1.1.1.1.2:

{-1}	cons?(t'2)
{-2}	a' ≤ t'1
{-3}	t'1 + length(t'2) ≤ a' + length(bl')
{-4}	type_of(t'1) = type_of(a')
{-5}	reg_base((#type_of := Mem_, offset := 0#))(type_of(a')) ≤ t'1
{-6}	t'1 < reg_size((#type_of := Mem_, offset := max_linear_offset#))(type_of(a'))
{-7}	rem(expt(2, min_page))(offset(t'1)) + length(t'2) ≤ expt(2, min_page)
{-8}	Mem?(a' type_of)
{-9}	0 ≤ a' offset
{-10}	a' offset < max_linear_offset
{-11}	cons?[Byte](bl')
{-12}	is_linear_plain_memory?(pm')
{-13}	(address_block(a', length(bl'))) ⊆ (pm' ro_addr ∪ pm' rw_addr)
{-14}	pm' states(s')
{1}	union(pm' ro_addr, pm' rw_addr)(t'1)
{2}	OK?((linear_resolve(t'1, Read) ## (λ (pa: Memory_Address_4G)(ns: Linear_memory[Physical_memory, pm_phy]): memory_read_side_effect(pm_phy mem)(pa, t'2, TRUE)(s')) (s')))

Hiding formulas: (-7 -12 2),

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of split_linear_read_side_effects_ok.1.1.1.1.1.2.

split_linear_read_side_effects_ok.1.1.1.1.2:

```

{-1} every(λ (t_1: [Address, list[Byte]]):
      cons?(t_1'2) ∧
      a' ≤ t_1'1 ∧
      t_1'1 + length(t_1'2) ≤ a' + length(bl') ∧
      type_of(t_1'1) = type_of(a') ∧
      reg_base(min_linear)(type_of(a')) ≤ t_1'1 ∧
      t_1'1 < reg_size(max_linear)(type_of(a')) ∧
      rem(expt(2, min_page))(offset(t_1'1)) + length(t_1'2) ≤
      expt(2, min_page))
      (split(min_page, a', bl'))
  ⊃
  every(λ (s: [Memory_Address_4G, list[Byte]]):
      cons?(s'2) ∧
      a' ≤ s'1 ∧
      s'1 + length(s'2) ≤ a' + length(bl') ∧
      type_of(s'1) = type_of(a') ∧
      reg_base(min_linear)(type_of(a')) ≤ s'1 ∧
      s'1 < reg_size(max_linear)(type_of(a')) ∧
      rem(expt(2, min_page))(offset(s'1)) + length(s'2) ≤
      expt(2, min_page))
      (split(min_page, a', bl'))
{-2} every(λ (t: [Address, list[Byte]]):
      (cons?(t'2) ∧
      a' ≤ t'1 ∧
      t'1 + length(t'2) ≤ a' + length(bl') ∧
      type_of(t'1) = type_of(a') ∧
      reg_base((#type_of := Mem_, offset := 0#))(type_of(a')) ≤ t'1 ∧
      t'1 <
      reg_size((#type_of := Mem_, offset := max_linear_offset#))(type_of(a')))
      ∧
      rem(expt(2, min_page))(offset(t'1)) + length(t'2) ≤
      expt(2, min_page))
      (split(min_page, a', bl'))
{-3} Mem?(a' type_of)
{-4} 0 ≤ a' offset
{-5} a' offset < max_linear_offset
{-6} cons?[Byte](bl')
{-7} is_linear_plain_memory?(pm')
{-8} (address_block(a', length(bl'))) ⊆ (pm' ro_addr ∪ pm' rw_addr)
{-9} pm' states(s')
-----
{1} ∀ (s: [Memory_Address_4G, list[Byte]]):
      reg_base(Mem(0))(type_of(a')) ≤ s'1 ∧
      type_of(s'1) = type_of(a') ∧
      s'1 + length(s'2) ≤ a' + length(bl') ∧ a' ≤ s'1 ∧ cons?(s'2)
      ⊃ Mem?(Mem(max_linear_offset) type_of)
{2} every(λ (e: [Memory_Address_4G, list[Byte]]):
      OK?(linear_read_side_effect_in_page(e)(s')))
      (split(min_page, a', bl'))

```

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `split_linear_read_side_effects_ok.1.1.1.1.2`.

split_linear_read_side_effects_ok.1.1.1.1.3:

{-1}	$\begin{aligned} & \text{every}(\lambda (t_1: [\text{Address}, \text{list}[\text{Byte}]]): \\ & \quad \text{cons?}(t_1'2) \wedge \\ & \quad a' \leq t_1'1 \wedge \\ & \quad t_1'1 + \text{length}(t_1'2) \leq a' + \text{length}(bl') \wedge \\ & \quad \text{type_of}(t_1'1) = \text{type_of}(a') \wedge \\ & \quad \text{reg_base}(\text{min_linear})(\text{type_of}(a')) \leq t_1'1 \wedge \\ & \quad t_1'1 < \text{reg_size}(\text{max_linear})(\text{type_of}(a')) \wedge \\ & \quad \text{rem}(\text{expt}(2, \text{min_page}))(\text{offset}(t_1'1)) + \text{length}(t_1'2) \leq \\ & \quad \text{expt}(2, \text{min_page})) \\ & \quad (\text{split}(\text{min_page}, a', bl')) \end{aligned}$
	\supset
	$\begin{aligned} & \text{every}(\lambda (s: [\text{Memory_Address_4G}, \text{list}[\text{Byte}]]): \\ & \quad \text{cons?}(s'2) \wedge \\ & \quad a' \leq s'1 \wedge \\ & \quad s'1 + \text{length}(s'2) \leq a' + \text{length}(bl') \wedge \\ & \quad \text{type_of}(s'1) = \text{type_of}(a') \wedge \\ & \quad \text{reg_base}(\text{min_linear})(\text{type_of}(a')) \leq s'1 \wedge \\ & \quad s'1 < \text{reg_size}(\text{max_linear})(\text{type_of}(a')) \wedge \\ & \quad \text{rem}(\text{expt}(2, \text{min_page}))(\text{offset}(s'1)) + \text{length}(s'2) \leq \\ & \quad \text{expt}(2, \text{min_page})) \\ & \quad (\text{split}(\text{min_page}, a', bl')) \end{aligned}$
{-2}	$\begin{aligned} & \text{every}(\lambda (t: [\text{Address}, \text{list}[\text{Byte}]]): \\ & \quad (\text{cons?}(t'2) \wedge \\ & \quad a' \leq t'1 \wedge \\ & \quad t'1 + \text{length}(t'2) \leq a' + \text{length}(bl') \wedge \\ & \quad \text{type_of}(t'1) = \text{type_of}(a') \wedge \\ & \quad \text{reg_base}((\# \text{type_of} := \text{Mem_}, \text{offset} := 0\#))(\text{type_of}(a')) \leq t'1 \wedge \\ & \quad t'1 < \\ & \quad \text{reg_size}((\# \text{type_of} := \text{Mem_}, \text{offset} := \text{max_linear_offset}\#))(\text{type_of}(a'))) \\ & \quad \wedge \\ & \quad \text{rem}(\text{expt}(2, \text{min_page}))(\text{offset}(t'1)) + \text{length}(t'2) \leq \\ & \quad \text{expt}(2, \text{min_page})) \\ & \quad (\text{split}(\text{min_page}, a', bl')) \end{aligned}$
{-3}	$\text{Mem?}(a' \text{ type_of})$
{-4}	$0 \leq a' \text{ offset}$
{-5}	$a' \text{ offset} < \text{max_linear_offset}$
{-6}	$\text{cons?}[\text{Byte}](bl')$
{-7}	$\text{is_linear_plain_memory?}(pm')$
{-8}	$(\text{address_block}(a', \text{length}(bl'))) \subseteq (pm' \text{ ro_addr} \cup pm' \text{ rw_addr})$
{-9}	$pm' \text{ states}(s')$
{1}	$\begin{aligned} & \forall (s: [\text{Memory_Address_4G}, \text{list}[\text{Byte}]]): \\ & \quad \text{type_of}(s'1) = \text{type_of}(a') \wedge \\ & \quad s'1 + \text{length}(s'2) \leq a' + \text{length}(bl') \wedge a' \leq s'1 \wedge \text{cons?}(s'2) \\ & \quad \supset \text{Mem?}(\text{Mem}(0) \text{ type_of}) \end{aligned}$
{2}	$\begin{aligned} & \text{every}(\lambda (e: [\text{Memory_Address_4G}, \text{list}[\text{Byte}]]): \\ & \quad \text{OK?}(\text{linear_read_side_effect_in_page}(e)(s')) \\ & \quad (\text{split}(\text{min_page}, a', bl')) \end{aligned}$

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of split_linear_read_side_effects_ok.1.1.1.1.3.

split_linear_read_side_effects_ok.1.1.1.2:

{-1}	$\text{every}(\lambda (t: [\text{Address}, \text{list}[\text{Byte}]]):$ $\quad (\text{cons?}(t'2) \wedge$ $\quad \quad a' \leq t'1 \wedge$ $\quad \quad t'1 + \text{length}(t'2) \leq a' + \text{length}(bl') \wedge$ $\quad \quad \text{type_of}(t'1) = \text{type_of}(a') \wedge$ $\quad \quad \text{reg_base}(\text{\#type_of := Mem_}, \text{offset := 0\#})(\text{type_of}(a')) \leq t'1 \wedge$ $\quad \quad t'1 <$ $\quad \quad \text{reg_size}(\text{\#type_of := Mem_}, \text{offset := max_linear_offset\#})(\text{type_of}(a'))$ $\quad \quad \wedge$ $\quad \quad \text{rem}(\text{expt}(2, \text{min_page})(\text{offset}(t'1)) + \text{length}(t'2) \leq$ $\quad \quad \text{expt}(2, \text{min_page}))$ $\quad \quad (\text{split}(\text{min_page}, a', bl'))$
{-2}	$\text{Mem?}(a' \text{'type_of})$
{-3}	$0 \leq a' \text{'offset}$
{-4}	$a' \text{'offset} < \text{max_linear_offset}$
{-5}	$\text{cons?}[\text{Byte}](bl')$
{-6}	$\text{is_linear_plain_memory?}(pm')$
{-7}	$(\text{address_block}(a', \text{length}(bl')) \subseteq (pm' \text{'ro_addr} \cup pm' \text{'rw_addr}))$
{-8}	$pm' \text{'states}(s')$
{1}	$\forall (t: [\text{Address}, \text{list}[\text{Byte}]]):$ $\quad \text{reg_base}(\text{Mem}(0))(\text{type_of}(a')) \leq t'1 \wedge$ $\quad \text{type_of}(t'1) = \text{type_of}(a') \wedge$ $\quad t'1 + \text{length}(t'2) \leq a' + \text{length}(bl') \wedge a' \leq t'1 \wedge \text{cons?}(t'2)$ $\quad \supset \text{Mem?}(\text{Mem}(\text{max_linear_offset}) \text{'type_of})$
{2}	$\text{every}(\lambda (e: [\text{Memory_Address_4G}, \text{list}[\text{Byte}]]):$ $\quad \text{OK?}(\text{linear_read_side_effect_in_page}(e)(s'))$ $\quad (\text{split}(\text{min_page}, a', bl'))$

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `split_linear_read_side_effects_ok.1.1.1.2`.

split_linear_read_side_effects_ok.1.1.1.3:

{-1}	$\text{every}(\lambda (t: [\text{Address}, \text{list}[\text{Byte}]]):$ $\begin{aligned} & (\text{cons?}(t'2) \wedge \\ & \quad a' \leq t'1 \wedge \\ & \quad t'1 + \text{length}(t'2) \leq a' + \text{length}(bl') \wedge \\ & \quad \text{type_of}(t'1) = \text{type_of}(a') \wedge \\ & \quad \text{reg_base}((\#\text{type_of} := \text{Mem_}, \text{offset} := 0\#))(\text{type_of}(a')) \leq t'1 \wedge \\ & \quad t'1 < \\ & \quad \text{reg_size}((\#\text{type_of} := \text{Mem_}, \text{offset} := \text{max_linear_offset}\#))(\text{type_of}(a'))) \\ & \wedge \\ & \text{rem}(\text{expt}(2, \text{min_page}))(\text{offset}(t'1)) + \text{length}(t'2) \leq \\ & \quad \text{expt}(2, \text{min_page}) \\ & \quad (\text{split}(\text{min_page}, a', bl')) \end{aligned}$
{-2}	$\text{Mem?}(a' \text{'type_of})$
{-3}	$0 \leq a' \text{'offset}$
{-4}	$a' \text{'offset} < \text{max_linear_offset}$
{-5}	$\text{cons?}[\text{Byte}](bl')$
{-6}	$\text{is_linear_plain_memory?}(pm')$
{-7}	$(\text{address_block}(a', \text{length}(bl'))) \subseteq (pm' \text{'ro_addr} \cup pm' \text{'rw_addr})$
{-8}	$pm' \text{'states}(s')$
{1}	$\forall (t: [\text{Address}, \text{list}[\text{Byte}]]):$ $\begin{aligned} & \text{type_of}(t'1) = \text{type_of}(a') \wedge \\ & \quad t'1 + \text{length}(t'2) \leq a' + \text{length}(bl') \wedge a' \leq t'1 \wedge \text{cons?}(t'2) \\ & \quad \supset \text{Mem?}(\text{Mem}(0) \text{'type_of}) \end{aligned}$
{2}	$\text{every}(\lambda (e: [\text{Memory_Address_4G}, \text{list}[\text{Byte}]]):$ $\begin{aligned} & \text{OK?}(\text{linear_read_side_effect_in_page}(e)(s')) \\ & \quad (\text{split}(\text{min_page}, a', bl')) \end{aligned}$

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of split_linear_read_side_effects_ok.1.1.1.3.

split_linear_read_side_effects_ok.1.1.2:

{-1}	$\text{every}(\lambda (t: [\text{Address}, \text{list}[\text{Byte}]]):$ $\text{cons?}(t'2) \wedge$ $a' \leq t'1 \wedge t'1 + \text{length}(t'2) \leq a' + \text{length}(bl')$ $(\text{split}(\text{min_page}, a', bl'))$ \wedge $\text{every}(\lambda (t: [\text{Address}, \text{list}[\text{Byte}]]):$ $\text{type_of}(t'1) = \text{type_of}(a') \wedge$ $\text{reg_base}(\text{min_linear})(\text{type_of}(a')) \leq t'1 \wedge$ $t'1 < \text{reg_size}(\text{max_linear})(\text{type_of}(a'))$ $(\text{split}(\text{min_page}, a', bl'))$ \supset $\text{every}(\lambda (t_1: [\text{Address}, \text{list}[\text{Byte}]]):$ $(\text{cons?}(t_1'2) \wedge$ $a' \leq t_1'1 \wedge t_1'1 + \text{length}(t_1'2) \leq a' + \text{length}(bl'))$ \wedge $\text{type_of}(t_1'1) = \text{type_of}(a') \wedge$ $\text{reg_base}(\text{min_linear})(\text{type_of}(a')) \leq t_1'1 \wedge$ $t_1'1 < \text{reg_size}(\text{max_linear})(\text{type_of}(a'))$ $(\text{split}(\text{min_page}, a', bl'))$
{-2}	$\text{every}(\lambda (a_2: \text{Address}, bl_2: \text{list}[\text{Byte}]): \text{cons?}(bl_2)(\text{split}(\text{min_page}, a', bl')) \wedge$ $\text{every}(\lambda (a_2: \text{Address}, bl_2: \text{list}[\text{Byte}]):$ $a' \leq a_2 \wedge a_2 + \text{length}(bl_2) \leq a' + \text{length}(bl'))$ $(\text{split}(\text{min_page}, a', bl'))$ \supset $\text{every}(\lambda (t: [\text{Address}, \text{list}[\text{Byte}]]):$ $\text{cons?}(t'2) \wedge$ $a' \leq t'1 \wedge t'1 + \text{length}(t'2) \leq a' + \text{length}(bl'))$ $(\text{split}(\text{min_page}, a', bl'))$
{-3}	$\text{every}(\lambda (a_2: \text{Address}, bl_2: \text{list}[\text{Byte}]):$ $\text{rem}(\text{expt}(2, \text{min_page}))(\text{offset}(a_2)) + \text{length}(bl_2) \leq \text{expt}(2, \text{min_page}))$ $(\text{split}(\text{min_page}, a', bl'))$
{-4}	$\text{every}(\lambda (t: [\text{Address}, \text{list}[\text{Byte}]]):$ $\text{type_of}(t'1) = \text{type_of}(a') \wedge$ $\text{reg_base}((\# \text{type_of} := \text{Mem_}, \text{offset} := 0\#))(\text{type_of}(a')) \leq t'1 \wedge$ $t'1 < \text{reg_size}((\# \text{type_of} := \text{Mem_}, \text{offset} := \text{max_linear_offset}\#))(\text{type_of}(a'))$ $(\text{split}(\text{min_page}, a', bl'))$
{-5}	$\text{every}(\lambda (a_2: \text{Address}, bl_2: \text{list}[\text{Byte}]): \text{cons?}(bl_2)(\text{split}(\text{min_page}, a', bl'))$
{-6}	$\text{every}(\lambda (a_2: \text{Address}, bl_2: \text{list}[\text{Byte}]):$ $a' \leq a_2 \wedge a_2 + \text{length}(bl_2) \leq a' + \text{length}(bl'))$ $(\text{split}(\text{min_page}, a', bl'))$
{-7}	$\text{Mem?}(a' \text{ type_of})$
{-8}	$0 \leq a' \text{ offset}$
{-9}	$a' \text{ offset} < \text{max_linear_offset}$
{-10}	$\text{every}(\lambda (x: \text{number}):$ $\text{number_field_pred}(x) \wedge$ $\text{real_pred}(x) \wedge$ $\text{rational_pred}(x) \wedge \text{integer_pred}(x) \wedge x \geq 0 \wedge x < \text{max_byte})$ (bl')
{-11}	$\text{cons?}[\text{Byte}](bl')$
{-12}	$\text{is_linear_plain_memory?}(pm')$
{-13}	$(\text{address_block}(a', \text{length}(bl')) \subseteq (pm' \text{ ro_addr} \cup pm' \text{ rw_addr}))$
{-14}	$pm' \text{ states}(s')$
{1}	$\forall (t_1: [\text{Address}, \text{list}[\text{Byte}]]):$ $\text{reg_base}(\text{Mem}(0))(\text{type_of}(a')) \leq t_1'1 \wedge$ $\text{type_of}(t_1'1) = \text{type_of}(a') \wedge$ $t_1'1 + \text{length}(t_1'2) \leq a' + \text{length}(bl') \wedge$ $a' \leq t_1'1 \wedge \text{cons?}(t_1'2)$ $\supset \text{Mem?}(\text{Mem}(\text{max_linear_offset}) \text{ type_of})$
{2}	$\text{every}(\lambda (e: [\text{Memory_Address_4G}, \text{list}[\text{Byte}]]):$ $\text{OK?}(\text{linear_read_side_effect_in_page}(e)(s'))$ $(\text{split}(\text{min_page}, a', bl'))$

C.117 Proofs for Linear_Memory_Properties (challenge-linear.pvs)

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `split_linear_read_side_effects_ok.1.1.2`.

split_linear_read_side_effects_ok.1.1.3:

```

{-1} every(λ (t: [Address, list[Byte]]):
      cons?(t'2) ∧
      a' ≤ t'1 ∧ t'1 + length(t'2) ≤ a' + length(bl')
      (split(min_page, a', bl'))
      ∧
      every(λ (t: [Address, list[Byte]]):
            type_of(t'1) = type_of(a') ∧
            reg_base(min_linear)(type_of(a')) ≤ t'1 ∧
            t'1 < reg_size(max_linear)(type_of(a'))
            (split(min_page, a', bl'))
            ⊃
            every(λ (t_1: [Address, list[Byte]]):
                  (cons?(t_1'2) ∧
                   a' ≤ t_1'1 ∧ t_1'1 + length(t_1'2) ≤ a' + length(bl'))
                  ∧
                  type_of(t_1'1) = type_of(a') ∧
                  reg_base(min_linear)(type_of(a')) ≤ t_1'1 ∧
                  t_1'1 < reg_size(max_linear)(type_of(a'))
                  (split(min_page, a', bl'))
            )
      )
      (split(min_page, a', bl'))
{-2} every(λ (a_2: Address, bl2: list[Byte]): cons?(bl2))(split(min_page, a', bl')) ∧
      every(λ (a_2: Address, bl2: list[Byte]):
            a' ≤ a_2 ∧ a_2 + length(bl2) ≤ a' + length(bl')
            (split(min_page, a', bl'))
            ⊃
            every(λ (t: [Address, list[Byte]]):
                  cons?(t'2) ∧
                  a' ≤ t'1 ∧ t'1 + length(t'2) ≤ a' + length(bl')
                  (split(min_page, a', bl'))
            )
      )
      (split(min_page, a', bl'))
{-3} every(λ (a_2: Address, bl2: list[Byte]):
      rem(expt(2, min_page))(offset(a_2)) + length(bl2) ≤ expt(2, min_page))
      (split(min_page, a', bl'))
{-4} every(λ (t: [Address, list[Byte]]):
      type_of(t'1) = type_of(a') ∧
      reg_base((#type_of := Mem_, offset := 0#))(type_of(a')) ≤ t'1 ∧
      t'1 < reg_size((#type_of := Mem_, offset := max_linear_offset#))(type_of(a'))
      (split(min_page, a', bl'))
{-5} every(λ (a_2: Address, bl2: list[Byte]): cons?(bl2))(split(min_page, a', bl'))
{-6} every(λ (a_2: Address, bl2: list[Byte]):
      a' ≤ a_2 ∧ a_2 + length(bl2) ≤ a' + length(bl')
      (split(min_page, a', bl'))
{-7} Mem?(a' type_of)
{-8} 0 ≤ a' offset
{-9} a' offset < max_linear_offset
{-10} every(λ (x: number):
      number_field_pred(x) ∧
      real_pred(x) ∧
      rational_pred(x) ∧ integer_pred(x) ∧ x ≥ 0 ∧ x < max_byte)
      (bl')
{-11} cons?[Byte](bl')
{-12} is_linear_plain_memory?(pm')
{-13} (address_block(a', length(bl')) ⊆ (pm'ro_addr ∪ pm'rw_addr))
{-14} pm' states(s')


---


{1} ∀ (t_1: [Address, list[Byte]]):
      type_of(t_1'1) = type_of(a') ∧
      t_1'1 + length(t_1'2) ≤ a' + length(bl') ∧
      a' ≤ t_1'1 ∧ cons?(t_1'2)
      ⊃ Mem?(Mem(0) type_of)
{2} every(λ (e: [Memory_Address_4G, list[Byte]]):
      OK?(linear_read_side_effect_in_page(e)(s')))
      (split(min_page, a', bl'))

```

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `split_linear_read_side_effects_ok.1.1.3`.

`split_linear_read_side_effects_ok.1.2`:

{-1}	$\text{every}(\lambda (a_2: \text{Address}, \text{bl2}: \text{list}[\text{Byte}]): \text{cons}?(\text{bl2}))(\text{split}(\text{min_page}, a', \text{bl}')) \wedge$ $\text{every}(\lambda (a_2: \text{Address}, \text{bl2}: \text{list}[\text{Byte}]):$ $a' \leq a_2 \wedge a_2 + \text{length}(\text{bl2}) \leq a' + \text{length}(\text{bl}'))$ $(\text{split}(\text{min_page}, a', \text{bl}'))$ \supset $\text{every}(\lambda (t: [\text{Address}, \text{list}[\text{Byte}]]):$ $\text{cons}?(t'2) \wedge$ $a' \leq t'1 \wedge t'1 + \text{length}(t'2) \leq a' + \text{length}(\text{bl}'))$ $(\text{split}(\text{min_page}, a', \text{bl}'))$
{-2}	$\text{every}(\lambda (a_2: \text{Address}, \text{bl2}: \text{list}[\text{Byte}]):$ $\text{rem}(\text{expt}(2, \text{min_page}))(\text{offset}(a_2)) + \text{length}(\text{bl2}) \leq \text{expt}(2, \text{min_page}))$ $(\text{split}(\text{min_page}, a', \text{bl}'))$
{-3}	$\text{every}(\lambda (t: [\text{Address}, \text{list}[\text{Byte}]]):$ $\text{type_of}(t'1) = \text{type_of}(a') \wedge$ $\text{reg_base}(\text{\#type_of} := \text{Mem_}, \text{offset} := 0\#)(\text{type_of}(a')) \leq t'1 \wedge$ $t'1 < \text{reg_size}(\text{\#type_of} := \text{Mem_}, \text{offset} := \text{max_linear_offset\#})(\text{type_of}(a')))$ $(\text{split}(\text{min_page}, a', \text{bl}'))$
{-4}	$\text{every}(\lambda (a_2: \text{Address}, \text{bl2}: \text{list}[\text{Byte}]): \text{cons}?(\text{bl2}))(\text{split}(\text{min_page}, a', \text{bl}'))$
{-5}	$\text{every}(\lambda (a_2: \text{Address}, \text{bl2}: \text{list}[\text{Byte}]):$ $a' \leq a_2 \wedge a_2 + \text{length}(\text{bl2}) \leq a' + \text{length}(\text{bl}'))$ $(\text{split}(\text{min_page}, a', \text{bl}'))$
{-6}	$\text{Mem}?(a' \text{ type_of})$
{-7}	$0 \leq a' \text{ offset}$
{-8}	$a' \text{ offset} < \text{max_linear_offset}$
{-9}	$\text{every}(\lambda (x: \text{number}):$ $\text{number_field_pred}(x) \wedge$ $\text{real_pred}(x) \wedge$ $\text{rational_pred}(x) \wedge \text{integer_pred}(x) \wedge x \geq 0 \wedge x < \text{max_byte})$ (bl')
{-10}	$\text{cons}?[\text{Byte}](\text{bl}')$
{-11}	$\text{is_linear_plain_memory}?(\text{pm}')$
{-12}	$(\text{address_block}(a', \text{length}(\text{bl}')) \subseteq (\text{pm}' \text{ ro_addr} \cup \text{pm}' \text{ rw_addr}))$
{-13}	$\text{pm}' \text{ states}(s')$
{1}	$\forall (t: [\text{Address}, \text{list}[\text{Byte}]]):$ $\text{reg_base}(\text{Mem}(0))(\text{type_of}(a')) \leq t'1 \wedge \text{type_of}(t'1) = \text{type_of}(a') \supset$ $\text{Mem}?(\text{Mem}(\text{max_linear_offset}) \text{ type_of})$
{2}	$\text{every}(\lambda (e: [\text{Memory_Address_4G}, \text{list}[\text{Byte}]]):$ $\text{OK}?(\text{linear_read_side_effect_in_page}(e)(s'))$ $(\text{split}(\text{min_page}, a', \text{bl}'))$

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `split_linear_read_side_effects_ok.1.2`.

split_linear_read_side_effects_ok.1.3:

{-1}	$\text{every}(\lambda (a_2: \text{Address}, \text{bl2}: \text{list}[\text{Byte}]): \text{cons}?(a_2))(\text{split}(\text{min_page}, a', \text{bl}')) \wedge$ $\text{every}(\lambda (a_2: \text{Address}, \text{bl2}: \text{list}[\text{Byte}]):$ $a' \leq a_2 \wedge a_2 + \text{length}(\text{bl2}) \leq a' + \text{length}(\text{bl}'))$ $(\text{split}(\text{min_page}, a', \text{bl}'))$ \supset $\text{every}(\lambda (t: [\text{Address}, \text{list}[\text{Byte}]]):$ $\text{cons}?(t'2) \wedge$ $a' \leq t'1 \wedge t'1 + \text{length}(t'2) \leq a' + \text{length}(\text{bl}'))$ $(\text{split}(\text{min_page}, a', \text{bl}'))$
{-2}	$\text{every}(\lambda (a_2: \text{Address}, \text{bl2}: \text{list}[\text{Byte}]):$ $\text{rem}(\text{expt}(2, \text{min_page}))(\text{offset}(a_2)) + \text{length}(\text{bl2}) \leq \text{expt}(2, \text{min_page}))$ $(\text{split}(\text{min_page}, a', \text{bl}'))$
{-3}	$\text{every}(\lambda (t: [\text{Address}, \text{list}[\text{Byte}]]):$ $\text{type_of}(t'1) = \text{type_of}(a') \wedge$ $\text{reg_base}((\# \text{type_of} := \text{Mem_}, \text{offset} := 0\#))(\text{type_of}(a')) \leq t'1 \wedge$ $t'1 < \text{reg_size}((\# \text{type_of} := \text{Mem_}, \text{offset} := \text{max_linear_offset}\#))(\text{type_of}(a')))$ $(\text{split}(\text{min_page}, a', \text{bl}'))$
{-4}	$\text{every}(\lambda (a_2: \text{Address}, \text{bl2}: \text{list}[\text{Byte}]): \text{cons}?(a_2))(\text{split}(\text{min_page}, a', \text{bl}'))$
{-5}	$\text{every}(\lambda (a_2: \text{Address}, \text{bl2}: \text{list}[\text{Byte}]):$ $a' \leq a_2 \wedge a_2 + \text{length}(\text{bl2}) \leq a' + \text{length}(\text{bl}'))$ $(\text{split}(\text{min_page}, a', \text{bl}'))$
{-6}	$\text{Mem}?(a' \text{ type_of})$
{-7}	$0 \leq a' \text{ offset}$
{-8}	$a' \text{ offset} < \text{max_linear_offset}$
{-9}	$\text{every}(\lambda (x: \text{number}):$ $\text{number_field_pred}(x) \wedge$ $\text{real_pred}(x) \wedge$ $\text{rational_pred}(x) \wedge \text{integer_pred}(x) \wedge x \geq 0 \wedge x < \text{max_byte})$ (bl')
{-10}	$\text{cons}?[\text{Byte}](\text{bl}')$
{-11}	$\text{is_linear_plain_memory}?(pm')$
{-12}	$(\text{address_block}(a', \text{length}(\text{bl}')) \subseteq (pm' \text{ ro_addr} \cup pm' \text{ rw_addr}))$
{-13}	$pm' \text{ states}(s')$
{1}	$\forall (t: [\text{Address}, \text{list}[\text{Byte}]]):$ $\text{type_of}(t'1) = \text{type_of}(a') \supset \text{Mem}?(Mem(0) \text{ type_of})$
{2}	$\text{every}(\lambda (e: [\text{Memory_Address_4G}, \text{list}[\text{Byte}]]):$ $\text{OK}?(linear_read_side_effect_in_page(e)(s'))$ $(\text{split}(\text{min_page}, a', \text{bl}'))$

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `split_linear_read_side_effects_ok.1.3`.

split_linear_read_side_effects_ok.2:

<pre> {-1} every(λ (a2: Address, bl2: list[Byte]): rem(expt(2, min_page))(offset(a2)) + length(bl2) ≤ expt(2, min_page)) (split(min_page, a', bl')) {-2} every(λ (a2: Address, bl2: list[Byte]): cons?(bl2))(split(min_page, a', bl')) {-3} every(λ (a2: Address, bl2: list[Byte]): a' ≤ a2 ∧ a2 + length(bl2) ≤ a' + length(bl')) (split(min_page, a', bl')) {-4} Mem?(a' type_of) {-5} 0 ≤ a' offset {-6} a' offset < max_linear_offset {-7} every(λ (x: number): number_field_pred(x) ∧ real_pred(x) ∧ rational_pred(x) ∧ integer_pred(x) ∧ x ≥ 0 ∧ x < max_byte) (bl') {-8} cons?[Byte](bl') {-9} is_linear_plain_memory?(pm') {-10} (address_block(a', length(bl')) ⊆ (pm' ro_addr ∪ pm' rw_addr)) {-11} pm' states(s') </pre>	<hr style="border: 0.5px solid black;"/> <pre> {1} a' + length(bl') ≤ reg_size(#type_of := Mem_, offset := max_linear_offset#)(type_of(a')) {2} every(λ (e: [Memory_Address_4G, list[Byte]]): OK?(linear_read_side_effect_in_page(e)(s')) (split(min_page, a', bl')) </pre>
---	---

Hiding formulas: (-1 -2 -3 -7 2),

Expanding the definition of length,

Using lemma pm_memory_addr,

Simplifying, rewriting, and recording with decision procedures,

Hiding formulas: -6,

Expanding the definition of subset?,

Installing automatic rewrites from: (bus_width!)

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of split_linear_read_side_effects_ok.2.

split_linear_read_side_effects_ok.3:

{-1}	every(λ (a_2 : Address, bl2: list[Byte]): rem(expt(2, min_page))(offset(a_2)) + length(bl2) \leq expt(2, min_page)) (split(min_page, a' , bl'))
{-2}	every(λ (a_2 : Address, bl2: list[Byte]): cons?(bl2))(split(min_page, a' , bl'))
{-3}	every(λ (a_2 : Address, bl2: list[Byte]): $a' \leq a_2 \wedge a_2 + \text{length}(bl2) \leq a' + \text{length}(bl')$ (split(min_page, a' , bl'))
{-4}	Mem?(a' 'type_of)
{-5}	$0 \leq a'$ 'offset
{-6}	a' 'offset < max_linear_offset
{-7}	every(λ (x : number): number_field_pred(x) \wedge real_pred(x) \wedge rational_pred(x) \wedge integer_pred(x) \wedge $x \geq 0 \wedge x < \text{max_byte}$) (bl')
{-8}	cons?[Byte](bl')
{-9}	is_linear_plain_memory?(pm')
{-10}	(address_block(a' , length(bl')) \subseteq (pm'ro_addr \cup pm'rw_addr))
{-11}	pm' states(s')
{1}	reg_base((#type_of := Mem_, offset := 0#))(type_of(a')) $\leq a'$
{2}	every(λ (e : [Memory_Address_4G, list[Byte]]): OK?(linear_read_side_effect_in_page(e)(s')) (split(min_page, a' , bl'))

Expanding the definition of reg_base,

Expanding the definition of \leq ,

which is trivially true.

This completes the proof of split_linear_read_side_effects_ok.3.

Q.E.D.

C.117.79

Linear_Memory_Properties.split_linear_write_side_effects_ok

Terse proof for split_linear_write_side_effects_ok.

split_linear_write_side_effects_ok:

{1}	\forall (pm: Plain_Memory[Linear_memory], s : Linear_memory[Physical_memory, pm_phy], a : Memory_Address_4G, bl: (cons?[Byte])): is_linear_plain_memory?(pm) \wedge (address_block(a , length(bl)) \subseteq pm'rw_addr) \wedge pm' states(s) \supset every(λ (e : [Memory_Address_4G, list[Byte]]): OK?(linear_write_side_effect_in_page(e)(s)) (split(min_page, a , bl))
-----	--

Repeatedly Skolemizing and flattening,

Installing automatic rewrites from: Mem!! max_linear!! min_linear!!

Using lemma split_range,

Using lemma split_no_null,

Using lemma split_type,

Using lemma split_pair_cross_size,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 3 subgoals:

split_linear_write_side_effects_ok.1:

{-1}	every(λ (a_2 : Address, $bl2$: list[Byte]): rem(expt(2, min_page))(offset(a_2)) + length($bl2$) \leq expt(2, min_page)) (split(min_page, a' , bl'))
{-2}	every(λ (t : [Address, list[Byte]]): type_of($t'1$) = type_of(a') \wedge reg_base((#type_of := Mem_, offset := 0#))(type_of(a')) \leq $t'1$ \wedge $t'1$ < reg_size((#type_of := Mem_, offset := max_linear_offset#))(type_of(a'))) (split(min_page, a' , bl'))
{-3}	every(λ (a_2 : Address, $bl2$: list[Byte]): cons?($bl2$))(split(min_page, a' , bl'))
{-4}	every(λ (a_2 : Address, $bl2$: list[Byte]): $a' \leq a_2$ \wedge a_2 + length($bl2$) \leq a' + length(bl')) (split(min_page, a' , bl'))
{-5}	Mem?(a' 'type_of)
{-6}	$0 \leq a'$ 'offset
{-7}	a' 'offset < max_linear_offset
{-8}	every(λ (x : number): number_field_pred(x) \wedge real_pred(x) \wedge rational_pred(x) \wedge integer_pred(x) \wedge $x \geq 0$ \wedge x < max_byte) (bl')
{-9}	cons?[Byte](bl')
{-10}	is_linear_plain_memory?(pm')
{-11}	(address_block(a' , length(bl')) \subseteq pm' 'rw_addr)
{-12}	pm' 'states(s')
{1}	every(λ (e : [Memory_Address_4G, list[Byte]]): OK?(linear_write_side_effect_in_page(e)(s'))) (split(min_page, a' , bl'))

Using lemma every_conjunct_left,

Using lemma every_conjunct_left,

we get 3 subgoals:

split_linear_write_side_effects_ok.1.1:

```

{-1} every(λ (t: [Address, list[Byte]]):
      cons?(t'2) ∧
      a' ≤ t'1 ∧ t'1 + length(t'2) ≤ a' + length(bl')
      (split(min_page, a', bl'))
    ∧
    every(λ (t: [Address, list[Byte]]):
      type_of(t'1) = type_of(a') ∧
      reg_base(min_linear)(type_of(a')) ≤ t'1 ∧
      t'1 < reg_size(max_linear)(type_of(a'))
      (split(min_page, a', bl'))
    ⊃
    every(λ (t_1: [Address, list[Byte]]):
      (cons?(t_1'2) ∧
      a' ≤ t_1'1 ∧ t_1'1 + length(t_1'2) ≤ a' + length(bl'))
      ∧
      type_of(t_1'1) = type_of(a') ∧
      reg_base(min_linear)(type_of(a')) ≤ t_1'1 ∧
      t_1'1 < reg_size(max_linear)(type_of(a'))
      (split(min_page, a', bl'))
    {-2} every(λ (a_2: Address, bl2: list[Byte]): cons?(bl2))(split(min_page, a', bl')) ∧
    every(λ (a_2: Address, bl2: list[Byte]):
      a' ≤ a_2 ∧ a_2 + length(bl2) ≤ a' + length(bl')
      (split(min_page, a', bl'))
    ⊃
    every(λ (t: [Address, list[Byte]]):
      cons?(t'2) ∧
      a' ≤ t'1 ∧ t'1 + length(t'2) ≤ a' + length(bl')
      (split(min_page, a', bl'))
    {-3} every(λ (a_2: Address, bl2: list[Byte]):
      rem(expt(2, min_page))(offset(a_2)) + length(bl2) ≤ expt(2, min_page))
      (split(min_page, a', bl'))
    {-4} every(λ (t: [Address, list[Byte]]):
      type_of(t'1) = type_of(a') ∧
      reg_base((#type_of := Mem_, offset := 0#))(type_of(a')) ≤ t'1 ∧
      t'1 < reg_size((#type_of := Mem_, offset := max_linear_offset#))(type_of(a'))
      (split(min_page, a', bl'))
    {-5} every(λ (a_2: Address, bl2: list[Byte]): cons?(bl2))(split(min_page, a', bl'))
    {-6} every(λ (a_2: Address, bl2: list[Byte]):
      a' ≤ a_2 ∧ a_2 + length(bl2) ≤ a' + length(bl')
      (split(min_page, a', bl'))
    {-7} Mem?(a' type_of)
    {-8} 0 ≤ a' offset
    {-9} a' offset < max_linear_offset
    {-10} every(λ (x: number):
      number_field_pred(x) ∧
      real_pred(x) ∧
      rational_pred(x) ∧ integer_pred(x) ∧ x ≥ 0 ∧ x < max_byte)
      (bl')
    {-11} cons?[Byte](bl')
    {-12} is_linear_plain_memory?(pm')
    {-13} (address_block(a', length(bl')) ⊆ pm'rw_addr)
    {-14} pm' states(s')
  }
  {1} every(λ (e: [Memory_Address_4G, list[Byte]]):
    OK?(linear_write_side_effect_in_page(e)(s'))
    (split(min_page, a', bl'))

```

Using lemma `every_conjunct_left`,

we get 3 subgoals:

split_linear_write_side_effects_ok.1.1.1:

```

{-1} every(λ (t_1: [Address, list[Byte]]):
  cons?(t_1'2) ∧
  a' ≤ t_1'1 ∧
  t_1'1 + length(t_1'2) ≤ a' + length(bl') ∧
  type_of(t_1'1) = type_of(a') ∧
  reg_base(min_linear)(type_of(a')) ≤ t_1'1 ∧
  t_1'1 < reg_size(max_linear)(type_of(a')))
  (split(min_page, a', bl'))
  ∧
  every(λ (a_2: Address, bl2: list[Byte]):
    rem(expt(2, min_page))(offset(a_2)) + length(bl2) ≤
    expt(2, min_page))
    (split(min_page, a', bl'))
  ⊃
  every(λ (t: [Address, list[Byte]]):
    (cons?(t'2) ∧
     a' ≤ t'1 ∧
     t'1 + length(t'2) ≤ a' + length(bl') ∧
     type_of(t'1) = type_of(a') ∧
     reg_base(min_linear)(type_of(a')) ≤ t'1 ∧
     t'1 < reg_size(max_linear)(type_of(a')))
     ∧
     rem(expt(2, min_page))(offset(t'1)) + length(t'2) ≤
     expt(2, min_page))
    (split(min_page, a', bl'))
  {-2} every(λ (t: [Address, list[Byte]]):
    cons?(t'2) ∧
    a' ≤ t'1 ∧ t'1 + length(t'2) ≤ a' + length(bl'))
    (split(min_page, a', bl'))
    ∧
    every(λ (t: [Address, list[Byte]]):
      type_of(t'1) = type_of(a') ∧
      reg_base(min_linear)(type_of(a')) ≤ t'1 ∧
      t'1 < reg_size(max_linear)(type_of(a')))
      (split(min_page, a', bl'))
    ⊃
    every(λ (t_1: [Address, list[Byte]]):
      (cons?(t_1'2) ∧
       a' ≤ t_1'1 ∧ t_1'1 + length(t_1'2) ≤ a' + length(bl'))
      ∧
      type_of(t_1'1) = type_of(a') ∧
      reg_base(min_linear)(type_of(a')) ≤ t_1'1 ∧
      t_1'1 < reg_size(max_linear)(type_of(a')))
      (split(min_page, a', bl'))
  {-3} every(λ (a_2: Address, bl2: list[Byte]): cons?(bl2))(split(min_page, a', bl')) ∧
  every(λ (a_2: Address, bl2: list[Byte]):
    a' ≤ a_2 ∧ a_2 + length(bl2) ≤ a' + length(bl'))
    (split(min_page, a', bl'))
  ⊃
  every(λ (t: [Address, list[Byte]]):
    cons?(t'2) ∧
    a' ≤ t'1 ∧ t'1 + length(t'2) ≤ a' + length(bl'))
    (split(min_page, a', bl'))
  {-4} every(λ (a_2: Address, bl2: list[Byte]):
    rem(expt(2, min_page))(offset(a_2)) + length(bl2) ≤ expt(2, min_page))
    (split(min_page, a', bl'))
  {-5} every(λ (t: [Address, list[Byte]]):
    type_of(t'1) = type_of(a') ∧
    reg_base((#type_of := Mem_, offset := 0#))(type_of(a')) ≤ t'1 ∧
    t'1 < reg_size((#type_of := Mem_, offset := max_linear_offset#))(type_of(a')))
    (split(min_page, a', bl'))
  {-6} every(λ (a_2: Address, bl2: list[Byte]): cons?(bl2))(split(min_page, a', bl'))
  {-7} every(λ (a_2: Address, bl2: list[Byte]):

```

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Hiding formulas: (-1 -3 -4 -5 -6 -7 -11),

Installing automatic rewrites from: (##! expr_2_super! expr_2_super_res!)

Using lemma every_extend[[Address, list[Byte]], [Memory_Address_4G, list[Byte]]],

we get 3 subgoals:

split_linear_write_side_effects_ok.1.1.1.1:

<pre> {-1} every(λ (t_1: [Address, list[Byte]]): cons?(t_1'2) ∧ a' ≤ t_1'1 ∧ t_1'1 + length(t_1'2) ≤ a' + length(bl') ∧ type_of(t_1'1) = type_of(a') ∧ reg_base(min_linear)(type_of(a')) ≤ t_1'1 ∧ t_1'1 < reg_size(max_linear)(type_of(a')) ∧ rem(expt(2, min_page))(offset(t_1'1)) + length(t_1'2) ≤ expt(2, min_page) (split(min_page, a', bl')) ⊃ every(λ (s: [Memory_Address_4G, list[Byte]]): cons?(s'2) ∧ a' ≤ s'1 ∧ s'1 + length(s'2) ≤ a' + length(bl') ∧ type_of(s'1) = type_of(a') ∧ reg_base(min_linear)(type_of(a')) ≤ s'1 ∧ s'1 < reg_size(max_linear)(type_of(a')) ∧ rem(expt(2, min_page))(offset(s'1)) + length(s'2) ≤ expt(2, min_page) (split(min_page, a', bl'))) {-2} every(λ (t: [Address, list[Byte]]): (cons?(t'2) ∧ a' ≤ t'1 ∧ t'1 + length(t'2) ≤ a' + length(bl') ∧ type_of(t'1) = type_of(a') ∧ reg_base((#type_of := Mem_, offset := 0#))(type_of(a')) ≤ t'1 ∧ t'1 < reg_size((#type_of := Mem_, offset := max_linear_offset#))(type_of(a')) ∧ rem(expt(2, min_page))(offset(t'1)) + length(t'2) ≤ expt(2, min_page) (split(min_page, a', bl'))) {-3} Mem?(a' type_of) {-4} 0 ≤ a' offset {-5} a' offset < max_linear_offset {-6} cons?[Byte](bl') {-7} is_linear_plain_memory?(pm') {-8} (address_block(a', length(bl')) ⊆ pm'rw_addr) {-9} pm' states(s') </pre>	<pre> {1} every(λ (e: [Memory_Address_4G, list[Byte]]): OK?(linear_write_side_effect_in_page(e)(s')) (split(min_page, a', bl')) </pre>
--	---

Using lemma every_implied,

we get 3 subgoals:

split_linear_write_side_effects_ok.1.1.1.1.1:

```

{-1}  (∀ (t: [Memory_Address_4G, list[Byte]]):
      cons?(t'2) ∧
      a' ≤ t'1 ∧
      t'1 + length(t'2) ≤ a' + length(bl') ∧
      type_of(t'1) = type_of(a') ∧
      reg_base(min_linear)(type_of(a')) ≤ t'1 ∧
      t'1 < reg_size(max_linear)(type_of(a')) ∧
      rem(expt(2, min_page))(offset(t'1)) + length(t'2) ≤
      expt(2, min_page)
    ⊃ OK?(linear_write_side_effect_in_page(t)(s')))
  ∧
  every(λ (s: [Memory_Address_4G, list[Byte]]):
      cons?(s'2) ∧
      a' ≤ s'1 ∧
      s'1 + length(s'2) ≤ a' + length(bl') ∧
      type_of(s'1) = type_of(a') ∧
      reg_base(min_linear)(type_of(a')) ≤ s'1 ∧
      s'1 < reg_size(max_linear)(type_of(a')) ∧
      rem(expt(2, min_page))(offset(s'1)) + length(s'2) ≤
      expt(2, min_page)
    (split(min_page, a', bl')))
  ⊃
  every(λ (e: [Memory_Address_4G, list[Byte]]):
      OK?(linear_write_side_effect_in_page(e)(s')))
    (split(min_page, a', bl'))
{-2}  every(λ (t_1: [Address, list[Byte]]):
      cons?(t_1'2) ∧
      a' ≤ t_1'1 ∧
      t_1'1 + length(t_1'2) ≤ a' + length(bl') ∧
      type_of(t_1'1) = type_of(a') ∧
      reg_base(min_linear)(type_of(a')) ≤ t_1'1 ∧
      t_1'1 < reg_size(max_linear)(type_of(a')) ∧
      rem(expt(2, min_page))(offset(t_1'1)) + length(t_1'2) ≤
      expt(2, min_page)
    (split(min_page, a', bl')))
  ⊃
  every(λ (s: [Memory_Address_4G, list[Byte]]):
      cons?(s'2) ∧
      a' ≤ s'1 ∧
      s'1 + length(s'2) ≤ a' + length(bl') ∧
      type_of(s'1) = type_of(a') ∧
      reg_base(min_linear)(type_of(a')) ≤ s'1 ∧
      s'1 < reg_size(max_linear)(type_of(a')) ∧
      rem(expt(2, min_page))(offset(s'1)) + length(s'2) ≤
      expt(2, min_page)
    (split(min_page, a', bl')))
{-3}  every(λ (t: [Address, list[Byte]]):
      (cons?(t'2) ∧
      a' ≤ t'1 ∧
      t'1 + length(t'2) ≤ a' + length(bl') ∧
      type_of(t'1) = type_of(a') ∧
      reg_base((#type_of := Mem_, offset := 0#))(type_of(a')) ≤ t'1 ∧
      t'1 <
      reg_size((#type_of := Mem_, offset := max_linear_offset#))(type_of(a')))
      ∧
      rem(expt(2, min_page))(offset(t'1)) + length(t'2) ≤
      expt(2, min_page)
    (split(min_page, a', bl')))
{-4}  Mem?(a' type_of)
{-5}  0 ≤ a' offset
{-6}  a' offset < max_linear_offset
{-7}  cons?[Byte](bl')
{-8}  is_linear_plain_memory?(pm')

```

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Hiding formulas: (-1 -2 2),

Expanding the definition of linear_write_side_effect_in_page,

Repeatedly Skolemizing and flattening,

Case splitting on pm!1'rw_addr(t!1'1),

we get 2 subgoals:

split_linear_write_side_effects_ok.1.1.1.1.1.1:

{-1}	pm'rw_addr(t'1)
{-2}	cons?(t'2)
{-3}	$a' \leq t'1$
{-4}	$t'1 + \text{length}(t'2) \leq a' + \text{length}(bl')$
{-5}	$\text{type_of}(t'1) = \text{type_of}(a')$
{-6}	$\text{reg_base}(\text{\#type_of} := \text{Mem_}, \text{offset} := 0\#)(\text{type_of}(a')) \leq t'1$
{-7}	$t'1 < \text{reg_size}(\text{\#type_of} := \text{Mem_}, \text{offset} := \text{max_linear_offset}\#)(\text{type_of}(a'))$
{-8}	$\text{rem}(\text{expt}(2, \text{min_page}))(\text{offset}(t'1)) + \text{length}(t'2) \leq \text{expt}(2, \text{min_page})$
{-9}	$\text{Mem?}(a' \text{ type_of})$
{-10}	$0 \leq a' \text{ offset}$
{-11}	$a' \text{ offset} < \text{max_linear_offset}$
{-12}	$\text{cons?}[\text{Byte}](bl')$
{-13}	$\text{is_linear_plain_memory?}(pm')$
{-14}	$(\text{address_block}(a', \text{length}(bl'))) \subseteq pm' \text{ rw_addr}$
{-15}	$pm' \text{ states}(s')$
{1}	$\text{OK?}((\text{linear_resolve}(t'1, \text{Write}) \text{ \#\#}$ $(\lambda (pa: \text{Memory_Address_4G})(ns: \text{Linear_memory}[\text{Physical_memory}, pm_phy]):$ $\text{memory_write_side_effect}(pm_phy' \text{ mem})(pa, t'2, \text{TRUE})(s'))$ $(s'))$

Using lemma pm_linear_resolve_write_ok,

Case splitting on $\text{subset?}(\text{address_block}(\text{data}(\text{linear_resolve}(t!1'1, \text{Write})(s!1)), \text{length}(t!1'2)), pm_phy'rw_addr)$,

we get 3 subgoals:

```
split_linear_write_side_effects_ok.1.1.1.1.1.1.1.1:
```

{-1}	$(\text{address_block}(\text{data}(\text{linear_resolve}(t'1, \text{Write})(s')), \text{length}(t'2)) \subseteq \text{pm_phy}'\text{rw_addr})$
{-2}	$\text{is_linear_plain_memory?}(\text{pm}') \supset \text{OK?}(\text{linear_resolve}(t'1, \text{Write})(s'))$
{-3}	$\text{pm}'\text{rw_addr}(t'1)$
{-4}	$\text{cons?}(t'2)$
{-5}	$a' \leq t'1$
{-6}	$t'1 + \text{length}(t'2) \leq a' + \text{length}(\text{bl}')$
{-7}	$\text{type_of}(t'1) = \text{type_of}(a')$
{-8}	$\text{reg_base}((\#\text{type_of} := \text{Mem_}, \text{offset} := 0\#))(\text{type_of}(a')) \leq t'1$
{-9}	$t'1 < \text{reg_size}((\#\text{type_of} := \text{Mem_}, \text{offset} := \text{max_linear_offset}\#))(\text{type_of}(a'))$
{-10}	$\text{rem}(\text{expt}(2, \text{min_page}))(\text{offset}(t'1)) + \text{length}(t'2) \leq \text{expt}(2, \text{min_page})$
{-11}	$\text{Mem?}(a'\text{type_of})$
{-12}	$0 \leq a'\text{offset}$
{-13}	$a'\text{offset} < \text{max_linear_offset}$
{-14}	$\text{cons?}[\text{Byte}](\text{bl}')$
{-15}	$\text{is_linear_plain_memory?}(\text{pm}')$
{-16}	$(\text{address_block}(a', \text{length}(\text{bl}')) \subseteq \text{pm}'\text{rw_addr})$
{-17}	$\text{pm}'\text{states}(s')$
{1}	$\text{OK?}((\text{linear_resolve}(t'1, \text{Write}) \#\#$ $(\lambda (\text{pa}: \text{Memory_Address_4G})(\text{ns}: \text{Linear_memory}[\text{Physical_memory}, \text{pm_phy}]):$ $\text{memory_write_side_effect}(\text{pm_phy}'\text{mem})(\text{pa}, t'2, \text{TRUE})(s'))$ $(s'))$

Using lemma `linear_resolve_states`,

Using lemma `pm_states`,

Using lemma `pm_plain_phy`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 2 subgoals:

```
split_linear_write_side_effects_ok.1.1.1.1.1.1.1.1.1:
```

{-1}	$\text{is_linear_plain_memory?}(\text{pm}')$
{-2}	$\text{plain_memory?}(\text{pm_phy})$
{-3}	$\text{pm}'\text{states} = \text{pm_phy}'\text{states}$
{-4}	$\text{pm}'\text{states}(s')$
{-5}	$\text{OK?}(\text{linear_resolve}(t'1, \text{Write})(s'))$
{-6}	$\text{pm_phy}'\text{states}(\text{state}(\text{linear_resolve}(t'1, \text{Write})(s')))$
{-7}	$(\text{address_block}(\text{data}(\text{linear_resolve}(t'1, \text{Write})(s')), \text{length}(t'2)) \subseteq \text{pm_phy}'\text{rw_addr})$
{-8}	$\text{pm}'\text{rw_addr}(t'1)$
{-9}	$\text{cons?}(t'2)$
{-10}	$a' \leq t'1$
{-11}	$t'1 + \text{length}(t'2) \leq a' + \text{length}(\text{bl}')$
{-12}	$\text{type_of}(t'1) = \text{type_of}(a')$
{-13}	$\text{reg_base}((\#\text{type_of} := \text{Mem_}, \text{offset} := 0\#))(\text{type_of}(a')) \leq t'1$
{-14}	$t'1 < \text{reg_size}((\#\text{type_of} := \text{Mem_}, \text{offset} := \text{max_linear_offset}\#))(\text{type_of}(a'))$
{-15}	$\text{rem}(\text{expt}(2, \text{min_page}))(\text{offset}(t'1)) + \text{length}(t'2) \leq \text{expt}(2, \text{min_page})$
{-16}	$\text{Mem?}(a'\text{type_of})$
{-17}	$0 \leq a'\text{offset}$
{-18}	$a'\text{offset} < \text{max_linear_offset}$
{-19}	$\text{cons?}[\text{Byte}](\text{bl}')$
{-20}	$(\text{address_block}(a', \text{length}(\text{bl}')) \subseteq \text{pm}'\text{rw_addr})$
{1}	$\text{OK?}(\text{memory_write_side_effect}(\text{pm_phy}'\text{mem}$ $(\text{data}(\text{linear_resolve}(t'1, \text{Write})(s')), t'2, \text{TRUE})(s'))$

Using lemma plain_memory_transformers_ok_write_side_effects_block,

Using lemma super_transformers_ok_ok,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Expanding the definition of memory_write_side_effect_super_transformers,

Instantiating quantified variables,

Rewriting using subset_reflexive, matching in *,

This completes the proof of split_linear_write_side_effects_ok.1.1.1.1.1.1.1.1.1.

split_linear_write_side_effects_ok.1.1.1.1.1.1.1.1.2:

{-1}	is_linear_plain_memory?(pm')
{-2}	plain_memory?(pm_phy)
{-3}	pm' states = pm_phy states
{-4}	pm' states(s')
{-5}	OK?(linear_resolve(t'1, Write)(s'))
{-6}	(address_block(data(linear_resolve(t'1, Write)(s')), length(t'2)) \subseteq pm_phy rw_addr)
{-7}	pm' rw_addr(t'1)
{-8}	cons?(t'2)
{-9}	$a' \leq t'1$
{-10}	$t'1 + \text{length}(t'2) \leq a' + \text{length}(bl')$
{-11}	type_of(t'1) = type_of(a')
{-12}	reg_base((#type_of := Mem_, offset := 0#))(type_of(a')) $\leq t'1$
{-13}	$t'1 < \text{reg_size}((\#type_of := \text{Mem_}, \text{offset} := \text{max_linear_offset\#})(\text{type_of}(a')))$
{-14}	$\text{rem}(\text{expt}(2, \text{min_page}))(\text{offset}(t'1) + \text{length}(t'2)) \leq \text{expt}(2, \text{min_page})$
{-15}	Mem?(a' type_of)
{-16}	$0 \leq a' \text{ offset}$
{-17}	$a' \text{ offset} < \text{max_linear_offset}$
{-18}	cons?[Byte](bl')
{-19}	(address_block(a', length(bl')) \subseteq pm' rw_addr)
<hr/>	
{1}	union(pm' ro_addr, pm' rw_addr)(t'1)
{2}	OK?(memory_write_side_effect(pm_phy mem) (data(linear_resolve(t'1, Write)(s')), t'2, TRUE)(s'))

Rewriting using union_right, matching in *,

This completes the proof of split_linear_write_side_effects_ok.1.1.1.1.1.1.1.1.2.

split_linear_write_side_effects_ok.1.1.1.1.1.1.2:

{-1}	is_linear_plain_memory?(pm') \supset OK?(linear_resolve(t'1, Write)(s'))
{-2}	pm'rw_addr(t'1)
{-3}	cons?(t'2)
{-4}	$a' \leq t'1$
{-5}	$t'1 + \text{length}(t'2) \leq a' + \text{length}(b')$
{-6}	type_of(t'1) = type_of(a')
{-7}	reg_base((#type_of := Mem_, offset := 0#))(type_of(a')) $\leq t'1$
{-8}	$t'1 < \text{reg_size}((\#type_of := \text{Mem_}, \text{offset} := \text{max_linear_offset\#}))(type_of(a'))$
{-9}	$\text{rem}(\text{expt}(2, \text{min_page}))(\text{offset}(t'1)) + \text{length}(t'2) \leq \text{expt}(2, \text{min_page})$
{-10}	Mem?(a'type_of)
{-11}	$0 \leq a'\text{offset}$
{-12}	$a'\text{offset} < \text{max_linear_offset}$
{-13}	cons?[Byte](b')
{-14}	is_linear_plain_memory?(pm')
{-15}	$(\text{address_block}(a', \text{length}(b'))) \subseteq \text{pm'rw_addr}$
{-16}	pm'states(s')
{1}	$(\text{address_block}(\text{data}(\text{linear_resolve}(t'1, \text{Write})(s')), \text{length}(t'2)) \subseteq \text{pm_phy'rw_addr})$
{2}	OK?((linear_resolve(t'1, Write) ## $(\lambda (\text{pa}: \text{Memory_Address_4G})(\text{ns}: \text{Linear_memory}[\text{Physical_memory}, \text{pm_phy}]):$ $\text{memory_write_side_effect}(\text{pm_phy' mem})(\text{pa}, t'2, \text{TRUE})(s'))$ $(s'))$

Hiding formulas: 2,

Using lemma same_page_address_block_write,

we get 3 subgoals:

split_linear_write_side_effects_ok.1.1.1.1.1.1.2.1:

{-1}	is_linear_plain_memory?(pm') \wedge $pm' \text{ states}(s') \wedge$ $pm' \text{ rw_addr}(t'1) \wedge$ $a' \leq t'1 \wedge$ $t'1 + \text{length}(t'2) \leq a' + \text{length}(bl') \wedge$ $\text{rem}(\text{expt}(2, \text{min_page}))(\text{offset}(t'1)) + \text{length}(t'2) \leq \text{expt}(2, \text{min_page})$ $\wedge (\text{address_block}(a', \text{length}(bl')) \subseteq pm' \text{ rw_addr})$ \supset $(\text{address_block}(\text{data}(\text{linear_resolve}(t'1, \text{Write})(s')), \text{length}(t'2)) \subseteq pm_phy \text{ rw_addr})$
{-2}	is_linear_plain_memory?(pm') \supset OK?(linear_resolve(t'1, Write)(s'))
{-3}	pm' rw_addr(t'1)
{-4}	cons?(t'2)
{-5}	$a' \leq t'1$
{-6}	$t'1 + \text{length}(t'2) \leq a' + \text{length}(bl')$
{-7}	$\text{type_of}(t'1) = \text{type_of}(a')$
{-8}	$\text{reg_base}(\text{(#type_of := Mem_}, \text{offset := 0\#}))(\text{type_of}(a')) \leq t'1$
{-9}	$t'1 < \text{reg_size}(\text{(#type_of := Mem_}, \text{offset := max_linear_offset\#}))(\text{type_of}(a'))$
{-10}	$\text{rem}(\text{expt}(2, \text{min_page}))(\text{offset}(t'1)) + \text{length}(t'2) \leq \text{expt}(2, \text{min_page})$
{-11}	Mem?(a' type_of)
{-12}	$0 \leq a' \text{ offset}$
{-13}	$a' \text{ offset} < \text{max_linear_offset}$
{-14}	cons?[Byte](bl')
{-15}	is_linear_plain_memory?(pm')
{-16}	$(\text{address_block}(a', \text{length}(bl')) \subseteq pm' \text{ rw_addr})$
{-17}	pm' states(s')
{1}	$(\text{address_block}(\text{data}(\text{linear_resolve}(t'1, \text{Write})(s')), \text{length}(t'2)) \subseteq pm_phy \text{ rw_addr})$

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of split_linear_write_side_effects_ok.1.1.1.1.1.1.2.1.

split_linear_write_side_effects_ok.1.1.1.1.1.1.2.2:

{-1}	is_linear_plain_memory?(pm') \supset OK?(linear_resolve(t'1, Write)(s'))
{-2}	pm' rw_addr(t'1)
{-3}	cons?(t'2)
{-4}	$a' \leq t'1$
{-5}	$t'1 + \text{length}(t'2) \leq a' + \text{length}(bl')$
{-6}	$\text{type_of}(t'1) = \text{type_of}(a')$
{-7}	$\text{reg_base}(\text{(#type_of := Mem_}, \text{offset := 0\#}))(\text{type_of}(a')) \leq t'1$
{-8}	$t'1 < \text{reg_size}(\text{(#type_of := Mem_}, \text{offset := max_linear_offset\#}))(\text{type_of}(a'))$
{-9}	$\text{rem}(\text{expt}(2, \text{min_page}))(\text{offset}(t'1)) + \text{length}(t'2) \leq \text{expt}(2, \text{min_page})$
{-10}	Mem?(a' type_of)
{-11}	$0 \leq a' \text{ offset}$
{-12}	$a' \text{ offset} < \text{max_linear_offset}$
{-13}	cons?[Byte](bl')
{-14}	is_linear_plain_memory?(pm')
{-15}	$(\text{address_block}(a', \text{length}(bl')) \subseteq pm' \text{ rw_addr})$
{-16}	pm' states(s')
{1}	$\text{length}[Byte](t'2) > 0$
{2}	$(\text{address_block}(\text{data}(\text{linear_resolve}(t'1, \text{Write})(s')), \text{length}(t'2)) \subseteq pm_phy \text{ rw_addr})$

Expanding the definition of length,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

C Proof scripts

This completes the proof of `split_linear_write_side_effects_ok.1.1.1.1.1.1.2.2.`

`split_linear_write_side_effects_ok.1.1.1.1.1.1.2.3:`

{-1}	<code>is_linear_plain_memory?(pm') \supset OK?(linear_resolve(t'1, Write)(s'))</code>
{-2}	<code>pm'rw_addr(t'1)</code>
{-3}	<code>cons?(t'2)</code>
{-4}	<code>a' \leq t'1</code>
{-5}	<code>t'1 + length(t'2) \leq a' + length(bl')</code>
{-6}	<code>type_of(t'1) = type_of(a')</code>
{-7}	<code>reg_base((#type_of := Mem_, offset := 0#))(type_of(a')) \leq t'1</code>
{-8}	<code>t'1 < reg_size((#type_of := Mem_, offset := max_linear_offset#))(type_of(a'))</code>
{-9}	<code>rem(expt(2, min_page))(offset(t'1)) + length(t'2) \leq expt(2, min_page)</code>
{-10}	<code>Mem?(a'type_of)</code>
{-11}	<code>0 \leq a'offset</code>
{-12}	<code>a'offset < max_linear_offset</code>
{-13}	<code>cons?[Byte](bl')</code>
{-14}	<code>is_linear_plain_memory?(pm')</code>
{-15}	<code>(address_block(a', length(bl'))) \subseteq pm'rw_addr</code>
{-16}	<code>pm'states(s')</code>
{1}	<code>length[Byte](bl') > 0</code>
{2}	<code>(address_block(data(linear_resolve(t'1, Write)(s')), length(t'2))) \subseteq pm_phyrw_addr</code>

Expanding the definition of length,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `split_linear_write_side_effects_ok.1.1.1.1.1.1.2.3.`

`split_linear_write_side_effects_ok.1.1.1.1.1.1.3:`

{-1}	<code>is_linear_plain_memory?(pm') \supset OK?(linear_resolve(t'1, Write)(s'))</code>
{-2}	<code>pm'rw_addr(t'1)</code>
{-3}	<code>cons?(t'2)</code>
{-4}	<code>a' \leq t'1</code>
{-5}	<code>t'1 + length(t'2) \leq a' + length(bl')</code>
{-6}	<code>type_of(t'1) = type_of(a')</code>
{-7}	<code>reg_base((#type_of := Mem_, offset := 0#))(type_of(a')) \leq t'1</code>
{-8}	<code>t'1 < reg_size((#type_of := Mem_, offset := max_linear_offset#))(type_of(a'))</code>
{-9}	<code>rem(expt(2, min_page))(offset(t'1)) + length(t'2) \leq expt(2, min_page)</code>
{-10}	<code>Mem?(a'type_of)</code>
{-11}	<code>0 \leq a'offset</code>
{-12}	<code>a'offset < max_linear_offset</code>
{-13}	<code>cons?[Byte](bl')</code>
{-14}	<code>is_linear_plain_memory?(pm')</code>
{-15}	<code>(address_block(a', length(bl'))) \subseteq pm'rw_addr</code>
{-16}	<code>pm'states(s')</code>
{1}	<code>OK?[Physical_memory, Address](linear_resolve[Physical_memory, pm_phy](t'1, Write)(s'))</code>
{2}	<code>OK?((linear_resolve(t'1, Write) ## $(\lambda$ (pa: Memory_Address_4G)(ns: Linear_memory[Physical_memory, pm_phy]): memory_write_side_effect(pm_phy'mem)(pa, t'2, TRUE)(s')) (s'))</code>

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `split_linear_write_side_effects_ok.1.1.1.1.1.1.3.`

split_linear_write_side_effects_ok.1.1.1.1.1.2:

{-1}	cons?(t'2)
{-2}	a' ≤ t'1
{-3}	t'1 + length(t'2) ≤ a' + length(bl')
{-4}	type_of(t'1) = type_of(a')
{-5}	reg_base((#type_of := Mem_, offset := 0#))(type_of(a')) ≤ t'1
{-6}	t'1 < reg_size((#type_of := Mem_, offset := max_linear_offset#))(type_of(a'))
{-7}	rem(expt(2, min_page))(offset(t'1)) + length(t'2) ≤ expt(2, min_page)
{-8}	Mem?(a' type_of)
{-9}	0 ≤ a' offset
{-10}	a' offset < max_linear_offset
{-11}	cons?[Byte](bl')
{-12}	is_linear_plain_memory?(pm')
{-13}	(address_block(a', length(bl'))) ⊆ pm'rw_addr
{-14}	pm' states(s')
{1}	pm'rw_addr(t'1)
{2}	OK?((linear_resolve(t'1, Write) ## (λ (pa: Memory_Address_4G)(ns: Linear_memory[Physical_memory, pm_phy]): memory_write_side_effect(pm_phy mem)(pa, t'2, TRUE)(s')) (s'))

Hiding formulas: (-7 -12 2),

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of split_linear_write_side_effects_ok.1.1.1.1.1.2.

split_linear_write_side_effects_ok.1.1.1.1.2:

```

{-1} every(λ (t_1: [Address, list[Byte]]):
      cons?(t_1'2) ∧
      a' ≤ t_1'1 ∧
      t_1'1 + length(t_1'2) ≤ a' + length(bl') ∧
      type_of(t_1'1) = type_of(a') ∧
      reg_base(min_linear)(type_of(a')) ≤ t_1'1 ∧
      t_1'1 < reg_size(max_linear)(type_of(a')) ∧
      rem(expt(2, min_page))(offset(t_1'1)) + length(t_1'2) ≤
      expt(2, min_page))
      (split(min_page, a', bl'))
  ⊃
  every(λ (s: [Memory_Address_4G, list[Byte]]):
      cons?(s'2) ∧
      a' ≤ s'1 ∧
      s'1 + length(s'2) ≤ a' + length(bl') ∧
      type_of(s'1) = type_of(a') ∧
      reg_base(min_linear)(type_of(a')) ≤ s'1 ∧
      s'1 < reg_size(max_linear)(type_of(a')) ∧
      rem(expt(2, min_page))(offset(s'1)) + length(s'2) ≤
      expt(2, min_page))
      (split(min_page, a', bl'))
{-2} every(λ (t: [Address, list[Byte]]):
      (cons?(t'2) ∧
      a' ≤ t'1 ∧
      t'1 + length(t'2) ≤ a' + length(bl') ∧
      type_of(t'1) = type_of(a') ∧
      reg_base((#type_of := Mem_, offset := 0#))(type_of(a')) ≤ t'1 ∧
      t'1 <
      reg_size((#type_of := Mem_, offset := max_linear_offset#))(type_of(a')))
      ∧
      rem(expt(2, min_page))(offset(t'1)) + length(t'2) ≤
      expt(2, min_page))
      (split(min_page, a', bl'))
{-3} Mem?(a' type_of)
{-4} 0 ≤ a' offset
{-5} a' offset < max_linear_offset
{-6} cons?[Byte](bl')
{-7} is_linear_plain_memory?(pm')
{-8} (address_block(a', length(bl')) ⊆ pm'rw_addr)
{-9} pm' states(s')


---


{1} ∀ (s: [Memory_Address_4G, list[Byte]]):
      reg_base(Mem(0))(type_of(a')) ≤ s'1 ∧
      type_of(s'1) = type_of(a') ∧
      s'1 + length(s'2) ≤ a' + length(bl') ∧ a' ≤ s'1 ∧ cons?(s'2)
      ⊃ Mem?(Mem(max_linear_offset) type_of)
{2} every(λ (e: [Memory_Address_4G, list[Byte]]):
      OK?(linear_write_side_effect_in_page(e)(s')))
      (split(min_page, a', bl'))

```

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `split_linear_write_side_effects_ok.1.1.1.1.2`.

split_linear_write_side_effects_ok.1.1.1.1.3:

{-1}	$\begin{aligned} & \text{every}(\lambda (t_1: [\text{Address}, \text{list}[\text{Byte}]]): \\ & \quad \text{cons?}(t_1'2) \wedge \\ & \quad a' \leq t_1'1 \wedge \\ & \quad t_1'1 + \text{length}(t_1'2) \leq a' + \text{length}(bl') \wedge \\ & \quad \text{type_of}(t_1'1) = \text{type_of}(a') \wedge \\ & \quad \text{reg_base}(\text{min_linear})(\text{type_of}(a')) \leq t_1'1 \wedge \\ & \quad t_1'1 < \text{reg_size}(\text{max_linear})(\text{type_of}(a')) \wedge \\ & \quad \text{rem}(\text{expt}(2, \text{min_page}))(\text{offset}(t_1'1)) + \text{length}(t_1'2) \leq \\ & \quad \text{expt}(2, \text{min_page})) \\ & \quad (\text{split}(\text{min_page}, a', bl')) \end{aligned}$
	\supset
	$\begin{aligned} & \text{every}(\lambda (s: [\text{Memory_Address_4G}, \text{list}[\text{Byte}]]): \\ & \quad \text{cons?}(s'2) \wedge \\ & \quad a' \leq s'1 \wedge \\ & \quad s'1 + \text{length}(s'2) \leq a' + \text{length}(bl') \wedge \\ & \quad \text{type_of}(s'1) = \text{type_of}(a') \wedge \\ & \quad \text{reg_base}(\text{min_linear})(\text{type_of}(a')) \leq s'1 \wedge \\ & \quad s'1 < \text{reg_size}(\text{max_linear})(\text{type_of}(a')) \wedge \\ & \quad \text{rem}(\text{expt}(2, \text{min_page}))(\text{offset}(s'1)) + \text{length}(s'2) \leq \\ & \quad \text{expt}(2, \text{min_page})) \\ & \quad (\text{split}(\text{min_page}, a', bl')) \end{aligned}$
{-2}	$\begin{aligned} & \text{every}(\lambda (t: [\text{Address}, \text{list}[\text{Byte}]]): \\ & \quad (\text{cons?}(t'2) \wedge \\ & \quad a' \leq t'1 \wedge \\ & \quad t'1 + \text{length}(t'2) \leq a' + \text{length}(bl') \wedge \\ & \quad \text{type_of}(t'1) = \text{type_of}(a') \wedge \\ & \quad \text{reg_base}((\# \text{type_of} := \text{Mem_}, \text{offset} := 0\#))(\text{type_of}(a')) \leq t'1 \wedge \\ & \quad t'1 < \\ & \quad \text{reg_size}((\# \text{type_of} := \text{Mem_}, \text{offset} := \text{max_linear_offset}\#))(\text{type_of}(a'))) \\ & \quad \wedge \\ & \quad \text{rem}(\text{expt}(2, \text{min_page}))(\text{offset}(t'1)) + \text{length}(t'2) \leq \\ & \quad \text{expt}(2, \text{min_page})) \\ & \quad (\text{split}(\text{min_page}, a', bl')) \end{aligned}$
{-3}	$\text{Mem?}(a' \text{ type_of})$
{-4}	$0 \leq a' \text{ offset}$
{-5}	$a' \text{ offset} < \text{max_linear_offset}$
{-6}	$\text{cons?}[\text{Byte}](bl')$
{-7}	$\text{is_linear_plain_memory?}(pm')$
{-8}	$(\text{address_block}(a', \text{length}(bl'))) \subseteq pm' \text{ rw_addr}$
{-9}	$pm' \text{ states}(s')$
{1}	$\begin{aligned} & \forall (s: [\text{Memory_Address_4G}, \text{list}[\text{Byte}]]): \\ & \quad \text{type_of}(s'1) = \text{type_of}(a') \wedge \\ & \quad s'1 + \text{length}(s'2) \leq a' + \text{length}(bl') \wedge a' \leq s'1 \wedge \text{cons?}(s'2) \\ & \quad \supset \text{Mem?}(\text{Mem}(0) \text{ type_of}) \end{aligned}$
{2}	$\begin{aligned} & \text{every}(\lambda (e: [\text{Memory_Address_4G}, \text{list}[\text{Byte}]]): \\ & \quad \text{OK?}(\text{linear_write_side_effect_in_page}(e)(s')) \\ & \quad (\text{split}(\text{min_page}, a', bl')) \end{aligned}$

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of split_linear_write_side_effects_ok.1.1.1.1.3.

split_linear_write_side_effects_ok.1.1.1.2:

<pre> {-1} every(λ (t: [Address, list[Byte]]): (cons?(t'2) ∧ a' ≤ t'1 ∧ t'1 + length(t'2) ≤ a' + length(bl') ∧ type_of(t'1) = type_of(a') ∧ reg_base((#type_of := Mem_, offset := 0#))(type_of(a')) ≤ t'1 ∧ t'1 < reg_size((#type_of := Mem_, offset := max_linear_offset#))(type_of(a'))) ∧ rem(expt(2, min_page))(offset(t'1) + length(t'2)) ≤ expt(2, min_page) (split(min_page, a', bl')) {-2} Mem?(a' 'type_of) {-3} 0 ≤ a' 'offset {-4} a' 'offset < max_linear_offset {-5} cons?[Byte](bl') {-6} is_linear_plain_memory?(pm') {-7} (address_block(a', length(bl')) ⊆ pm' 'rw_addr) {-8} pm' 'states(s') </pre>	<pre> {1} ∀ (t: [Address, list[Byte]]): reg_base(Mem(0))(type_of(a')) ≤ t'1 ∧ type_of(t'1) = type_of(a') ∧ t'1 + length(t'2) ≤ a' + length(bl') ∧ a' ≤ t'1 ∧ cons?(t'2) ⊃ Mem?(Mem(max_linear_offset) 'type_of) {2} every(λ (e: [Memory_Address_4G, list[Byte]]): OK?(linear_write_side_effect_in_page(e)(s'))) (split(min_page, a', bl')) </pre>
---	---

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `split_linear_write_side_effects_ok.1.1.1.2`.

split_linear_write_side_effects_ok.1.1.1.3:

{-1}	$\text{every}(\lambda (t: [\text{Address}, \text{list}[\text{Byte}]]):$ $\begin{aligned} & (\text{cons?}(t'2) \wedge \\ & \quad a' \leq t'1 \wedge \\ & \quad t'1 + \text{length}(t'2) \leq a' + \text{length}(bl') \wedge \\ & \quad \text{type_of}(t'1) = \text{type_of}(a') \wedge \\ & \quad \text{reg_base}((\#\text{type_of} := \text{Mem_}, \text{offset} := 0\#))(\text{type_of}(a')) \leq t'1 \wedge \\ & \quad t'1 < \\ & \quad \text{reg_size}((\#\text{type_of} := \text{Mem_}, \text{offset} := \text{max_linear_offset}\#))(\text{type_of}(a'))) \\ & \wedge \\ & \text{rem}(\text{expt}(2, \text{min_page}))(\text{offset}(t'1)) + \text{length}(t'2) \leq \\ & \quad \text{expt}(2, \text{min_page}) \\ & \quad (\text{split}(\text{min_page}, a', bl')) \end{aligned}$
{-2}	$\text{Mem?}(a' \text{'type_of})$
{-3}	$0 \leq a' \text{'offset}$
{-4}	$a' \text{'offset} < \text{max_linear_offset}$
{-5}	$\text{cons?}[\text{Byte}](bl')$
{-6}	$\text{is_linear_plain_memory?}(pm')$
{-7}	$(\text{address_block}(a', \text{length}(bl'))) \subseteq pm' \text{'rw_addr}$
{-8}	$pm' \text{'states}(s')$
{1}	$\forall (t: [\text{Address}, \text{list}[\text{Byte}]]):$ $\begin{aligned} & \text{type_of}(t'1) = \text{type_of}(a') \wedge \\ & \quad t'1 + \text{length}(t'2) \leq a' + \text{length}(bl') \wedge a' \leq t'1 \wedge \text{cons?}(t'2) \\ & \quad \supset \text{Mem?}(\text{Mem}(0) \text{'type_of}) \end{aligned}$
{2}	$\text{every}(\lambda (e: [\text{Memory_Address_4G}, \text{list}[\text{Byte}]]):$ $\begin{aligned} & \text{OK?}(\text{linear_write_side_effect_in_page}(e)(s')) \\ & \quad (\text{split}(\text{min_page}, a', bl')) \end{aligned}$

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of split_linear_write_side_effects_ok.1.1.1.3.

split_linear_write_side_effects_ok.1.1.2:

```

{-1} every(λ (t: [Address, list[Byte]]):
      cons?(t'2) ∧
      a' ≤ t'1 ∧ t'1 + length(t'2) ≤ a' + length(bl')
      (split(min_page, a', bl'))
    ∧
    every(λ (t: [Address, list[Byte]]):
      type_of(t'1) = type_of(a') ∧
      reg_base(min_linear)(type_of(a')) ≤ t'1 ∧
      t'1 < reg_size(max_linear)(type_of(a'))
      (split(min_page, a', bl'))
    ⊃
    every(λ (t_1: [Address, list[Byte]]):
      (cons?(t_1'2) ∧
      a' ≤ t_1'1 ∧ t_1'1 + length(t_1'2) ≤ a' + length(bl'))
      ∧
      type_of(t_1'1) = type_of(a') ∧
      reg_base(min_linear)(type_of(a')) ≤ t_1'1 ∧
      t_1'1 < reg_size(max_linear)(type_of(a'))
      (split(min_page, a', bl'))
    {-2} every(λ (a_2: Address, bl2: list[Byte]): cons?(bl2))(split(min_page, a', bl')) ∧
    every(λ (a_2: Address, bl2: list[Byte]):
      a' ≤ a_2 ∧ a_2 + length(bl2) ≤ a' + length(bl')
      (split(min_page, a', bl'))
    ⊃
    every(λ (t: [Address, list[Byte]]):
      cons?(t'2) ∧
      a' ≤ t'1 ∧ t'1 + length(t'2) ≤ a' + length(bl')
      (split(min_page, a', bl'))
    {-3} every(λ (a_2: Address, bl2: list[Byte]):
      rem(expt(2, min_page))(offset(a_2)) + length(bl2) ≤ expt(2, min_page))
      (split(min_page, a', bl'))
    {-4} every(λ (t: [Address, list[Byte]]):
      type_of(t'1) = type_of(a') ∧
      reg_base((#type_of := Mem_, offset := 0#))(type_of(a')) ≤ t'1 ∧
      t'1 < reg_size((#type_of := Mem_, offset := max_linear_offset#))(type_of(a'))
      (split(min_page, a', bl'))
    {-5} every(λ (a_2: Address, bl2: list[Byte]): cons?(bl2))(split(min_page, a', bl'))
    {-6} every(λ (a_2: Address, bl2: list[Byte]):
      a' ≤ a_2 ∧ a_2 + length(bl2) ≤ a' + length(bl')
      (split(min_page, a', bl'))
    {-7} Mem?(a' type_of)
    {-8} 0 ≤ a' offset
    {-9} a' offset < max_linear_offset
    {-10} every(λ (x: number):
      number_field_pred(x) ∧
      real_pred(x) ∧
      rational_pred(x) ∧ integer_pred(x) ∧ x ≥ 0 ∧ x < max_byte)
      (bl')
    {-11} cons?[Byte](bl')
    {-12} is_linear_plain_memory?(pm')
    {-13} (address_block(a', length(bl')) ⊆ pm'rw_addr)
    {-14} pm' states(s')
  }
  {1} ∀ (t_1: [Address, list[Byte]]):
    reg_base(Mem(0))(type_of(a')) ≤ t_1'1 ∧
    type_of(t_1'1) = type_of(a') ∧
    t_1'1 + length(t_1'2) ≤ a' + length(bl') ∧
    a' ≤ t_1'1 ∧ cons?(t_1'2)
    ⊃ Mem?(Mem(max_linear_offset) type_of)
  {2} every(λ (e: [Memory_Address_4G, list[Byte]]):
    OK?(linear_write_side_effect_in_page(e)(s'))
    (split(min_page, a', bl'))

```

C.117 Proofs for Linear_Memory_Properties (challenge-linear.pvs)

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `split_linear_write_side_effects_ok.1.1.2`.

split_linear_write_side_effects_ok.1.1.3:

```

{-1} every(λ (t: [Address, list[Byte]]):
      cons?(t'2) ∧
      a' ≤ t'1 ∧ t'1 + length(t'2) ≤ a' + length(bl')
      (split(min_page, a', bl'))
      ∧
      every(λ (t: [Address, list[Byte]]):
            type_of(t'1) = type_of(a') ∧
            reg_base(min_linear)(type_of(a')) ≤ t'1 ∧
            t'1 < reg_size(max_linear)(type_of(a'))
            (split(min_page, a', bl'))
            ⊃
            every(λ (t_1: [Address, list[Byte]]):
                  (cons?(t_1'2) ∧
                   a' ≤ t_1'1 ∧ t_1'1 + length(t_1'2) ≤ a' + length(bl'))
                  ∧
                  type_of(t_1'1) = type_of(a') ∧
                  reg_base(min_linear)(type_of(a')) ≤ t_1'1 ∧
                  t_1'1 < reg_size(max_linear)(type_of(a'))
                  (split(min_page, a', bl'))
            )
      )
      (split(min_page, a', bl'))
      ∧
      every(λ (a_2: Address, bl2: list[Byte]): cons?(bl2))(split(min_page, a', bl')) ∧
      every(λ (a_2: Address, bl2: list[Byte]):
            a' ≤ a_2 ∧ a_2 + length(bl2) ≤ a' + length(bl')
            (split(min_page, a', bl'))
            ⊃
            every(λ (t: [Address, list[Byte]]):
                  cons?(t'2) ∧
                  a' ≤ t'1 ∧ t'1 + length(t'2) ≤ a' + length(bl')
                  (split(min_page, a', bl'))
            )
      )
      (split(min_page, a', bl'))
      ∧
      every(λ (a_2: Address, bl2: list[Byte]):
            rem(expt(2, min_page))(offset(a_2)) + length(bl2) ≤ expt(2, min_page)
            (split(min_page, a', bl'))
      )
      (split(min_page, a', bl'))
      ∧
      every(λ (t: [Address, list[Byte]]):
            type_of(t'1) = type_of(a') ∧
            reg_base((#type_of := Mem_, offset := 0#))(type_of(a')) ≤ t'1 ∧
            t'1 < reg_size((#type_of := Mem_, offset := max_linear_offset#))(type_of(a'))
            (split(min_page, a', bl'))
      )
      (split(min_page, a', bl'))
      ∧
      every(λ (a_2: Address, bl2: list[Byte]): cons?(bl2))(split(min_page, a', bl'))
      (split(min_page, a', bl'))
      ∧
      every(λ (a_2: Address, bl2: list[Byte]):
            a' ≤ a_2 ∧ a_2 + length(bl2) ≤ a' + length(bl')
            (split(min_page, a', bl'))
      )
      (split(min_page, a', bl'))
      ∧
      Mem?(a' type_of)
      0 ≤ a' offset
      a' offset < max_linear_offset
      every(λ (x: number):
            number_field_pred(x) ∧
            real_pred(x) ∧
            rational_pred(x) ∧ integer_pred(x) ∧ x ≥ 0 ∧ x < max_byte)
      (bl')
      cons?[Byte](bl')
      is_linear_plain_memory?(pm')
      (address_block(a', length(bl')) ⊆ pm'rw_addr)
      pm' states(s')
      {1} ∀ (t_1: [Address, list[Byte]]):
            type_of(t_1'1) = type_of(a') ∧
            t_1'1 + length(t_1'2) ≤ a' + length(bl') ∧
            a' ≤ t_1'1 ∧ cons?(t_1'2)
            ⊃ Mem?(Mem(0) type_of)
      {2} every(λ (e: [Memory_Address_4G, list[Byte]]):
            OK?(linear_write_side_effect_in_page(e)(s'))
            (split(min_page, a', bl'))

```

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `split_linear_write_side_effects_ok.1.1.3`.

`split_linear_write_side_effects_ok.1.2`:

{-1}	$\text{every}(\lambda (a_2: \text{Address}, \text{bl2}: \text{list}[\text{Byte}]): \text{cons}?(\text{bl2}))(\text{split}(\text{min_page}, a', \text{bl}')) \wedge$ $\text{every}(\lambda (a_2: \text{Address}, \text{bl2}: \text{list}[\text{Byte}]):$ $a' \leq a_2 \wedge a_2 + \text{length}(\text{bl2}) \leq a' + \text{length}(\text{bl}'))$ $(\text{split}(\text{min_page}, a', \text{bl}'))$ \supset $\text{every}(\lambda (t: [\text{Address}, \text{list}[\text{Byte}]]):$ $\text{cons}?(t'2) \wedge$ $a' \leq t'1 \wedge t'1 + \text{length}(t'2) \leq a' + \text{length}(\text{bl}'))$ $(\text{split}(\text{min_page}, a', \text{bl}'))$
{-2}	$\text{every}(\lambda (a_2: \text{Address}, \text{bl2}: \text{list}[\text{Byte}]):$ $\text{rem}(\text{expt}(2, \text{min_page}))(\text{offset}(a_2)) + \text{length}(\text{bl2}) \leq \text{expt}(2, \text{min_page}))$ $(\text{split}(\text{min_page}, a', \text{bl}'))$
{-3}	$\text{every}(\lambda (t: [\text{Address}, \text{list}[\text{Byte}]]):$ $\text{type_of}(t'1) = \text{type_of}(a') \wedge$ $\text{reg_base}(\text{\#type_of} := \text{Mem_}, \text{offset} := 0\#)(\text{type_of}(a')) \leq t'1 \wedge$ $t'1 < \text{reg_size}(\text{\#type_of} := \text{Mem_}, \text{offset} := \text{max_linear_offset\#})(\text{type_of}(a')))$ $(\text{split}(\text{min_page}, a', \text{bl}'))$
{-4}	$\text{every}(\lambda (a_2: \text{Address}, \text{bl2}: \text{list}[\text{Byte}]): \text{cons}?(\text{bl2}))(\text{split}(\text{min_page}, a', \text{bl}'))$
{-5}	$\text{every}(\lambda (a_2: \text{Address}, \text{bl2}: \text{list}[\text{Byte}]):$ $a' \leq a_2 \wedge a_2 + \text{length}(\text{bl2}) \leq a' + \text{length}(\text{bl}'))$ $(\text{split}(\text{min_page}, a', \text{bl}'))$
{-6}	$\text{Mem}?(a' \text{ type_of})$
{-7}	$0 \leq a' \text{ offset}$
{-8}	$a' \text{ offset} < \text{max_linear_offset}$
{-9}	$\text{every}(\lambda (x: \text{number}):$ $\text{number_field_pred}(x) \wedge$ $\text{real_pred}(x) \wedge$ $\text{rational_pred}(x) \wedge \text{integer_pred}(x) \wedge x \geq 0 \wedge x < \text{max_byte})$ (bl')
{-10}	$\text{cons}?[\text{Byte}](\text{bl}')$
{-11}	$\text{is_linear_plain_memory}?(\text{pm}')$
{-12}	$(\text{address_block}(a', \text{length}(\text{bl}')) \subseteq \text{pm}' \text{rw_addr})$
{-13}	$\text{pm}' \text{states}(s')$
{1}	$\forall (t: [\text{Address}, \text{list}[\text{Byte}]]):$ $\text{reg_base}(\text{Mem}(0))(\text{type_of}(a')) \leq t'1 \wedge \text{type_of}(t'1) = \text{type_of}(a') \supset$ $\text{Mem}?(\text{Mem}(\text{max_linear_offset}) \text{ type_of})$
{2}	$\text{every}(\lambda (e: [\text{Memory_Address_4G}, \text{list}[\text{Byte}]]):$ $\text{OK}?(\text{linear_write_side_effect_in_page}(e)(s'))$ $(\text{split}(\text{min_page}, a', \text{bl}'))$

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `split_linear_write_side_effects_ok.1.2`.

split_linear_write_side_effects_ok.1.3:

{-1}	$\text{every}(\lambda (a_2: \text{Address}, \text{bl2}: \text{list}[\text{Byte}]): \text{cons}?(\text{bl2}))(\text{split}(\text{min_page}, a', \text{bl}')) \wedge$ $\text{every}(\lambda (a_2: \text{Address}, \text{bl2}: \text{list}[\text{Byte}]):$ $a' \leq a_2 \wedge a_2 + \text{length}(\text{bl2}) \leq a' + \text{length}(\text{bl}'))$ $(\text{split}(\text{min_page}, a', \text{bl}'))$ \supset $\text{every}(\lambda (t: [\text{Address}, \text{list}[\text{Byte}]]):$ $\text{cons}?(t'2) \wedge$ $a' \leq t'1 \wedge t'1 + \text{length}(t'2) \leq a' + \text{length}(\text{bl}'))$ $(\text{split}(\text{min_page}, a', \text{bl}'))$
{-2}	$\text{every}(\lambda (a_2: \text{Address}, \text{bl2}: \text{list}[\text{Byte}]):$ $\text{rem}(\text{expt}(2, \text{min_page}))(\text{offset}(a_2)) + \text{length}(\text{bl2}) \leq \text{expt}(2, \text{min_page}))$ $(\text{split}(\text{min_page}, a', \text{bl}'))$
{-3}	$\text{every}(\lambda (t: [\text{Address}, \text{list}[\text{Byte}]]):$ $\text{type_of}(t'1) = \text{type_of}(a') \wedge$ $\text{reg_base}((\# \text{type_of} := \text{Mem_}, \text{offset} := 0\#))(\text{type_of}(a')) \leq t'1 \wedge$ $t'1 < \text{reg_size}((\# \text{type_of} := \text{Mem_}, \text{offset} := \text{max_linear_offset}\#))(\text{type_of}(a')))$ $(\text{split}(\text{min_page}, a', \text{bl}'))$
{-4}	$\text{every}(\lambda (a_2: \text{Address}, \text{bl2}: \text{list}[\text{Byte}]): \text{cons}?(\text{bl2}))(\text{split}(\text{min_page}, a', \text{bl}'))$
{-5}	$\text{every}(\lambda (a_2: \text{Address}, \text{bl2}: \text{list}[\text{Byte}]):$ $a' \leq a_2 \wedge a_2 + \text{length}(\text{bl2}) \leq a' + \text{length}(\text{bl}'))$ $(\text{split}(\text{min_page}, a', \text{bl}'))$
{-6}	$\text{Mem}?(a' \text{ 'type_of})$
{-7}	$0 \leq a' \text{ 'offset}$
{-8}	$a' \text{ 'offset} < \text{max_linear_offset}$
{-9}	$\text{every}(\lambda (x: \text{number}):$ $\text{number_field_pred}(x) \wedge$ $\text{real_pred}(x) \wedge$ $\text{rational_pred}(x) \wedge \text{integer_pred}(x) \wedge x \geq 0 \wedge x < \text{max_byte})$ (bl')
{-10}	$\text{cons}?[\text{Byte}](\text{bl}')$
{-11}	$\text{is_linear_plain_memory}?(\text{pm}')$
{-12}	$(\text{address_block}(a', \text{length}(\text{bl}')) \subseteq \text{pm}' \text{ 'rw_addr})$
{-13}	$\text{pm}' \text{ 'states}(s')$
{1}	$\forall (t: [\text{Address}, \text{list}[\text{Byte}]]):$ $\text{type_of}(t'1) = \text{type_of}(a') \supset \text{Mem}?(\text{Mem}(0) \text{ 'type_of})$
{2}	$\text{every}(\lambda (e: [\text{Memory_Address_4G}, \text{list}[\text{Byte}]]):$ $\text{OK}?(\text{linear_write_side_effect_in_page}(e)(s'))$ $(\text{split}(\text{min_page}, a', \text{bl}'))$

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `split_linear_write_side_effects_ok.1.3`.

split_linear_write_side_effects_ok.2:

<pre> {-1} every(λ (a2: Address, bl2: list[Byte]): rem(expt(2, min_page))(offset(a2)) + length(bl2) ≤ expt(2, min_page)) (split(min_page, a', bl')) {-2} every(λ (a2: Address, bl2: list[Byte]): cons?(bl2))(split(min_page, a', bl')) {-3} every(λ (a2: Address, bl2: list[Byte]): a' ≤ a2 ∧ a2 + length(bl2) ≤ a' + length(bl')) (split(min_page, a', bl')) {-4} Mem?(a' type_of) {-5} 0 ≤ a' offset {-6} a' offset < max_linear_offset {-7} every(λ (x: number): number_field_pred(x) ∧ real_pred(x) ∧ rational_pred(x) ∧ integer_pred(x) ∧ x ≥ 0 ∧ x < max_byte) (bl') {-8} cons?[Byte](bl') {-9} is_linear_plain_memory?(pm') {-10} (address_block(a', length(bl')) ⊆ pm'rw_addr) {-11} pm' states(s') </pre>	<hr style="border: 0.5px solid black;"/> <pre> {1} a' + length(bl') ≤ reg_size(#type_of := Mem_, offset := max_linear_offset#)(type_of(a')) {2} every(λ (e: [Memory_Address_4G, list[Byte]]): OK?(linear_write_side_effect_in_page(e)(s'))) (split(min_page, a', bl')) </pre>
--	---

Hiding formulas: (-1 -2 -3 -7 2),

Expanding the definition of length,

Using lemma pm_memory_addr,

Simplifying, rewriting, and recording with decision procedures,

Hiding formulas: -6,

Expanding the definition of subset?,

Installing automatic rewrites from: (Mem! max_linear! bus_width!)

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of split_linear_write_side_effects_ok.2.

split_linear_write_side_effects_ok.3:

{-1}	every(λ (a_2 : Address, $bl2$: list[Byte]): rem(expt(2, min_page))(offset(a_2)) + length($bl2$) \leq expt(2, min_page)) (split(min_page, a' , bl'))
{-2}	every(λ (a_2 : Address, $bl2$: list[Byte]): cons?($bl2$))(split(min_page, a' , bl'))
{-3}	every(λ (a_2 : Address, $bl2$: list[Byte]): $a' \leq a_2 \wedge a_2 + \text{length}(bl2) \leq a' + \text{length}(bl')$ (split(min_page, a' , bl'))
{-4}	Mem?(a' 'type_of)
{-5}	$0 \leq a'$ 'offset
{-6}	a' 'offset < max_linear_offset
{-7}	every(λ (x : number): number_field_pred(x) \wedge real_pred(x) \wedge rational_pred(x) \wedge integer_pred(x) $\wedge x \geq 0 \wedge x < \text{max_byte}$) (bl')
{-8}	cons?[Byte](bl')
{-9}	is_linear_plain_memory?(pm')
{-10}	(address_block(a' , length(bl')) $\subseteq pm'$ 'rw_addr)
{-11}	pm' 'states(s')
{1}	reg_base((#type_of := Mem_, offset := 0#))(type_of(a')) $\leq a'$
{2}	every(λ (e : [Memory_Address_4G, list[Byte]]): OK?(linear_write_side_effect_in_page(e)(s')) (split(min_page, a' , bl'))

Expanding the definition of reg_base,

Expanding the definition of <=,

which is trivially true.

This completes the proof of split_linear_write_side_effects_ok.3.

Q.E.D.

C.117.80

Linear_Memory_Properties.split_linear_read_side_effects_data

Terse proof for split_linear_read_side_effects_data.

split_linear_read_side_effects_data:

{1}	\forall (pm : Plain_Memory[Linear_memory], s : Linear_memory[Physical_memory, pm_phy], a : Memory_Address_4G, bl : (cons?[Byte])): is_linear_plain_memory?(pm) \wedge (address_block(a , length(bl)) $\subseteq (pm'$ 'ro_addr $\cup pm'$ 'rw_addr)) $\wedge pm'$ 'states(s) \supset every(λ (e : [Memory_Address_4G, list[Byte]]): OK?(linear_read_side_effect_in_page(e)(s)) \supset data(linear_read_side_effect_in_page(e)(s)) = e' 2) (split(min_page, a , bl))
-----	---

Repeatedly Skolemizing and flattening,

Installing automatic rewrites from: Mem!! max_linear!! min_linear!!

Using lemma split_range,

Using lemma split_no_null,

Using lemma split_type,

Using lemma split_pair_cross_size,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 3 subgoals:

split_linear_read_side_effects_data.1:

<div style="display: flex; flex-direction: column; gap: 5px;"> <div style="display: flex; align-items: flex-start;"> {-1} <div style="margin-left: 10px;"> every(λ (a_2: Address, $bl2$: list[Byte]): rem(expt(2, min_page))(offset(a_2) + length($bl2$) \leq expt(2, min_page)) (split(min_page, a', bl')) </div> </div> <div style="display: flex; align-items: flex-start;"> {-2} <div style="margin-left: 10px;"> every(λ (t: [Address, list[Byte]]): type_of($t'1$) = type_of(a') \wedge reg_base((#type_of := Mem_, offset := 0#))(type_of(a')) \leq $t'1$ \wedge $t'1$ < reg_size((#type_of := Mem_, offset := max_linear_offset#))(type_of(a')) (split(min_page, a', bl')) </div> </div> <div style="display: flex; align-items: flex-start;"> {-3} <div style="margin-left: 10px;"> every(λ (a_2: Address, $bl2$: list[Byte]): cons?($bl2$))(split(min_page, a', bl')) </div> </div> <div style="display: flex; align-items: flex-start;"> {-4} <div style="margin-left: 10px;"> every(λ (a_2: Address, $bl2$: list[Byte]): $a' \leq a_2 \wedge a_2 + \text{length}(bl2) \leq a' + \text{length}(bl')$ (split(min_page, a', bl')) </div> </div> <div style="display: flex; align-items: flex-start;"> {-5} <div style="margin-left: 10px;"> Mem?(a''type_of) </div> </div> <div style="display: flex; align-items: flex-start;"> {-6} <div style="margin-left: 10px;"> $0 \leq a'$'offset </div> </div> <div style="display: flex; align-items: flex-start;"> {-7} <div style="margin-left: 10px;"> a''offset < max_linear_offset </div> </div> <div style="display: flex; align-items: flex-start;"> {-8} <div style="margin-left: 10px;"> every(λ (x: number): number_field_pred(x) \wedge real_pred(x) \wedge rational_pred(x) \wedge integer_pred(x) \wedge $x \geq 0 \wedge x < \text{max_byte}$ (bl') </div> </div> <div style="display: flex; align-items: flex-start;"> {-9} <div style="margin-left: 10px;"> cons?[Byte](bl') </div> </div> <div style="display: flex; align-items: flex-start;"> {-10} <div style="margin-left: 10px;"> is_linear_plain_memory?(pm') </div> </div> <div style="display: flex; align-items: flex-start;"> {-11} <div style="margin-left: 10px;"> (address_block(a', length(bl')) \subseteq (pm''ro_addr \cup pm''rw_addr)) </div> </div> <div style="display: flex; align-items: flex-start;"> {-12} <div style="margin-left: 10px;"> pm''states(s') </div> </div> </div>	<hr style="border: 0.5px solid black; margin-bottom: 5px;"/> <div style="display: flex; align-items: flex-start;"> {1} <div style="margin-left: 10px;"> every(λ (e: [Memory_Address_4G, list[Byte]]): OK?(linear_read_side_effect_in_page(e)(s')) \supset data(linear_read_side_effect_in_page(e)(s')) = $e'2$ (split(min_page, a', bl')) </div> </div>
---	---

Using lemma every_conjunct_left,

Using lemma every_conjunct_left,

we get 3 subgoals:

split_linear_read_side_effects_data.1.1:

```

{-1} every(λ (t: [Address, list[Byte]]):
      cons?(t'2) ∧
      a' ≤ t'1 ∧ t'1 + length(t'2) ≤ a' + length(bl')
      (split(min_page, a', bl'))
      ∧
      every(λ (t: [Address, list[Byte]]):
            type_of(t'1) = type_of(a') ∧
            reg_base(min_linear)(type_of(a')) ≤ t'1 ∧
            t'1 < reg_size(max_linear)(type_of(a'))
            (split(min_page, a', bl'))
            ⊃
            every(λ (t_1: [Address, list[Byte]]):
                  (cons?(t_1'2) ∧
                   a' ≤ t_1'1 ∧ t_1'1 + length(t_1'2) ≤ a' + length(bl'))
                  ∧
                  type_of(t_1'1) = type_of(a') ∧
                  reg_base(min_linear)(type_of(a')) ≤ t_1'1 ∧
                  t_1'1 < reg_size(max_linear)(type_of(a'))
                  (split(min_page, a', bl'))
            )
      )
      every(λ (a_2: Address, bl2: list[Byte]): cons?(bl2))(split(min_page, a', bl')) ∧
      every(λ (a_2: Address, bl2: list[Byte]):
            a' ≤ a_2 ∧ a_2 + length(bl2) ≤ a' + length(bl')
            (split(min_page, a', bl'))
            ⊃
            every(λ (t: [Address, list[Byte]]):
                  cons?(t'2) ∧
                  a' ≤ t'1 ∧ t'1 + length(t'2) ≤ a' + length(bl')
                  (split(min_page, a', bl'))
            )
      )
      every(λ (a_2: Address, bl2: list[Byte]):
            rem(expt(2, min_page))(offset(a_2)) + length(bl2) ≤ expt(2, min_page)
            (split(min_page, a', bl'))
      )
      every(λ (t: [Address, list[Byte]]):
            type_of(t'1) = type_of(a') ∧
            reg_base((#type_of := Mem_, offset := 0#))(type_of(a')) ≤ t'1 ∧
            t'1 < reg_size((#type_of := Mem_, offset := max_linear_offset#))(type_of(a'))
            (split(min_page, a', bl'))
      )
      every(λ (a_2: Address, bl2: list[Byte]): cons?(bl2))(split(min_page, a', bl'))
      every(λ (a_2: Address, bl2: list[Byte]):
            a' ≤ a_2 ∧ a_2 + length(bl2) ≤ a' + length(bl')
            (split(min_page, a', bl'))
      )
      Mem?(a' type_of)
      0 ≤ a' offset
      a' offset < max_linear_offset
      every(λ (x: number):
            number_field_pred(x) ∧
            real_pred(x) ∧
            rational_pred(x) ∧ integer_pred(x) ∧ x ≥ 0 ∧ x < max_byte
            (bl')
      )
      cons?[Byte](bl')
      is_linear_plain_memory?(pm')
      (address_block(a', length(bl')) ⊆ (pm'ro_addr ∪ pm'rw_addr))
      pm' states(s')
      {1} every(λ (e: [Memory_Address_4G, list[Byte]]):
            OK?(linear_read_side_effect_in_page(e)(s')) ⊃
            data(linear_read_side_effect_in_page(e)(s')) = e'2
            (split(min_page, a', bl'))

```

C.117 Proofs for Linear_Memory_Properties (challenge-linear.pvs)

Using lemma `every_conjunct_left`,

we get 3 subgoals:

split_linear_read_side_effects_data.1.1.1:

```

{-1} every(λ (t_1: [Address, list[Byte]]):
  cons?(t_1'2) ∧
  a' ≤ t_1'1 ∧
  t_1'1 + length(t_1'2) ≤ a' + length(bl') ∧
  type_of(t_1'1) = type_of(a') ∧
  reg_base(min_linear)(type_of(a')) ≤ t_1'1 ∧
  t_1'1 < reg_size(max_linear)(type_of(a')))
  (split(min_page, a', bl'))
  ∧
  every(λ (a_2: Address, bl2: list[Byte]):
    rem(expt(2, min_page))(offset(a_2)) + length(bl2) ≤
    expt(2, min_page))
    (split(min_page, a', bl'))
  ⊃
  every(λ (t: [Address, list[Byte]]):
    (cons?(t'2) ∧
     a' ≤ t'1 ∧
     t'1 + length(t'2) ≤ a' + length(bl') ∧
     type_of(t'1) = type_of(a') ∧
     reg_base(min_linear)(type_of(a')) ≤ t'1 ∧
     t'1 < reg_size(max_linear)(type_of(a')))
    ∧
    rem(expt(2, min_page))(offset(t'1)) + length(t'2) ≤
    expt(2, min_page))
    (split(min_page, a', bl'))
{-2} every(λ (t: [Address, list[Byte]]):
  cons?(t'2) ∧
  a' ≤ t'1 ∧ t'1 + length(t'2) ≤ a' + length(bl'))
  (split(min_page, a', bl'))
  ∧
  every(λ (t: [Address, list[Byte]]):
    type_of(t'1) = type_of(a') ∧
    reg_base(min_linear)(type_of(a')) ≤ t'1 ∧
    t'1 < reg_size(max_linear)(type_of(a')))
    (split(min_page, a', bl'))
  ⊃
  every(λ (t_1: [Address, list[Byte]]):
    (cons?(t_1'2) ∧
     a' ≤ t_1'1 ∧ t_1'1 + length(t_1'2) ≤ a' + length(bl'))
    ∧
    type_of(t_1'1) = type_of(a') ∧
    reg_base(min_linear)(type_of(a')) ≤ t_1'1 ∧
    t_1'1 < reg_size(max_linear)(type_of(a')))
    (split(min_page, a', bl'))
{-3} every(λ (a_2: Address, bl2: list[Byte]): cons?(bl2))(split(min_page, a', bl')) ∧
  every(λ (a_2: Address, bl2: list[Byte]):
    a' ≤ a_2 ∧ a_2 + length(bl2) ≤ a' + length(bl'))
  (split(min_page, a', bl'))
  ⊃
  every(λ (t: [Address, list[Byte]]):
    cons?(t'2) ∧
    a' ≤ t'1 ∧ t'1 + length(t'2) ≤ a' + length(bl'))
    (split(min_page, a', bl'))
{-4} every(λ (a_2: Address, bl2: list[Byte]):
  rem(expt(2, min_page))(offset(a_2)) + length(bl2) ≤ expt(2, min_page))
  (split(min_page, a', bl'))
{-5} every(λ (t: [Address, list[Byte]]):
  type_of(t'1) = type_of(a') ∧
  reg_base((#type_of := Mem_, offset := 0#))(type_of(a')) ≤ t'1 ∧
  t'1 < reg_size((#type_of := Mem_, offset := max_linear_offset#))(type_of(a')))
  (split(min_page, a', bl'))
{-6} every(λ (a_2: Address, bl2: list[Byte]): cons?(bl2))(split(min_page, a', bl'))
{-7} every(λ (a_2: Address, bl2: list[Byte]):

```

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Hiding formulas: (-1 -3 -4 -5 -6 -7 -11),

Installing automatic rewrites from: (max_linear! Mem! ##! expr_2_super! expr_2_super_res!)

Using lemma every_extend[[Address, list[Byte]], [Memory_Address_4G, list[Byte]]],

we get 2 subgoals:

split_linear_read_side_effects_data.1.1.1.1:

<pre> {-1} every(λ (t_1: [Address, list[Byte]]): cons?(t_1'2) ∧ a' ≤ t_1'1 ∧ t_1'1 + length(t_1'2) ≤ a' + length(bl') ∧ type_of(t_1'1) = type_of(a') ∧ reg_base(min_linear)(type_of(a')) ≤ t_1'1 ∧ t_1'1 < reg_size(max_linear)(type_of(a')) ∧ rem(expt(2, min_page))(offset(t_1'1)) + length(t_1'2) ≤ expt(2, min_page) (split(min_page, a', bl')) ⊃ every(λ (s: [Memory_Address_4G, list[Byte]]): cons?(s'2) ∧ a' ≤ s'1 ∧ s'1 + length(s'2) ≤ a' + length(bl') ∧ type_of(s'1) = type_of(a') ∧ reg_base(min_linear)(type_of(a')) ≤ s'1 ∧ s'1 < reg_size(max_linear)(type_of(a')) ∧ rem(expt(2, min_page))(offset(s'1)) + length(s'2) ≤ expt(2, min_page) (split(min_page, a', bl'))) {-2} every(λ (t: [Address, list[Byte]]): (cons?(t'2) ∧ a' ≤ t'1 ∧ t'1 + length(t'2) ≤ a' + length(bl') ∧ type_of(t'1) = type_of(a') ∧ reg_base((#type_of := Mem_, offset := 0#))(type_of(a')) ≤ t'1 ∧ t'1 < reg_size((#type_of := Mem_, offset := max_linear_offset#))(type_of(a'))) ∧ rem(expt(2, min_page))(offset(t'1)) + length(t'2) ≤ expt(2, min_page) (split(min_page, a', bl')) {-3} Mem?(a' type_of) {-4} 0 ≤ a' offset {-5} a' offset < max_linear_offset {-6} cons?[Byte](bl') {-7} is_linear_plain_memory?(pm') {-8} (address_block(a', length(bl')) ⊆ (pm'ro_addr ∪ pm'rw_addr)) {-9} pm' states(s') </pre>	<hr style="border: 0.5px solid black;"/> <pre> {1} every(λ (e: [Memory_Address_4G, list[Byte]]): OK?(linear_read_side_effect_in_page(e)(s')) ⊃ data(linear_read_side_effect_in_page(e)(s')) = e'2 (split(min_page, a', bl')) </pre>
--	--

Using lemma every_implied,

we get 2 subgoals:

split_linear_read_side_effects_data.1.1.1.1.1:

```

{-1}  (∀ (t: [Memory_Address_4G, list[Byte]]):
      cons?(t'2) ∧
      a' ≤ t'1 ∧
      t'1 + length(t'2) ≤ a' + length(bl') ∧
      type_of(t'1) = type_of(a') ∧
      reg_base(min_linear)(type_of(a')) ≤ t'1 ∧
      t'1 < reg_size(max_linear)(type_of(a')) ∧
      rem(expt(2, min_page))(offset(t'1)) + length(t'2) ≤
      expt(2, min_page)
      ⊃
      OK?(linear_read_side_effect_in_page(t)(s')) ⊃
      data(linear_read_side_effect_in_page(t)(s')) = t'2)
      ∧
      every(λ (s: [Memory_Address_4G, list[Byte]]):
        cons?(s'2) ∧
        a' ≤ s'1 ∧
        s'1 + length(s'2) ≤ a' + length(bl') ∧
        type_of(s'1) = type_of(a') ∧
        reg_base(min_linear)(type_of(a')) ≤ s'1 ∧
        s'1 < reg_size(max_linear)(type_of(a')) ∧
        rem(expt(2, min_page))(offset(s'1)) + length(s'2) ≤
        expt(2, min_page))
        (split(min_page, a', bl'))
      ⊃
      every(λ (e: [Memory_Address_4G, list[Byte]]):
        OK?(linear_read_side_effect_in_page(e)(s')) ⊃
        data(linear_read_side_effect_in_page(e)(s')) = e'2)
        (split(min_page, a', bl'))
{-2}  every(λ (t_1: [Address, list[Byte]]):
      cons?(t_1'2) ∧
      a' ≤ t_1'1 ∧
      t_1'1 + length(t_1'2) ≤ a' + length(bl') ∧
      type_of(t_1'1) = type_of(a') ∧
      reg_base(min_linear)(type_of(a')) ≤ t_1'1 ∧
      t_1'1 < reg_size(max_linear)(type_of(a')) ∧
      rem(expt(2, min_page))(offset(t_1'1)) + length(t_1'2) ≤
      expt(2, min_page))
      (split(min_page, a', bl'))
      ⊃
      every(λ (s: [Memory_Address_4G, list[Byte]]):
        cons?(s'2) ∧
        a' ≤ s'1 ∧
        s'1 + length(s'2) ≤ a' + length(bl') ∧
        type_of(s'1) = type_of(a') ∧
        reg_base(min_linear)(type_of(a')) ≤ s'1 ∧
        s'1 < reg_size(max_linear)(type_of(a')) ∧
        rem(expt(2, min_page))(offset(s'1)) + length(s'2) ≤
        expt(2, min_page))
        (split(min_page, a', bl'))
{-3}  every(λ (t: [Address, list[Byte]]):
      (cons?(t'2) ∧
      a' ≤ t'1 ∧
      t'1 + length(t'2) ≤ a' + length(bl') ∧
      type_of(t'1) = type_of(a') ∧
      reg_base((#type_of := Mem_, offset := 0#))(type_of(a')) ≤ t'1 ∧
      t'1 <
      reg_size((#type_of := Mem_, offset := max_linear_offset#))(type_of(a')))
      ∧
      rem(expt(2, min_page))(offset(t'1)) + length(t'2) ≤
      expt(2, min_page))
      (split(min_page, a', bl'))
{-4}  Mem?(a' type_of)
{-5}  0 ≤ a' offset

```

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Hiding formulas: (-1 -2 2),

Expanding the definition of linear_read_side_effect_in_page,

Repeatedly Skolemizing and flattening,

Case splitting on union(pm!1'ro_addr, pm!1'rw_addr)(t!1'1),

we get 2 subgoals:

split_linear_read_side_effects_data.1.1.1.1.1.1:

<p>{-1} union(pm'ro_addr, pm'rw_addr)(t'1)</p> <p>{-2} cons?(t'2)</p> <p>{-3} a' ≤ t'1</p> <p>{-4} t'1 + length(t'2) ≤ a' + length(bl')</p> <p>{-5} type_of(t'1) = type_of(a')</p> <p>{-6} reg_base((#type_of := Mem_, offset := 0#))(type_of(a')) ≤ t'1</p> <p>{-7} t'1 < reg_size((#type_of := Mem_, offset := max_linear_offset#))(type_of(a'))</p> <p>{-8} rem(expt(2, min_page))(offset(t'1) + length(t'2)) ≤ expt(2, min_page)</p> <p>{-9} OK?(linear_resolve(t'1, Read) ## (λ (pa: Memory_Address_4G)(ns: Linear_memory[Physical_memory, pm_phy]): memory_read_side_effect(pm_phy'mem)(pa, t'2, TRUE)(s')) (s'))</p> <p>{-10} Mem?(a'type_of)</p> <p>{-11} 0 ≤ a'offset</p> <p>{-12} a'offset < max_linear_offset</p> <p>{-13} cons?[Byte](bl')</p> <p>{-14} is_linear_plain_memory?(pm')</p> <p>{-15} (address_block(a', length(bl'))) ⊆ (pm'ro_addr ∪ pm'rw_addr)</p> <p>{-16} pm'states(s')</p>	<p>{1} data(linear_resolve(t'1, Read) ## (λ (pa: Memory_Address_4G)(ns: Linear_memory[Physical_memory, pm_phy]): memory_read_side_effect(pm_phy'mem)(pa, t'2, TRUE)(s')) (s')) = t'2</p>
--	--

Using lemma pm_linear_resolve_read_ok,

Case splitting on subset?(address_block(data(linear_resolve(t!1'1, Read)(s!1)), length(t!1'2)), union(pm_phy'ro_addr, pm_phy'rw_addr)),

we get 3 subgoals:

split_linear_read_side_effects_data.1.1.1.1.1.1.1:

{-1}	$(\text{address_block}(\text{data}(\text{linear_resolve}(t'1, \text{Read})(s')), \text{length}(t'2)) \subseteq (\text{pm_phy'ro_addr} \cup \text{pm_phy'rw_addr}))$
{-2}	$\text{is_linear_plain_memory?}(\text{pm}') \supset \text{OK?}(\text{linear_resolve}(t'1, \text{Read})(s'))$
{-3}	$\text{union}(\text{pm}'\text{ro_addr}, \text{pm}'\text{rw_addr})(t'1)$
{-4}	$\text{cons?}(t'2)$
{-5}	$a' \leq t'1$
{-6}	$t'1 + \text{length}(t'2) \leq a' + \text{length}(b')$
{-7}	$\text{type_of}(t'1) = \text{type_of}(a')$
{-8}	$\text{reg_base}((\#\text{type_of} := \text{Mem_}, \text{offset} := 0\#))(\text{type_of}(a')) \leq t'1$
{-9}	$t'1 < \text{reg_size}((\#\text{type_of} := \text{Mem_}, \text{offset} := \text{max_linear_offset}\#))(\text{type_of}(a'))$
{-10}	$\text{rem}(\text{expt}(2, \text{min_page}))(\text{offset}(t'1)) + \text{length}(t'2) \leq \text{expt}(2, \text{min_page})$
{-11}	$\text{OK?}((\text{linear_resolve}(t'1, \text{Read}) \#\#$ $(\lambda (\text{pa}: \text{Memory_Address_4G})(\text{ns}: \text{Linear_memory}[\text{Physical_memory}, \text{pm_phy}]):$ $\text{memory_read_side_effect}(\text{pm_phy}'\text{mem})(\text{pa}, t'2, \text{TRUE})(s'))$ $(s'))$
{-12}	$\text{Mem?}(a' \text{ type_of})$
{-13}	$0 \leq a' \text{ offset}$
{-14}	$a' \text{ offset} < \text{max_linear_offset}$
{-15}	$\text{cons?}[\text{Byte}](b')$
{-16}	$\text{is_linear_plain_memory?}(\text{pm}')$
{-17}	$(\text{address_block}(a', \text{length}(b')) \subseteq (\text{pm}'\text{ro_addr} \cup \text{pm}'\text{rw_addr}))$
{-18}	$\text{pm}'\text{states}(s')$
{1}	$\text{data}((\text{linear_resolve}(t'1, \text{Read}) \#\#$ $(\lambda (\text{pa}: \text{Memory_Address_4G})(\text{ns}: \text{Linear_memory}[\text{Physical_memory}, \text{pm_phy}]):$ $\text{memory_read_side_effect}(\text{pm_phy}'\text{mem})(\text{pa}, t'2, \text{TRUE})(s'))$ $(s'))$ $= t'2$

Using lemma linear_resolve_states,

Using lemma pm_states,

Using lemma pm_plain_phy,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Expanding the definition of plain_memory?,

Applying disjunctive simplification to flatten sequent,

Hiding formulas: (-2 -3 -4 -5 -6 -8),

Expanding the definition of side_effect_content_unchanged,

Instantiating quantified variables,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of split_linear_read_side_effects_data.1.1.1.1.1.1.1.

split_linear_read_side_effects_data.1.1.1.1.1.1.2:

<pre> {-1} is_linear_plain_memory?(pm') ⊃ OK?(linear_resolve(t'1, Read)(s')) {-2} union(pm'ro_addr, pm'rw_addr)(t'1) {-3} cons?(t'2) {-4} a' ≤ t'1 {-5} t'1 + length(t'2) ≤ a' + length(bl') {-6} type_of(t'1) = type_of(a') {-7} reg_base((#type_of := Mem_, offset := 0#))(type_of(a')) ≤ t'1 {-8} t'1 < reg_size((#type_of := Mem_, offset := max_linear_offset#))(type_of(a')) {-9} rem(expt(2, min_page))(offset(t'1)) + length(t'2) ≤ expt(2, min_page) {-10} OK?((linear_resolve(t'1, Read) ## (λ (pa: Memory_Address_4G)(ns: Linear_memory[Physical_memory, pm_phy]): memory_read_side_effect(pm_phy'mem)(pa, t'2, TRUE)(s'))) (s')) {-11} Mem?(a'type_of) {-12} 0 ≤ a'offset {-13} a'offset < max_linear_offset {-14} cons?[Byte](bl') {-15} is_linear_plain_memory?(pm') {-16} (address_block(a', length(bl'))) ⊆ (pm'ro_addr ∪ pm'rw_addr) {-17} pm'states(s') </pre>	<pre> {1} (address_block(data(linear_resolve(t'1, Read)(s')), length(t'2)) ⊆ (pm_phy'ro_addr ∪ pm_phy'rw_addr)) {2} data((linear_resolve(t'1, Read) ## (λ (pa: Memory_Address_4G)(ns: Linear_memory[Physical_memory, pm_phy]): memory_read_side_effect(pm_phy'mem)(pa, t'2, TRUE)(s'))) (s')) = t'2 </pre>
--	--

Hiding formulas: 2,

Using lemma same_page_address_block_read,

we get 3 subgoals:

split_linear_read_side_effects_data.1.1.1.1.1.1.2.1:

{-1}	is_linear_plain_memory?(pm') \wedge pm' 'states(s') \wedge union(pm' 'ro_addr, pm' 'rw_addr)(t' '1) \wedge a' \leq t' '1 \wedge t' '1 + length(t' '2) \leq a' + length(bl') \wedge rem(expt(2, min_page))(offset(t' '1)) + length(t' '2) \leq expt(2, min_page) \wedge (address_block(a', length(bl')) \subseteq (pm' 'ro_addr \cup pm' 'rw_addr)) \supset (address_block(data(linear_resolve(t' '1, Read)(s')), length(t' '2)) \subseteq (pm_phy'ro_addr \cup pm_phy'rw_addr))
{-2}	is_linear_plain_memory?(pm') \supset OK?(linear_resolve(t' '1, Read)(s'))
{-3}	union(pm' 'ro_addr, pm' 'rw_addr)(t' '1)
{-4}	cons?(t' '2)
{-5}	a' \leq t' '1
{-6}	t' '1 + length(t' '2) \leq a' + length(bl')
{-7}	type_of(t' '1) = type_of(a')
{-8}	reg_base((#type_of := Mem_, offset := 0#))(type_of(a')) \leq t' '1
{-9}	t' '1 < reg_size((#type_of := Mem_, offset := max_linear_offset#))(type_of(a'))
{-10}	rem(expt(2, min_page))(offset(t' '1)) + length(t' '2) \leq expt(2, min_page)
{-11}	OK?((linear_resolve(t' '1, Read) ## (λ (pa: Memory_Address_4G)(ns: Linear_memory[Physical_memory, pm_phy]): memory_read_side_effect(pm_phy' 'mem)(pa, t' '2, TRUE)(s')) (s'))
{-12}	Mem?(a' 'type_of)
{-13}	0 \leq a' 'offset
{-14}	a' 'offset < max_linear_offset
{-15}	cons?[Byte](bl')
{-16}	is_linear_plain_memory?(pm')
{-17}	(address_block(a', length(bl')) \subseteq (pm' 'ro_addr \cup pm' 'rw_addr))
{-18}	pm' 'states(s')
{1}	(address_block(data(linear_resolve(t' '1, Read)(s')), length(t' '2)) \subseteq (pm_phy'ro_addr \cup pm_phy'rw_addr))

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of split_linear_read_side_effects_data.1.1.1.1.1.1.2.1.

split_linear_read_side_effects_data.1.1.1.1.1.1.2.2:

{-1}	is_linear_plain_memory?(pm') \supset OK?(linear_resolve(t'1, Read)(s'))
{-2}	union(pm'ro_addr, pm'rw_addr)(t'1)
{-3}	cons?(t'2)
{-4}	$a' \leq t'1$
{-5}	$t'1 + \text{length}(t'2) \leq a' + \text{length}(bl')$
{-6}	type_of(t'1) = type_of(a')
{-7}	reg_base((#type_of := Mem_, offset := 0#))(type_of(a')) $\leq t'1$
{-8}	$t'1 < \text{reg_size}((\#type_of := \text{Mem_}, \text{offset} := \text{max_linear_offset\#}))(type_of(a'))$
{-9}	$\text{rem}(\text{expt}(2, \text{min_page}))(\text{offset}(t'1)) + \text{length}(t'2) \leq \text{expt}(2, \text{min_page})$
{-10}	OK?((linear_resolve(t'1, Read) ## $(\lambda (\text{pa}: \text{Memory_Address_4G})(\text{ns}: \text{Linear_memory}[\text{Physical_memory}, \text{pm_phy}]):$ $\text{memory_read_side_effect}(\text{pm_phy}'\text{mem})(\text{pa}, t'2, \text{TRUE})(s'))$ $(s'))$)
{-11}	Mem?(a'type_of)
{-12}	$0 \leq a'\text{offset}$
{-13}	$a'\text{offset} < \text{max_linear_offset}$
{-14}	cons?[Byte](bl')
{-15}	is_linear_plain_memory?(pm')
{-16}	$(\text{address_block}(a', \text{length}(bl'))) \subseteq (\text{pm}'\text{ro_addr} \cup \text{pm}'\text{rw_addr})$
{-17}	pm'states(s')
{1}	$\text{length}[\text{Byte}](t'2) > 0$
{2}	$(\text{address_block}(\text{data}(\text{linear_resolve}(t'1, \text{Read})(s')), \text{length}(t'2))) \subseteq (\text{pm_phy}'\text{ro_addr} \cup \text{pm_phy}'\text{rw_addr})$

Expanding the definition of length,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of split_linear_read_side_effects_data.1.1.1.1.1.1.2.2.

split_linear_read_side_effects_data.1.1.1.1.1.1.2.3:

{-1}	is_linear_plain_memory?(pm') \supset OK?(linear_resolve(t'1, Read)(s'))
{-2}	union(pm'ro_addr, pm'rw_addr)(t'1)
{-3}	cons?(t'2)
{-4}	$a' \leq t'1$
{-5}	$t'1 + \text{length}(t'2) \leq a' + \text{length}(bl')$
{-6}	type_of(t'1) = type_of(a')
{-7}	reg_base((#type_of := Mem_, offset := 0#))(type_of(a')) $\leq t'1$
{-8}	$t'1 < \text{reg_size}((\#type_of := \text{Mem_}, \text{offset} := \text{max_linear_offset\#}))(type_of(a'))$
{-9}	$\text{rem}(\text{expt}(2, \text{min_page}))(\text{offset}(t'1)) + \text{length}(t'2) \leq \text{expt}(2, \text{min_page})$
{-10}	OK?((linear_resolve(t'1, Read) ## $(\lambda (\text{pa}: \text{Memory_Address_4G})(\text{ns}: \text{Linear_memory}[\text{Physical_memory}, \text{pm_phy}]):$ $\text{memory_read_side_effect}(\text{pm_phy}'\text{mem})(\text{pa}, t'2, \text{TRUE})(s'))$ $(s'))$)
{-11}	Mem?(a'type_of)
{-12}	$0 \leq a'\text{offset}$
{-13}	$a'\text{offset} < \text{max_linear_offset}$
{-14}	cons?[Byte](bl')
{-15}	is_linear_plain_memory?(pm')
{-16}	$(\text{address_block}(a', \text{length}(bl'))) \subseteq (\text{pm}'\text{ro_addr} \cup \text{pm}'\text{rw_addr})$
{-17}	pm'states(s')
{1}	$\text{length}[\text{Byte}](bl') > 0$
{2}	$(\text{address_block}(\text{data}(\text{linear_resolve}(t'1, \text{Read})(s')), \text{length}(t'2))) \subseteq (\text{pm_phy}'\text{ro_addr} \cup \text{pm_phy}'\text{rw_addr})$

Expanding the definition of length,

C Proof scripts

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `split_linear_read_side_effects_data.1.1.1.1.1.1.2.3`.

`split_linear_read_side_effects_data.1.1.1.1.1.1.3`:

{-1}	<code>is_linear_plain_memory?(pm') \supset OK?(linear_resolve(t'1, Read)(s'))</code>
{-2}	<code>union(pm'ro_addr, pm'rw_addr)(t'1)</code>
{-3}	<code>cons?(t'2)</code>
{-4}	<code>a' \leq t'1</code>
{-5}	<code>t'1 + length(t'2) \leq a' + length(bl')</code>
{-6}	<code>type_of(t'1) = type_of(a')</code>
{-7}	<code>reg_base((#type_of := Mem_, offset := 0#))(type_of(a')) \leq t'1</code>
{-8}	<code>t'1 < reg_size((#type_of := Mem_, offset := max_linear_offset#))(type_of(a'))</code>
{-9}	<code>rem(expt(2, min_page))(offset(t'1)) + length(t'2) \leq expt(2, min_page)</code>
{-10}	<code>OK?(linear_resolve(t'1, Read) ## $(\lambda$ (pa: Memory_Address_4G)(ns: Linear_memory[Physical_memory, pm_phy]): memory_read_side_effect(pm_phy'mem)(pa, t'2, TRUE)(s')) (s'))</code>
{-11}	<code>Mem?(a'type_of)</code>
{-12}	<code>0 \leq a'offset</code>
{-13}	<code>a'offset < max_linear_offset</code>
{-14}	<code>cons?[Byte](bl')</code>
{-15}	<code>is_linear_plain_memory?(pm')</code>
{-16}	<code>(address_block(a', length(bl')) \subseteq (pm'ro_addr \cup pm'rw_addr))</code>
{-17}	<code>pm'states(s')</code>
{1}	<code>OK?[Physical_memory, Address](linear_resolve[Physical_memory, pm_phy](t'1, Read)(s'))</code>
{2}	<code>data((linear_resolve(t'1, Read) ## $(\lambda$ (pa: Memory_Address_4G)(ns: Linear_memory[Physical_memory, pm_phy]): memory_read_side_effect(pm_phy'mem)(pa, t'2, TRUE)(s')) (s')) = t'2</code>

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `split_linear_read_side_effects_data.1.1.1.1.1.1.3`.

split_linear_read_side_effects_data.1.1.1.1.1.2:

{-1}	cons?(t'2)
{-2}	a' ≤ t'1
{-3}	t'1 + length(t'2) ≤ a' + length(bl')
{-4}	type_of(t'1) = type_of(a')
{-5}	reg_base((#type_of := Mem_, offset := 0#))(type_of(a')) ≤ t'1
{-6}	t'1 < reg_size((#type_of := Mem_, offset := max_linear_offset#))(type_of(a'))
{-7}	rem(expt(2, min_page))(offset(t'1) + length(t'2)) ≤ expt(2, min_page)
{-8}	OK?(linear_resolve(t'1, Read) ## $(\lambda (pa: \text{Memory_Address_4G})(ns: \text{Linear_memory}[\text{Physical_memory}, pm_phy]):$ $\text{memory_read_side_effect}(pm_phy \text{ 'mem})(pa, t'2, \text{TRUE})(s'))$ $(s'))$
{-9}	Mem?(a' type_of)
{-10}	0 ≤ a' offset
{-11}	a' offset < max_linear_offset
{-12}	cons?[Byte](bl')
{-13}	is_linear_plain_memory?(pm')
{-14}	(address_block(a', length(bl'))) ⊆ (pm' ro_addr ∪ pm' rw_addr)
{-15}	pm' states(s')
{1}	union(pm' ro_addr, pm' rw_addr)(t'1)
{2}	data((linear_resolve(t'1, Read) ## $(\lambda (pa: \text{Memory_Address_4G})(ns: \text{Linear_memory}[\text{Physical_memory}, pm_phy]):$ $\text{memory_read_side_effect}(pm_phy \text{ 'mem})(pa, t'2, \text{TRUE})(s'))$ $(s'))$ $= t'2$

Hiding formulas: (-7 -8 -13 2),

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of split_linear_read_side_effects_data.1.1.1.1.1.2.

split_linear_read_side_effects_data.1.1.1.1.2:

```

{-1} every(λ (t_1: [Address, list[Byte]]):
      cons?(t_1'2) ∧
      a' ≤ t_1'1 ∧
      t_1'1 + length(t_1'2) ≤ a' + length(bl') ∧
      type_of(t_1'1) = type_of(a') ∧
      reg_base(min_linear)(type_of(a')) ≤ t_1'1 ∧
      t_1'1 < reg_size(max_linear)(type_of(a')) ∧
      rem(expt(2, min_page))(offset(t_1'1)) + length(t_1'2) ≤
      expt(2, min_page))
      (split(min_page, a', bl'))
  ⊃
  every(λ (s: [Memory_Address_4G, list[Byte]]):
      cons?(s'2) ∧
      a' ≤ s'1 ∧
      s'1 + length(s'2) ≤ a' + length(bl') ∧
      type_of(s'1) = type_of(a') ∧
      reg_base(min_linear)(type_of(a')) ≤ s'1 ∧
      s'1 < reg_size(max_linear)(type_of(a')) ∧
      rem(expt(2, min_page))(offset(s'1)) + length(s'2) ≤
      expt(2, min_page))
      (split(min_page, a', bl'))
{-2} every(λ (t: [Address, list[Byte]]):
      (cons?(t'2) ∧
      a' ≤ t'1 ∧
      t'1 + length(t'2) ≤ a' + length(bl') ∧
      type_of(t'1) = type_of(a') ∧
      reg_base((#type_of := Mem_, offset := 0#))(type_of(a')) ≤ t'1 ∧
      t'1 <
      reg_size((#type_of := Mem_, offset := max_linear_offset#))(type_of(a')))
      ∧
      rem(expt(2, min_page))(offset(t'1)) + length(t'2) ≤
      expt(2, min_page))
      (split(min_page, a', bl'))
{-3} Mem?(a' type_of)
{-4} 0 ≤ a' offset
{-5} a' offset < max_linear_offset
{-6} cons?[Byte](bl')
{-7} is_linear_plain_memory?(pm')
{-8} (address_block(a', length(bl'))) ⊆ (pm' ro_addr ∪ pm' rw_addr)
{-9} pm' states(s')
-----
{1} ∀ (s: [Memory_Address_4G, list[Byte]]):
      type_of(s'1) = type_of(a') ∧
      s'1 + length(s'2) ≤ a' + length(bl') ∧ a' ≤ s'1 ∧ cons?(s'2)
      ⊃ Mem?(Mem(0) type_of)
{2} every(λ (e: [Memory_Address_4G, list[Byte]]):
      OK?(linear_read_side_effect_in_page(e)(s')) ⊃
      data(linear_read_side_effect_in_page(e)(s')) = e'2)
      (split(min_page, a', bl'))

```

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `split_linear_read_side_effects_data.1.1.1.1.2`.

split_linear_read_side_effects_data.1.1.1.2:

<p>{-1} every(λ (t: [Address, list[Byte]]): (cons?($t'2$) \wedge $a' \leq t'1 \wedge$ $t'1 + \text{length}(t'2) \leq a' + \text{length}(bl') \wedge$ type_of($t'1$) = type_of(a') \wedge reg_base((#type_of := Mem_, offset := 0#))(type_of(a')) $\leq t'1 \wedge$ $t'1 <$ reg_size((#type_of := Mem_, offset := max_linear_offset#))(type_of(a'))) \wedge rem(expt(2, min_page))(offset($t'1$)) + length($t'2$) \leq expt(2, min_page)) (split(min_page, a', bl'))</p> <p>{-2} Mem?(a''type_of)</p> <p>{-3} $0 \leq a'$'offset</p> <p>{-4} a''offset < max_linear_offset</p> <p>{-5} cons?[Byte](bl')</p> <p>{-6} is_linear_plain_memory?(pm')</p> <p>{-7} (address_block(a', length(bl'))) \subseteq (pm'ro_addr \cup pm'rw_addr)</p> <p>{-8} pm' states(s')</p>	<hr/> <p>{1} \forall (t: [Address, list[Byte]]): type_of($t'1$) = type_of(a') \wedge $t'1 + \text{length}(t'2) \leq a' + \text{length}(bl') \wedge a' \leq t'1 \wedge$ cons?($t'2$) \supset Mem?(Mem(0)'type_of)</p> <p>{2} every(λ (e: [Memory_Address_4G, list[Byte]]): OK?(linear_read_side_effect_in_page(e)(s')) \supset data(linear_read_side_effect_in_page(e)(s')) = $e'2$) (split(min_page, a', bl'))</p>
--	---

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of split_linear_read_side_effects_data.1.1.1.2.

split_linear_read_side_effects_data.1.1.2:

```

{-1} every(λ (t: [Address, list[Byte]]):
      cons?(t'2) ∧
      a' ≤ t'1 ∧ t'1 + length(t'2) ≤ a' + length(bl')
      (split(min_page, a', bl'))
    ∧
    every(λ (t: [Address, list[Byte]]):
      type_of(t'1) = type_of(a') ∧
      reg_base(min_linear)(type_of(a')) ≤ t'1 ∧
      t'1 < reg_size(max_linear)(type_of(a'))
      (split(min_page, a', bl'))
    ⊃
    every(λ (t_1: [Address, list[Byte]]):
      (cons?(t_1'2) ∧
      a' ≤ t_1'1 ∧ t_1'1 + length(t_1'2) ≤ a' + length(bl'))
      ∧
      type_of(t_1'1) = type_of(a') ∧
      reg_base(min_linear)(type_of(a')) ≤ t_1'1 ∧
      t_1'1 < reg_size(max_linear)(type_of(a'))
      (split(min_page, a', bl'))
  {-2} every(λ (a_2: Address, bl2: list[Byte]): cons?(bl2))(split(min_page, a', bl')) ∧
    every(λ (a_2: Address, bl2: list[Byte]):
      a' ≤ a_2 ∧ a_2 + length(bl2) ≤ a' + length(bl')
      (split(min_page, a', bl'))
    ⊃
    every(λ (t: [Address, list[Byte]]):
      cons?(t'2) ∧
      a' ≤ t'1 ∧ t'1 + length(t'2) ≤ a' + length(bl')
      (split(min_page, a', bl'))
  {-3} every(λ (a_2: Address, bl2: list[Byte]):
    rem(expt(2, min_page))(offset(a_2)) + length(bl2) ≤ expt(2, min_page))
    (split(min_page, a', bl'))
  {-4} every(λ (t: [Address, list[Byte]]):
    type_of(t'1) = type_of(a') ∧
    reg_base((#type_of := Mem_, offset := 0#))(type_of(a')) ≤ t'1 ∧
    t'1 < reg_size((#type_of := Mem_, offset := max_linear_offset#))(type_of(a'))
    (split(min_page, a', bl'))
  {-5} every(λ (a_2: Address, bl2: list[Byte]): cons?(bl2))(split(min_page, a', bl'))
  {-6} every(λ (a_2: Address, bl2: list[Byte]):
    a' ≤ a_2 ∧ a_2 + length(bl2) ≤ a' + length(bl')
    (split(min_page, a', bl'))
  {-7} Mem?(a' type_of)
  {-8} 0 ≤ a' offset
  {-9} a' offset < max_linear_offset
  {-10} every(λ (x: number):
    number_field_pred(x) ∧
    real_pred(x) ∧
    rational_pred(x) ∧ integer_pred(x) ∧ x ≥ 0 ∧ x < max_byte)
    (bl')
  {-11} cons?[Byte](bl')
  {-12} is_linear_plain_memory?(pm')
  {-13} (address_block(a', length(bl'))) ⊆ (pm'ro_addr ∪ pm'rw_addr)
  {-14} pm' states(s')
  {1} ∀ (t_1: [Address, list[Byte]]):
    reg_base(Mem(0))(type_of(a')) ≤ t_1'1 ∧
    type_of(t_1'1) = type_of(a') ∧
    t_1'1 + length(t_1'2) ≤ a' + length(bl') ∧
    a' ≤ t_1'1 ∧ cons?(t_1'2)
    ⊃ Mem?(Mem(max_linear_offset) type_of)
  {2} every(λ (e: [Memory_Address_4G, list[Byte]]):
    OK?(linear_read_side_effect_in_page(e)(s')) ⊃
    data(linear_read_side_effect_in_page(e)(s')) = e'2
    (split(min_page, a', bl'))

```

C.117 Proofs for Linear_Memory_Properties (challenge-linear.pvs)

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `split_linear_read_side_effects_data.1.1.2`.

split_linear_read_side_effects_data.1.1.3:

```

{-1} every(λ (t: [Address, list[Byte]]):
      cons?(t'2) ∧
      a' ≤ t'1 ∧ t'1 + length(t'2) ≤ a' + length(bl')
      (split(min_page, a', bl'))
    ∧
    every(λ (t: [Address, list[Byte]]):
      type_of(t'1) = type_of(a') ∧
      reg_base(min_linear)(type_of(a')) ≤ t'1 ∧
      t'1 < reg_size(max_linear)(type_of(a'))
      (split(min_page, a', bl'))
    ⊃
    every(λ (t_1: [Address, list[Byte]]):
      (cons?(t_1'2) ∧
      a' ≤ t_1'1 ∧ t_1'1 + length(t_1'2) ≤ a' + length(bl'))
      ∧
      type_of(t_1'1) = type_of(a') ∧
      reg_base(min_linear)(type_of(a')) ≤ t_1'1 ∧
      t_1'1 < reg_size(max_linear)(type_of(a'))
      (split(min_page, a', bl'))
    {-2} every(λ (a_2: Address, bl2: list[Byte]): cons?(bl2))(split(min_page, a', bl')) ∧
    every(λ (a_2: Address, bl2: list[Byte]):
      a' ≤ a_2 ∧ a_2 + length(bl2) ≤ a' + length(bl')
      (split(min_page, a', bl'))
    ⊃
    every(λ (t: [Address, list[Byte]]):
      cons?(t'2) ∧
      a' ≤ t'1 ∧ t'1 + length(t'2) ≤ a' + length(bl')
      (split(min_page, a', bl'))
    {-3} every(λ (a_2: Address, bl2: list[Byte]):
      rem(expt(2, min_page))(offset(a_2)) + length(bl2) ≤ expt(2, min_page))
      (split(min_page, a', bl'))
    {-4} every(λ (t: [Address, list[Byte]]):
      type_of(t'1) = type_of(a') ∧
      reg_base((#type_of := Mem_, offset := 0#))(type_of(a')) ≤ t'1 ∧
      t'1 < reg_size((#type_of := Mem_, offset := max_linear_offset#))(type_of(a'))
      (split(min_page, a', bl'))
    {-5} every(λ (a_2: Address, bl2: list[Byte]): cons?(bl2))(split(min_page, a', bl'))
    {-6} every(λ (a_2: Address, bl2: list[Byte]):
      a' ≤ a_2 ∧ a_2 + length(bl2) ≤ a' + length(bl')
      (split(min_page, a', bl'))
    {-7} Mem?(a' type_of)
    {-8} 0 ≤ a' offset
    {-9} a' offset < max_linear_offset
    {-10} every(λ (x: number):
      number_field_pred(x) ∧
      real_pred(x) ∧
      rational_pred(x) ∧ integer_pred(x) ∧ x ≥ 0 ∧ x < max_byte)
      (bl')
    {-11} cons?[Byte](bl')
    {-12} is_linear_plain_memory?(pm')
    {-13} (address_block(a', length(bl'))) ⊆ (pm'ro_addr ∪ pm'rw_addr)
    {-14} pm' states(s')
  {1} ∀ (t_1: [Address, list[Byte]]):
      type_of(t_1'1) = type_of(a') ∧
      t_1'1 + length(t_1'2) ≤ a' + length(bl') ∧
      a' ≤ t_1'1 ∧ cons?(t_1'2)
      ⊃ Mem?(Mem(0) type_of)
  {2} every(λ (e: [Memory_Address_4G, list[Byte]]):
      OK?(linear_read_side_effect_in_page(e)(s')) ⊃
      data(linear_read_side_effect_in_page(e)(s')) = e'2
      (split(min_page, a', bl'))

```

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `split_linear_read_side_effects_data.1.1.3`.

`split_linear_read_side_effects_data.1.2`:

{-1}	$\text{every}(\lambda (a_2: \text{Address}, \text{bl2}: \text{list}[\text{Byte}]): \text{cons}^?(\text{bl2}))(\text{split}(\text{min_page}, a', \text{bl}')) \wedge$ $\text{every}(\lambda (a_2: \text{Address}, \text{bl2}: \text{list}[\text{Byte}]):$ $a' \leq a_2 \wedge a_2 + \text{length}(\text{bl2}) \leq a' + \text{length}(\text{bl}'))$ $(\text{split}(\text{min_page}, a', \text{bl}'))$ \supset $\text{every}(\lambda (t: [\text{Address}, \text{list}[\text{Byte}]]):$ $\text{cons}^?(t'2) \wedge$ $a' \leq t'1 \wedge t'1 + \text{length}(t'2) \leq a' + \text{length}(\text{bl}'))$ $(\text{split}(\text{min_page}, a', \text{bl}'))$
{-2}	$\text{every}(\lambda (a_2: \text{Address}, \text{bl2}: \text{list}[\text{Byte}]):$ $\text{rem}(\text{expt}(2, \text{min_page}))(\text{offset}(a_2)) + \text{length}(\text{bl2}) \leq \text{expt}(2, \text{min_page}))$ $(\text{split}(\text{min_page}, a', \text{bl}'))$
{-3}	$\text{every}(\lambda (t: [\text{Address}, \text{list}[\text{Byte}]]):$ $\text{type_of}(t'1) = \text{type_of}(a') \wedge$ $\text{reg_base}(\text{\#type_of} := \text{Mem_}, \text{offset} := 0\#)(\text{type_of}(a')) \leq t'1 \wedge$ $t'1 < \text{reg_size}(\text{\#type_of} := \text{Mem_}, \text{offset} := \text{max_linear_offset}\#)(\text{type_of}(a')))$ $(\text{split}(\text{min_page}, a', \text{bl}'))$
{-4}	$\text{every}(\lambda (a_2: \text{Address}, \text{bl2}: \text{list}[\text{Byte}]): \text{cons}^?(\text{bl2}))(\text{split}(\text{min_page}, a', \text{bl}'))$
{-5}	$\text{every}(\lambda (a_2: \text{Address}, \text{bl2}: \text{list}[\text{Byte}]):$ $a' \leq a_2 \wedge a_2 + \text{length}(\text{bl2}) \leq a' + \text{length}(\text{bl}'))$ $(\text{split}(\text{min_page}, a', \text{bl}'))$
{-6}	$\text{Mem}^?(a' \text{ type_of})$
{-7}	$0 \leq a' \text{ offset}$
{-8}	$a' \text{ offset} < \text{max_linear_offset}$
{-9}	$\text{every}(\lambda (x: \text{number}):$ $\text{number_field_pred}(x) \wedge$ $\text{real_pred}(x) \wedge$ $\text{rational_pred}(x) \wedge \text{integer_pred}(x) \wedge x \geq 0 \wedge x < \text{max_byte})$ (bl')
{-10}	$\text{cons}^?[\text{Byte}](\text{bl}')$
{-11}	$\text{is_linear_plain_memory}^?(\text{pm}')$
{-12}	$(\text{address_block}(a', \text{length}(\text{bl}')) \subseteq (\text{pm}' \text{ ro_addr} \cup \text{pm}' \text{ rw_addr}))$
{-13}	$\text{pm}' \text{ states}(s')$
{1}	$\forall (t: [\text{Address}, \text{list}[\text{Byte}]]):$ $\text{reg_base}(\text{Mem}(0))(\text{type_of}(a')) \leq t'1 \wedge \text{type_of}(t'1) = \text{type_of}(a') \supset$ $\text{Mem}^?(\text{Mem}(\text{max_linear_offset}) \text{ type_of})$
{2}	$\text{every}(\lambda (e: [\text{Memory_Address_4G}, \text{list}[\text{Byte}]]):$ $\text{OK}^?(\text{linear_read_side_effect_in_page}(e)(s')) \supset$ $\text{data}(\text{linear_read_side_effect_in_page}(e)(s')) = e'2)$ $(\text{split}(\text{min_page}, a', \text{bl}'))$

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `split_linear_read_side_effects_data.1.2`.

split_linear_read_side_effects_data.1.3:

{-1}	$\text{every}(\lambda (a_2: \text{Address}, \text{bl2}: \text{list}[\text{Byte}]): \text{cons}?(a_2))(\text{split}(\text{min_page}, a', \text{bl}')) \wedge$ $\text{every}(\lambda (a_2: \text{Address}, \text{bl2}: \text{list}[\text{Byte}]):$ $a' \leq a_2 \wedge a_2 + \text{length}(\text{bl2}) \leq a' + \text{length}(\text{bl}'))$ $(\text{split}(\text{min_page}, a', \text{bl}'))$ \supset $\text{every}(\lambda (t: [\text{Address}, \text{list}[\text{Byte}]]):$ $\text{cons}?(t'2) \wedge$ $a' \leq t'1 \wedge t'1 + \text{length}(t'2) \leq a' + \text{length}(\text{bl}'))$ $(\text{split}(\text{min_page}, a', \text{bl}'))$
{-2}	$\text{every}(\lambda (a_2: \text{Address}, \text{bl2}: \text{list}[\text{Byte}]):$ $\text{rem}(\text{expt}(2, \text{min_page}))(\text{offset}(a_2)) + \text{length}(\text{bl2}) \leq \text{expt}(2, \text{min_page}))$ $(\text{split}(\text{min_page}, a', \text{bl}'))$
{-3}	$\text{every}(\lambda (t: [\text{Address}, \text{list}[\text{Byte}]]):$ $\text{type_of}(t'1) = \text{type_of}(a') \wedge$ $\text{reg_base}(\text{\#type_of} := \text{Mem_}, \text{offset} := 0\#)(\text{type_of}(a')) \leq t'1 \wedge$ $t'1 < \text{reg_size}(\text{\#type_of} := \text{Mem_}, \text{offset} := \text{max_linear_offset\#})(\text{type_of}(a')))$ $(\text{split}(\text{min_page}, a', \text{bl}'))$
{-4}	$\text{every}(\lambda (a_2: \text{Address}, \text{bl2}: \text{list}[\text{Byte}]): \text{cons}?(a_2))(\text{split}(\text{min_page}, a', \text{bl}'))$
{-5}	$\text{every}(\lambda (a_2: \text{Address}, \text{bl2}: \text{list}[\text{Byte}]):$ $a' \leq a_2 \wedge a_2 + \text{length}(\text{bl2}) \leq a' + \text{length}(\text{bl}'))$ $(\text{split}(\text{min_page}, a', \text{bl}'))$
{-6}	Mem?(a' type_of)
{-7}	$0 \leq a' \text{offset}$
{-8}	$a' \text{offset} < \text{max_linear_offset}$
{-9}	$\text{every}(\lambda (x: \text{number}):$ $\text{number_field_pred}(x) \wedge$ $\text{real_pred}(x) \wedge$ $\text{rational_pred}(x) \wedge \text{integer_pred}(x) \wedge x \geq 0 \wedge x < \text{max_byte})$ (bl')
{-10}	cons?[Byte](bl')
{-11}	is_linear_plain_memory?(pm')
{-12}	$(\text{address_block}(a', \text{length}(\text{bl}')) \subseteq (\text{pm}'\text{ro_addr} \cup \text{pm}'\text{rw_addr}))$
{-13}	pm' states(s')
{1}	$\forall (t: [\text{Address}, \text{list}[\text{Byte}]]):$ $\text{type_of}(t'1) = \text{type_of}(a') \supset \text{Mem}?(a' \text{type_of})$
{2}	$\text{every}(\lambda (e: [\text{Memory_Address_4G}, \text{list}[\text{Byte}]]):$ $\text{OK}?(a' \text{type_of} \text{linear_read_side_effect_in_page}(e)(s')) \supset$ $\text{data}(a' \text{type_of} \text{linear_read_side_effect_in_page}(e)(s')) = e'2)$ $(\text{split}(\text{min_page}, a', \text{bl}'))$

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of split_linear_read_side_effects_data.1.3.

split_linear_read_side_effects_data.2:

<pre> {-1} every(λ (a2: Address, bl2: list[Byte]): rem(expt(2, min_page))(offset(a2)) + length(bl2) ≤ expt(2, min_page)) (split(min_page, a', bl')) {-2} every(λ (a2: Address, bl2: list[Byte]): cons?(bl2))(split(min_page, a', bl')) {-3} every(λ (a2: Address, bl2: list[Byte]): a' ≤ a2 ∧ a2 + length(bl2) ≤ a' + length(bl')) (split(min_page, a', bl')) {-4} Mem?(a' type_of) {-5} 0 ≤ a' offset {-6} a' offset < max_linear_offset {-7} every(λ (x: number): number_field_pred(x) ∧ real_pred(x) ∧ rational_pred(x) ∧ integer_pred(x) ∧ x ≥ 0 ∧ x < max_byte) (bl') {-8} cons?[Byte](bl') {-9} is_linear_plain_memory?(pm') {-10} (address_block(a', length(bl')) ⊆ (pm' ro_addr ∪ pm' rw_addr)) {-11} pm' states(s') </pre>	<hr style="border: 0.5px solid black;"/> <pre> {1} a' + length(bl') ≤ reg_size(#type_of := Mem_, offset := max_linear_offset#)(type_of(a')) {2} every(λ (e: [Memory_Address_4G, list[Byte]]): OK?(linear_read_side_effect_in_page(e)(s')) ⊃ data(linear_read_side_effect_in_page(e)(s')) = e'2) (split(min_page, a', bl')) </pre>
--	---

Hiding formulas: (-1 -2 -3 -7 2),

Expanding the definition of length,

Using lemma pm_memory_addr,

Simplifying, rewriting, and recording with decision procedures,

Hiding formulas: -6,

Expanding the definition of subset?,

Installing automatic rewrites from: (Mem! max_linear! bus_width!)

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of split_linear_read_side_effects_data.2.


```
split_linear_read_side_effects_data.3:
```

{-1}	every(λ (a_2 : Address, $bl2$: list[Byte]): rem(expt(2, min_page))(offset(a_2)) + length($bl2$) \leq expt(2, min_page)) (split(min_page, a' , bl'))
{-2}	every(λ (a_2 : Address, $bl2$: list[Byte]): cons?($bl2$))(split(min_page, a' , bl'))
{-3}	every(λ (a_2 : Address, $bl2$: list[Byte]): $a' \leq a_2 \wedge a_2 + \text{length}(bl2) \leq a' + \text{length}(bl')$ (split(min_page, a' , bl'))
{-4}	Mem?(a' 'type_of)
{-5}	$0 \leq a'$ 'offset
{-6}	a' 'offset < max_linear_offset
{-7}	every(λ (x : number): number_field_pred(x) \wedge real_pred(x) \wedge rational_pred(x) \wedge integer_pred(x) \wedge $x \geq 0 \wedge x < \text{max_byte}$) (bl')
{-8}	cons?[Byte](bl')
{-9}	is_linear_plain_memory?(pm')
{-10}	(address_block(a' , length(bl')) \subseteq (pm' 'ro_addr \cup pm' 'rw_addr))
{-11}	pm' 'states(s')
{1}	reg_base((#type_of := Mem_, offset := 0#))(type_of(a')) $\leq a'$
{2}	every(λ (e : [Memory_Address_4G, list[Byte]]): OK?(linear_read_side_effect_in_page(e)(s')) \supset data(linear_read_side_effect_in_page(e)(s')) = e' 2) (split(min_page, a' , bl'))

Expanding the definition of reg_base,

Expanding the definition of \leq ,

which is trivially true.

This completes the proof of split_linear_read_side_effects_data.3.

Q.E.D.

C.117.81

Linear_Memory_Properties.split_linear_write_side_effects_data

Terse proof for split_linear_write_side_effects_data.

```
split_linear_write_side_effects_data:
```

{1}	\forall (pm : Plain_Memory[Linear_memory], s : Linear_memory[Physical_memory, pm_phy], a : Memory_Address_4G, bl : (cons?[Byte])): is_linear_plain_memory?(pm) \wedge (address_block(a , length(bl)) \subseteq pm' 'rw_addr) \wedge pm' 'states(s) \supset every(λ (e : [Memory_Address_4G, list[Byte]]): OK?(linear_write_side_effect_in_page(e)(s)) \supset data(linear_write_side_effect_in_page(e)(s)) = e' 2) (split(min_page, a , bl))
-----	---

Repeatedly Skolemizing and flattening,

Installing automatic rewrites from: Mem!! max_linear!! min_linear!!

Using lemma split_range,

Using lemma split_no_null,

Using lemma split_type,

Using lemma split_pair_cross_size,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 3 subgoals:

split_linear_write_side_effects_data.1:

{-1}	every(λ (a_2 : Address, $bl2$: list[Byte]): rem(expt(2, min_page))(offset(a_2)) + length($bl2$) \leq expt(2, min_page)) (split(min_page, a' , bl'))
{-2}	every(λ (t : [Address, list[Byte]]): type_of(t '1) = type_of(a') \wedge reg_base((#type_of := Mem_, offset := 0#))(type_of(a')) \leq t '1 \wedge t '1 < reg_size((#type_of := Mem_, offset := max_linear_offset#))(type_of(a'))) (split(min_page, a' , bl'))
{-3}	every(λ (a_2 : Address, $bl2$: list[Byte]): cons?($bl2$))(split(min_page, a' , bl'))
{-4}	every(λ (a_2 : Address, $bl2$: list[Byte]): $a' \leq a_2 \wedge a_2 + \text{length}(bl2) \leq a' + \text{length}(bl')$ (split(min_page, a' , bl'))
{-5}	Mem?(a' 'type_of)
{-6}	$0 \leq a'$ 'offset
{-7}	a' 'offset < max_linear_offset
{-8}	every(λ (x : number): number_field_pred(x) \wedge real_pred(x) \wedge rational_pred(x) \wedge integer_pred(x) \wedge $x \geq 0 \wedge x < \text{max_byte}$) (bl')
{-9}	cons?[Byte](bl')
{-10}	is_linear_plain_memory?(pm')
{-11}	(address_block(a' , length(bl')) \subseteq pm' 'rw_addr)
{-12}	pm' 'states(s')
{1}	every(λ (e : [Memory_Address_4G, list[Byte]]): OK?(linear_write_side_effect_in_page(e)(s')) \supset data(linear_write_side_effect_in_page(e)(s')) = e '2) (split(min_page, a' , bl'))

Using lemma every_conjunct_left,

Using lemma every_conjunct_left,

we get 3 subgoals:

```
split_linear_write_side_effects_data.1.1:
```

```

{-1} every(λ (t: [Address, list[Byte]]):
      cons?(t'2) ∧
      a' ≤ t'1 ∧ t'1 + length(t'2) ≤ a' + length(bl')
      (split(min_page, a', bl'))
    ∧
    every(λ (t: [Address, list[Byte]]):
      type_of(t'1) = type_of(a') ∧
      reg_base(min_linear)(type_of(a')) ≤ t'1 ∧
      t'1 < reg_size(max_linear)(type_of(a'))
      (split(min_page, a', bl'))
    ⊃
    every(λ (t_1: [Address, list[Byte]]):
      (cons?(t_1'2) ∧
      a' ≤ t_1'1 ∧ t_1'1 + length(t_1'2) ≤ a' + length(bl'))
      ∧
      type_of(t_1'1) = type_of(a') ∧
      reg_base(min_linear)(type_of(a')) ≤ t_1'1 ∧
      t_1'1 < reg_size(max_linear)(type_of(a'))
      (split(min_page, a', bl'))
    {-2} every(λ (a_2: Address, bl2: list[Byte]): cons?(bl2))(split(min_page, a', bl')) ∧
    every(λ (a_2: Address, bl2: list[Byte]):
      a' ≤ a_2 ∧ a_2 + length(bl2) ≤ a' + length(bl')
      (split(min_page, a', bl'))
    ⊃
    every(λ (t: [Address, list[Byte]]):
      cons?(t'2) ∧
      a' ≤ t'1 ∧ t'1 + length(t'2) ≤ a' + length(bl')
      (split(min_page, a', bl'))
    {-3} every(λ (a_2: Address, bl2: list[Byte]):
      rem(expt(2, min_page))(offset(a_2)) + length(bl2) ≤ expt(2, min_page))
      (split(min_page, a', bl'))
    {-4} every(λ (t: [Address, list[Byte]]):
      type_of(t'1) = type_of(a') ∧
      reg_base((#type_of := Mem_, offset := 0#))(type_of(a')) ≤ t'1 ∧
      t'1 < reg_size((#type_of := Mem_, offset := max_linear_offset#))(type_of(a'))
      (split(min_page, a', bl'))
    {-5} every(λ (a_2: Address, bl2: list[Byte]): cons?(bl2))(split(min_page, a', bl'))
    {-6} every(λ (a_2: Address, bl2: list[Byte]):
      a' ≤ a_2 ∧ a_2 + length(bl2) ≤ a' + length(bl')
      (split(min_page, a', bl'))
    {-7} Mem?(a' type_of)
    {-8} 0 ≤ a' offset
    {-9} a' offset < max_linear_offset
    {-10} every(λ (x: number):
      number_field_pred(x) ∧
      real_pred(x) ∧
      rational_pred(x) ∧ integer_pred(x) ∧ x ≥ 0 ∧ x < max_byte)
      (bl')
    {-11} cons?[Byte](bl')
    {-12} is_linear_plain_memory?(pm')
    {-13} (address_block(a', length(bl')) ⊆ pm' rw_addr)
    {-14} pm' states(s')
  }
  {1} every(λ (e: [Memory_Address_4G, list[Byte]]):
    OK?(linear_write_side_effect_in_page(e)(s')) ⊃
    data(linear_write_side_effect_in_page(e)(s')) = e'2
    (split(min_page, a', bl'))

```

C.117 Proofs for Linear_Memory_Properties (challenge-linear.pvs)

Using lemma `every_conjunct_left`,

we get 3 subgoals:

split_linear_write_side_effects_data.1.1.1:

```

{-1} every(λ (t_1: [Address, list[Byte]]):
  cons?(t_1'2) ∧
  a' ≤ t_1'1 ∧
  t_1'1 + length(t_1'2) ≤ a' + length(bl') ∧
  type_of(t_1'1) = type_of(a') ∧
  reg_base(min_linear)(type_of(a')) ≤ t_1'1 ∧
  t_1'1 < reg_size(max_linear)(type_of(a')))
  (split(min_page, a', bl'))
  ∧
  every(λ (a_2: Address, bl2: list[Byte]):
    rem(expt(2, min_page))(offset(a_2)) + length(bl2) ≤
    expt(2, min_page))
    (split(min_page, a', bl'))
  ⊃
  every(λ (t: [Address, list[Byte]]):
    (cons?(t'2) ∧
    a' ≤ t'1 ∧
    t'1 + length(t'2) ≤ a' + length(bl') ∧
    type_of(t'1) = type_of(a') ∧
    reg_base(min_linear)(type_of(a')) ≤ t'1 ∧
    t'1 < reg_size(max_linear)(type_of(a')))
    ∧
    rem(expt(2, min_page))(offset(t'1)) + length(t'2) ≤
    expt(2, min_page))
    (split(min_page, a', bl'))
{-2} every(λ (t: [Address, list[Byte]]):
  cons?(t'2) ∧
  a' ≤ t'1 ∧ t'1 + length(t'2) ≤ a' + length(bl'))
  (split(min_page, a', bl'))
  ∧
  every(λ (t: [Address, list[Byte]]):
    type_of(t'1) = type_of(a') ∧
    reg_base(min_linear)(type_of(a')) ≤ t'1 ∧
    t'1 < reg_size(max_linear)(type_of(a')))
    (split(min_page, a', bl'))
  ⊃
  every(λ (t_1: [Address, list[Byte]]):
    (cons?(t_1'2) ∧
    a' ≤ t_1'1 ∧ t_1'1 + length(t_1'2) ≤ a' + length(bl'))
    ∧
    type_of(t_1'1) = type_of(a') ∧
    reg_base(min_linear)(type_of(a')) ≤ t_1'1 ∧
    t_1'1 < reg_size(max_linear)(type_of(a')))
    (split(min_page, a', bl'))
{-3} every(λ (a_2: Address, bl2: list[Byte]): cons?(bl2))(split(min_page, a', bl')) ∧
  every(λ (a_2: Address, bl2: list[Byte]):
    a' ≤ a_2 ∧ a_2 + length(bl2) ≤ a' + length(bl'))
    (split(min_page, a', bl'))
  ⊃
  every(λ (t: [Address, list[Byte]]):
    cons?(t'2) ∧
    a' ≤ t'1 ∧ t'1 + length(t'2) ≤ a' + length(bl'))
    (split(min_page, a', bl'))
{-4} every(λ (a_2: Address, bl2: list[Byte]):
  rem(expt(2, min_page))(offset(a_2)) + length(bl2) ≤ expt(2, min_page))
  (split(min_page, a', bl'))
{-5} every(λ (t: [Address, list[Byte]]):
  type_of(t'1) = type_of(a') ∧
  reg_base((#type_of := Mem_, offset := 0#))(type_of(a')) ≤ t'1 ∧
  t'1 < reg_size((#type_of := Mem_, offset := max_linear_offset#))(type_of(a')))
  (split(min_page, a', bl'))
{-6} every(λ (a_2: Address, bl2: list[Byte]): cons?(bl2))(split(min_page, a', bl'))
{-7} every(λ (a_2: Address, bl2: list[Byte]):

```

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Hiding formulas: (-1 -3 -4 -5 -6 -7 -11),

Installing automatic rewrites from: (##! expr_2_super! expr_2_super_res!)

Using lemma every_extend[[Address, list[Byte]], [Memory_Address_4G, list[Byte]]],

we get 3 subgoals:

split_linear_write_side_effects_data.1.1.1.1.1:

{-1}	<pre> every(λ (t_1: [Address, list[Byte]]): cons?(t_1'2) ∧ a' ≤ t_1'1 ∧ t_1'1 + length(t_1'2) ≤ a' + length(bl') ∧ type_of(t_1'1) = type_of(a') ∧ reg_base(min_linear)(type_of(a')) ≤ t_1'1 ∧ t_1'1 < reg_size(max_linear)(type_of(a')) ∧ rem(expt(2, min_page))(offset(t_1'1)) + length(t_1'2) ≤ expt(2, min_page) (split(min_page, a', bl')) ⊃ every(λ (s: [Memory_Address_4G, list[Byte]]): cons?(s'2) ∧ a' ≤ s'1 ∧ s'1 + length(s'2) ≤ a' + length(bl') ∧ type_of(s'1) = type_of(a') ∧ reg_base(min_linear)(type_of(a')) ≤ s'1 ∧ s'1 < reg_size(max_linear)(type_of(a')) ∧ rem(expt(2, min_page))(offset(s'1)) + length(s'2) ≤ expt(2, min_page) (split(min_page, a', bl'))) </pre>
{-2}	<pre> every(λ (t: [Address, list[Byte]]): (cons?(t'2) ∧ a' ≤ t'1 ∧ t'1 + length(t'2) ≤ a' + length(bl') ∧ type_of(t'1) = type_of(a') ∧ reg_base((#type_of := Mem_, offset := 0#))(type_of(a')) ≤ t'1 ∧ t'1 < reg_size((#type_of := Mem_, offset := max_linear_offset#))(type_of(a'))) ∧ rem(expt(2, min_page))(offset(t'1)) + length(t'2) ≤ expt(2, min_page) (split(min_page, a', bl')) </pre>
{-3}	Mem?(a' type_of)
{-4}	0 ≤ a' offset
{-5}	a' offset < max_linear_offset
{-6}	cons?[Byte](bl')
{-7}	is_linear_plain_memory?(pm')
{-8}	(address_block(a', length(bl')) ⊆ pm'rw_addr)
{-9}	pm' states(s')
{1}	<pre> every(λ (e: [Memory_Address_4G, list[Byte]]): OK?(linear_write_side_effect_in_page(e)(s')) ⊃ data(linear_write_side_effect_in_page(e)(s')) = e'2 (split(min_page, a', bl')) </pre>

Using lemma every_implied,

we get 3 subgoals:

split_linear_write_side_effects_data.1.1.1.1.1:

```

{-1}  (∀ (t: [Memory_Address_4G, list[Byte]]):
      cons?(t'2) ∧
      a' ≤ t'1 ∧
      t'1 + length(t'2) ≤ a' + length(bl') ∧
      type_of(t'1) = type_of(a') ∧
      reg_base(min_linear)(type_of(a')) ≤ t'1 ∧
      t'1 < reg_size(max_linear)(type_of(a')) ∧
      rem(expt(2, min_page))(offset(t'1)) + length(t'2) ≤
      expt(2, min_page)
      ⊃
      OK?(linear_write_side_effect_in_page(t)(s')) ⊃
      data(linear_write_side_effect_in_page(t)(s')) = t'2)
      ∧
      every(λ (s: [Memory_Address_4G, list[Byte]]):
        cons?(s'2) ∧
        a' ≤ s'1 ∧
        s'1 + length(s'2) ≤ a' + length(bl') ∧
        type_of(s'1) = type_of(a') ∧
        reg_base(min_linear)(type_of(a')) ≤ s'1 ∧
        s'1 < reg_size(max_linear)(type_of(a')) ∧
        rem(expt(2, min_page))(offset(s'1)) + length(s'2) ≤
        expt(2, min_page))
        (split(min_page, a', bl'))
      ⊃
      every(λ (e: [Memory_Address_4G, list[Byte]]):
        OK?(linear_write_side_effect_in_page(e)(s')) ⊃
        data(linear_write_side_effect_in_page(e)(s')) = e'2)
        (split(min_page, a', bl'))
{-2}  every(λ (t_1: [Address, list[Byte]]):
      cons?(t_1'2) ∧
      a' ≤ t_1'1 ∧
      t_1'1 + length(t_1'2) ≤ a' + length(bl') ∧
      type_of(t_1'1) = type_of(a') ∧
      reg_base(min_linear)(type_of(a')) ≤ t_1'1 ∧
      t_1'1 < reg_size(max_linear)(type_of(a')) ∧
      rem(expt(2, min_page))(offset(t_1'1)) + length(t_1'2) ≤
      expt(2, min_page))
      (split(min_page, a', bl'))
      ⊃
      every(λ (s: [Memory_Address_4G, list[Byte]]):
        cons?(s'2) ∧
        a' ≤ s'1 ∧
        s'1 + length(s'2) ≤ a' + length(bl') ∧
        type_of(s'1) = type_of(a') ∧
        reg_base(min_linear)(type_of(a')) ≤ s'1 ∧
        s'1 < reg_size(max_linear)(type_of(a')) ∧
        rem(expt(2, min_page))(offset(s'1)) + length(s'2) ≤
        expt(2, min_page))
        (split(min_page, a', bl'))
{-3}  every(λ (t: [Address, list[Byte]]):
      (cons?(t'2) ∧
      a' ≤ t'1 ∧
      t'1 + length(t'2) ≤ a' + length(bl') ∧
      type_of(t'1) = type_of(a') ∧
      reg_base((#type_of := Mem_, offset := 0#))(type_of(a')) ≤ t'1 ∧
      t'1 <
      reg_size((#type_of := Mem_, offset := max_linear_offset#))(type_of(a')))
      ∧
      rem(expt(2, min_page))(offset(t'1)) + length(t'2) ≤
      expt(2, min_page))
      (split(min_page, a', bl'))
{-4}  Mem?(a' type_of)
{-5}  0 ≤ a' offset

```

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Hiding formulas: (-1 -2 2),

Expanding the definition of linear_write_side_effect_in_page,

Repeatedly Skolemizing and flattening,

Case splitting on pm!1'rw_addr(t!1'1),

we get 2 subgoals:

split_linear_write_side_effects_data.1.1.1.1.1.1:

{-1}	pm'rw_addr(t'1)
{-2}	cons?(t'2)
{-3}	$a' \leq t'1$
{-4}	$t'1 + \text{length}(t'2) \leq a' + \text{length}(bl')$
{-5}	$\text{type_of}(t'1) = \text{type_of}(a')$
{-6}	$\text{reg_base}(\text{(#type_of := Mem_}, \text{offset := 0\#})(\text{type_of}(a')) \leq t'1$
{-7}	$t'1 < \text{reg_size}(\text{(#type_of := Mem_}, \text{offset := max_linear_offset\#})(\text{type_of}(a'))$
{-8}	$\text{rem}(\text{expt}(2, \text{min_page}))(\text{offset}(t'1) + \text{length}(t'2)) \leq \text{expt}(2, \text{min_page})$
{-9}	OK?((linear_resolve(t'1, Write) ## $(\lambda (\text{pa: Memory_Address_4G})(\text{ns: Linear_memory}[\text{Physical_memory}, \text{pm_phy}]):$ $\text{memory_write_side_effect}(\text{pm_phy}'\text{mem})(\text{pa}, t'2, \text{TRUE})(s'))$ $(s'))$
{-10}	Mem?(a'type_of)
{-11}	$0 \leq a'\text{offset}$
{-12}	$a'\text{offset} < \text{max_linear_offset}$
{-13}	cons?[Byte](bl')
{-14}	is_linear_plain_memory?(pm')
{-15}	$(\text{address_block}(a', \text{length}(bl'))) \subseteq \text{pm}'\text{rw_addr}$
{-16}	pm'states(s')
{1}	data((linear_resolve(t'1, Write) ## $(\lambda (\text{pa: Memory_Address_4G})(\text{ns: Linear_memory}[\text{Physical_memory}, \text{pm_phy}]):$ $\text{memory_write_side_effect}(\text{pm_phy}'\text{mem})(\text{pa}, t'2, \text{TRUE})(s'))$ $(s'))$ $= t'2$

Using lemma pm_linear_resolve_write_ok,

Case splitting on subset?(address_block(data(linear_resolve(t!1'1, Write)(s!1)), length(t!1'2)), pm_phy'rw_addr),

we get 3 subgoals:

split_linear_write_side_effects_data.1.1.1.1.1.1.1:

```

{-1}  (address_block(data(linear_resolve(t'1, Write)(s')), length(t'2)) ⊆ pm_phy'rw_addr)
{-2}  is_linear_plain_memory?(pm') ⊃ OK?(linear_resolve(t'1, Write)(s'))
{-3}  pm'rw_addr(t'1)
{-4}  cons?(t'2)
{-5}  a' ≤ t'1
{-6}  t'1 + length(t'2) ≤ a' + length(bl')
{-7}  type_of(t'1) = type_of(a')
{-8}  reg_base((#type_of := Mem_, offset := 0#))(type_of(a')) ≤ t'1
{-9}  t'1 < reg_size((#type_of := Mem_, offset := max_linear_offset#))(type_of(a'))
{-10} rem(expt(2, min_page))(offset(t'1)) + length(t'2) ≤ expt(2, min_page)
{-11} OK?((linear_resolve(t'1, Write) ##
          (λ (pa: Memory_Address_4G)(ns: Linear_memory[Physical_memory, pm_phy]):
            memory_write_side_effect(pm_phy'mem)(pa, t'2, TRUE)(s')))
          (s'))
{-12} Mem?(a'type_of)
{-13} 0 ≤ a'offset
{-14} a'offset < max_linear_offset
{-15} cons?[Byte](bl')
{-16} is_linear_plain_memory?(pm')
{-17} (address_block(a', length(bl')) ⊆ pm'rw_addr)
{-18} pm'states(s')
-----
{1}  data((linear_resolve(t'1, Write) ##
          (λ (pa: Memory_Address_4G)(ns: Linear_memory[Physical_memory, pm_phy]):
            memory_write_side_effect(pm_phy'mem)(pa, t'2, TRUE)(s')))
          (s'))
      = t'2

```

Using lemma linear_resolve_states,

Using lemma pm_states,

Using lemma pm_plain_phy,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 2 subgoals:

split_linear_write_side_effects_data.1.1.1.1.1.1.1.1:

{-1}	is_linear_plain_memory?(pm')
{-2}	plain_memory?(pm_phy)
{-3}	pm' states = pm_phy states
{-4}	pm' states(s')
{-5}	OK?(linear_resolve(t'1, Write)(s'))
{-6}	pm_phy states(state(linear_resolve(t'1, Write)(s')))
{-7}	(address_block(data(linear_resolve(t'1, Write)(s')), length(t'2)) \subseteq pm_phy rw_addr)
{-8}	pm' rw_addr(t'1)
{-9}	cons?(t'2)
{-10}	$a' \leq t'1$
{-11}	$t'1 + \text{length}(t'2) \leq a' + \text{length}(bl')$
{-12}	type_of(t'1) = type_of(a')
{-13}	reg_base((#type_of := Mem_, offset := 0#))(type_of(a')) $\leq t'1$
{-14}	$t'1 < \text{reg_size}((\#type_of := \text{Mem_}, \text{offset} := \text{max_linear_offset\#})(\text{type_of}(a')))$
{-15}	$\text{rem}(\text{expt}(2, \text{min_page}))(\text{offset}(t'1) + \text{length}(t'2)) \leq \text{expt}(2, \text{min_page})$
{-16}	OK?(memory_write_side_effect(pm_phy mem (data(linear_resolve(t'1, Write)(s')), t'2, TRUE)(s'))
{-17}	Mem?(a' type_of)
{-18}	$0 \leq a' \text{ offset}$
{-19}	$a' \text{ offset} < \text{max_linear_offset}$
{-20}	cons?[Byte](bl')
{-21}	(address_block(a', length(bl')) \subseteq pm' rw_addr)
{1}	data(memory_write_side_effect(pm_phy mem (data(linear_resolve(t'1, Write)(s')), t'2, TRUE)(s')) = t'2

Expanding the definition of plain_memory?,

Applying disjunctive simplification to flatten sequent,

Hiding formulas: (-2 -3 -4 -5 -6 -7),

Expanding the definition of side_effect_content_unchanged,

Instantiating quantified variables,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of split_linear_write_side_effects_data.1.1.1.1.1.1.1.1.

split_linear_write_side_effects_data.1.1.1.1.1.1.1.2:

{-1}	is_linear_plain_memory?(pm')
{-2}	plain_memory?(pm_phy)
{-3}	pm' states = pm_phy states
{-4}	pm' states(s')
{-5}	OK?(linear_resolve(t'1, Write)(s'))
{-6}	(address_block(data(linear_resolve(t'1, Write)(s')), length(t'2)) \subseteq pm_phy rw_addr)
{-7}	pm' rw_addr(t'1)
{-8}	cons?(t'2)
{-9}	$a' \leq t'1$
{-10}	$t'1 + \text{length}(t'2) \leq a' + \text{length}(bl')$
{-11}	type_of(t'1) = type_of(a')
{-12}	reg_base((#type_of := Mem_, offset := 0#))(type_of(a')) $\leq t'1$
{-13}	$t'1 < \text{reg_size}((\#type_of := \text{Mem_}, \text{offset} := \text{max_linear_offset\#}))(type_of(a'))$
{-14}	rem(expt(2, min_page))(offset(t'1)) + length(t'2) \leq expt(2, min_page)
{-15}	OK?(memory_write_side_effect(pm_phy mem) (data(linear_resolve(t'1, Write)(s')), t'2, TRUE)(s'))
{-16}	Mem?(a' type_of)
{-17}	$0 \leq a' \text{ offset}$
{-18}	$a' \text{ offset} < \text{max_linear_offset}$
{-19}	cons?[Byte](bl')
{-20}	(address_block(a', length(bl')) \subseteq pm' rw_addr)
{1}	union(pm' ro_addr, pm' rw_addr)(t'1)
{2}	data(memory_write_side_effect(pm_phy mem) (data(linear_resolve(t'1, Write)(s')), t'2, TRUE)(s')) = t'2

Keeping (-7 1) and hiding *,

Rewriting using union_right, matching in *,

This completes the proof of split_linear_write_side_effects_data.1.1.1.1.1.1.1.2.

split_linear_write_side_effects_data.1.1.1.1.1.1.2:

{-1}	is_linear_plain_memory?(pm') \supset OK?(linear_resolve(t'1, Write)(s'))
{-2}	pm'rw_addr(t'1)
{-3}	cons?(t'2)
{-4}	$a' \leq t'1$
{-5}	$t'1 + \text{length}(t'2) \leq a' + \text{length}(bl')$
{-6}	$\text{type_of}(t'1) = \text{type_of}(a')$
{-7}	$\text{reg_base}(\text{\#type_of := Mem_}, \text{offset := 0\#})(\text{type_of}(a')) \leq t'1$
{-8}	$t'1 < \text{reg_size}(\text{\#type_of := Mem_}, \text{offset := max_linear_offset\#})(\text{type_of}(a'))$
{-9}	$\text{rem}(\text{expt}(2, \text{min_page}))(\text{offset}(t'1)) + \text{length}(t'2) \leq \text{expt}(2, \text{min_page})$
{-10}	OK?(linear_resolve(t'1, Write) ## $(\lambda (\text{pa: Memory_Address_4G})(\text{ns: Linear_memory}[\text{Physical_memory}, \text{pm_phy}]):$ $\text{memory_write_side_effect}(\text{pm_phy}'\text{mem})(\text{pa}, t'2, \text{TRUE})(s'))$ $(s'))$
{-11}	Mem?(a'type_of)
{-12}	$0 \leq a'\text{offset}$
{-13}	$a'\text{offset} < \text{max_linear_offset}$
{-14}	cons?[Byte](bl')
{-15}	is_linear_plain_memory?(pm')
{-16}	$(\text{address_block}(a', \text{length}(bl'))) \subseteq \text{pm}'\text{rw_addr}$
{-17}	pm'states(s')
{1}	$(\text{address_block}(\text{data}(\text{linear_resolve}(t'1, \text{Write})(s')), \text{length}(t'2)) \subseteq \text{pm_phy}'\text{rw_addr})$
{2}	data((linear_resolve(t'1, Write) ## $(\lambda (\text{pa: Memory_Address_4G})(\text{ns: Linear_memory}[\text{Physical_memory}, \text{pm_phy}]):$ $\text{memory_write_side_effect}(\text{pm_phy}'\text{mem})(\text{pa}, t'2, \text{TRUE})(s'))$ $(s'))$ $= t'2$

Hiding formulas: 2,

Using lemma same_page_address_block_write,

we get 3 subgoals:

split_linear_write_side_effects_data.1.1.1.1.1.1.2.1:

{-1}	is_linear_plain_memory?(pm') \wedge pm' 'states(s') \wedge pm' 'rw_addr(t' '1) \wedge a' \leq t' '1 \wedge t' '1 + length(t' '2) \leq a' + length(bl') \wedge rem(expt(2, min_page))(offset(t' '1)) + length(t' '2) \leq expt(2, min_page) \wedge (address_block(a', length(bl')) \subseteq pm' 'rw_addr)
	\supset (address_block(data(linear_resolve(t' '1, Write)(s')), length(t' '2)) \subseteq pm_phy'rw_addr)
{-2}	is_linear_plain_memory?(pm') \supset OK?(linear_resolve(t' '1, Write)(s'))
{-3}	pm' 'rw_addr(t' '1)
{-4}	cons?(t' '2)
{-5}	a' \leq t' '1
{-6}	t' '1 + length(t' '2) \leq a' + length(bl')
{-7}	type_of(t' '1) = type_of(a')
{-8}	reg_base((#type_of := Mem_, offset := 0#))(type_of(a')) \leq t' '1
{-9}	t' '1 < reg_size((#type_of := Mem_, offset := max_linear_offset#))(type_of(a'))
{-10}	rem(expt(2, min_page))(offset(t' '1)) + length(t' '2) \leq expt(2, min_page)
{-11}	OK?((linear_resolve(t' '1, Write) ## (λ (pa: Memory_Address_4G)(ns: Linear_memory[Physical_memory, pm_phy]): memory_write_side_effect(pm_phy' mem)(pa, t' '2, TRUE)(s')) (s'))
{-12}	Mem?(a' 'type_of)
{-13}	0 \leq a' 'offset
{-14}	a' 'offset < max_linear_offset
{-15}	cons?[Byte](bl')
{-16}	is_linear_plain_memory?(pm')
{-17}	(address_block(a', length(bl')) \subseteq pm' 'rw_addr)
{-18}	pm' 'states(s')
{1}	(address_block(data(linear_resolve(t' '1, Write)(s')), length(t' '2)) \subseteq pm_phy'rw_addr)

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of split_linear_write_side_effects_data.1.1.1.1.1.1.2.1.

split_linear_write_side_effects_data.1.1.1.1.1.1.2.2:

{-1}	is_linear_plain_memory?(pm') \supset OK?(linear_resolve(t'1, Write)(s'))
{-2}	pm'rw_addr(t'1)
{-3}	cons?(t'2)
{-4}	$a' \leq t'1$
{-5}	$t'1 + \text{length}(t'2) \leq a' + \text{length}(bl')$
{-6}	type_of(t'1) = type_of(a')
{-7}	reg_base((#type_of := Mem_, offset := 0#))(type_of(a')) $\leq t'1$
{-8}	$t'1 < \text{reg_size}((\#type_of := \text{Mem_}, \text{offset} := \text{max_linear_offset\#}))(type_of(a'))$
{-9}	$\text{rem}(\text{expt}(2, \text{min_page}))(\text{offset}(t'1)) + \text{length}(t'2) \leq \text{expt}(2, \text{min_page})$
{-10}	OK?((linear_resolve(t'1, Write) ## $(\lambda (\text{pa}: \text{Memory_Address_4G})(\text{ns}: \text{Linear_memory}[\text{Physical_memory}, \text{pm_phy}]):$ $\text{memory_write_side_effect}(\text{pm_phy}'\text{mem})(\text{pa}, t'2, \text{TRUE})(s'))$ $(s'))$
{-11}	Mem?(a'type_of)
{-12}	$0 \leq a'\text{offset}$
{-13}	$a'\text{offset} < \text{max_linear_offset}$
{-14}	cons?[Byte](bl')
{-15}	is_linear_plain_memory?(pm')
{-16}	(address_block(a', length(bl'))) \subseteq pm'rw_addr
{-17}	pm'states(s')
{1}	length[Byte](t'2) > 0
{2}	(address_block(data(linear_resolve(t'1, Write)(s')), length(t'2))) \subseteq pm_phy'rw_addr

Expanding the definition of length,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of split_linear_write_side_effects_data.1.1.1.1.1.1.2.2.

split_linear_write_side_effects_data.1.1.1.1.1.1.2.3:

{-1}	is_linear_plain_memory?(pm') \supset OK?(linear_resolve(t'1, Write)(s'))
{-2}	pm'rw_addr(t'1)
{-3}	cons?(t'2)
{-4}	$a' \leq t'1$
{-5}	$t'1 + \text{length}(t'2) \leq a' + \text{length}(bl')$
{-6}	type_of(t'1) = type_of(a')
{-7}	reg_base((#type_of := Mem_, offset := 0#))(type_of(a')) $\leq t'1$
{-8}	$t'1 < \text{reg_size}((\#type_of := \text{Mem_}, \text{offset} := \text{max_linear_offset\#}))(type_of(a'))$
{-9}	$\text{rem}(\text{expt}(2, \text{min_page}))(\text{offset}(t'1)) + \text{length}(t'2) \leq \text{expt}(2, \text{min_page})$
{-10}	OK?((linear_resolve(t'1, Write) ## $(\lambda (\text{pa}: \text{Memory_Address_4G})(\text{ns}: \text{Linear_memory}[\text{Physical_memory}, \text{pm_phy}]):$ $\text{memory_write_side_effect}(\text{pm_phy}'\text{mem})(\text{pa}, t'2, \text{TRUE})(s'))$ $(s'))$
{-11}	Mem?(a'type_of)
{-12}	$0 \leq a'\text{offset}$
{-13}	$a'\text{offset} < \text{max_linear_offset}$
{-14}	cons?[Byte](bl')
{-15}	is_linear_plain_memory?(pm')
{-16}	(address_block(a', length(bl'))) \subseteq pm'rw_addr
{-17}	pm'states(s')
{1}	length[Byte](bl') > 0
{2}	(address_block(data(linear_resolve(t'1, Write)(s')), length(t'2))) \subseteq pm_phy'rw_addr

Expanding the definition of length,

C Proof scripts

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `split_linear_write_side_effects_data.1.1.1.1.1.1.2.3`.

`split_linear_write_side_effects_data.1.1.1.1.1.1.3`:

{-1}	<code>is_linear_plain_memory?(pm') \supset OK?(linear_resolve(t'1, Write)(s'))</code>
{-2}	<code>pm'rw_addr(t'1)</code>
{-3}	<code>cons?(t'2)</code>
{-4}	<code>a' \leq t'1</code>
{-5}	<code>t'1 + length(t'2) \leq a' + length(bl')</code>
{-6}	<code>type_of(t'1) = type_of(a')</code>
{-7}	<code>reg_base((#type_of := Mem_, offset := 0#))(type_of(a')) \leq t'1</code>
{-8}	<code>t'1 < reg_size((#type_of := Mem_, offset := max_linear_offset#))(type_of(a'))</code>
{-9}	<code>rem(expt(2, min_page))(offset(t'1)) + length(t'2) \leq expt(2, min_page)</code>
{-10}	<code>OK?(linear_resolve(t'1, Write) ## $(\lambda$ (pa: Memory_Address_4G)(ns: Linear_memory[Physical_memory, pm_phy]): memory_write_side_effect(pm_phy' mem)(pa, t'2, TRUE)(s')) (s'))</code>
{-11}	<code>Mem?(a' type_of)</code>
{-12}	<code>0 \leq a' offset</code>
{-13}	<code>a' offset < max_linear_offset</code>
{-14}	<code>cons?[Byte](bl')</code>
{-15}	<code>is_linear_plain_memory?(pm')</code>
{-16}	<code>(address_block(a', length(bl')) \subseteq pm'rw_addr)</code>
{-17}	<code>pm' states(s')</code>
{1}	<code>OK?[Physical_memory, Address](linear_resolve[Physical_memory, pm_phy](t'1, Write)(s'))</code>
{2}	<code>data((linear_resolve(t'1, Write) ## $(\lambda$ (pa: Memory_Address_4G)(ns: Linear_memory[Physical_memory, pm_phy]): memory_write_side_effect(pm_phy' mem)(pa, t'2, TRUE)(s')) (s')) = t'2</code>

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `split_linear_write_side_effects_data.1.1.1.1.1.1.3`.

split_linear_write_side_effects_data.1.1.1.1.1.2:

{-1}	cons?(t'2)
{-2}	a' ≤ t'1
{-3}	t'1 + length(t'2) ≤ a' + length(bl')
{-4}	type_of(t'1) = type_of(a')
{-5}	reg_base((#type_of := Mem_, offset := 0#))(type_of(a')) ≤ t'1
{-6}	t'1 < reg_size((#type_of := Mem_, offset := max_linear_offset#))(type_of(a'))
{-7}	rem(expt(2, min_page))(offset(t'1) + length(t'2)) ≤ expt(2, min_page)
{-8}	OK?(linear_resolve(t'1, Write) ## (λ (pa: Memory_Address_4G)(ns: Linear_memory[Physical_memory, pm_phy]): memory_write_side_effect(pm_phy'mem)(pa, t'2, TRUE)(s')) (s'))
{-9}	Mem?(a' type_of)
{-10}	0 ≤ a' offset
{-11}	a' offset < max_linear_offset
{-12}	cons?[Byte](bl')
{-13}	is_linear_plain_memory?(pm')
{-14}	(address_block(a', length(bl'))) ⊆ pm'rw_addr
{-15}	pm' states(s')
{1}	pm'rw_addr(t'1)
{2}	data((linear_resolve(t'1, Write) ## (λ (pa: Memory_Address_4G)(ns: Linear_memory[Physical_memory, pm_phy]): memory_write_side_effect(pm_phy'mem)(pa, t'2, TRUE)(s')) (s')) = t'2

Hiding formulas: (-7 -8 -13 2),

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of split_linear_write_side_effects_data.1.1.1.1.1.2.


```

split_linear_write_side_effects_data.1.1.1.1.2:
|- {1} every(λ (t_1: [Address, list[Byte]]):
    cons?(t_1'2) ∧
    a' ≤ t_1'1 ∧
    t_1'1 + length(t_1'2) ≤ a' + length(bl') ∧
    type_of(t_1'1) = type_of(a') ∧
    reg_base(min_linear)(type_of(a')) ≤ t_1'1 ∧
    t_1'1 < reg_size(max_linear)(type_of(a')) ∧
    rem(expt(2, min_page))(offset(t_1'1)) + length(t_1'2) ≤
    expt(2, min_page))
    (split(min_page, a', bl'))
  ⊃
  every(λ (s: [Memory_Address_4G, list[Byte]]):
    cons?(s'2) ∧
    a' ≤ s'1 ∧
    s'1 + length(s'2) ≤ a' + length(bl') ∧
    type_of(s'1) = type_of(a') ∧
    reg_base(min_linear)(type_of(a')) ≤ s'1 ∧
    s'1 < reg_size(max_linear)(type_of(a')) ∧
    rem(expt(2, min_page))(offset(s'1)) + length(s'2) ≤
    expt(2, min_page))
    (split(min_page, a', bl'))
|- {2} every(λ (t: [Address, list[Byte]]):
    (cons?(t'2) ∧
    a' ≤ t'1 ∧
    t'1 + length(t'2) ≤ a' + length(bl') ∧
    type_of(t'1) = type_of(a') ∧
    reg_base((#type_of := Mem_, offset := 0#))(type_of(a')) ≤ t'1 ∧
    t'1 <
    reg_size((#type_of := Mem_, offset := max_linear_offset#))(type_of(a')))
    ∧
    rem(expt(2, min_page))(offset(t'1)) + length(t'2) ≤
    expt(2, min_page))
    (split(min_page, a', bl'))
|- {3} Mem?(a' 'type_of)
|- {4} 0 ≤ a' 'offset
|- {5} a' 'offset < max_linear_offset
|- {6} cons?[Byte](bl')
|- {7} is_linear_plain_memory?(pm')
|- {8} (address_block(a', length(bl')) ⊆ pm' 'rw_addr)
|- {9} pm' 'states(s')
-----
|- {1} ∀ (s: [Memory_Address_4G, list[Byte]]):
    reg_base(Mem(0))(type_of(a')) ≤ s'1 ∧
    type_of(s'1) = type_of(a') ∧
    s'1 + length(s'2) ≤ a' + length(bl') ∧ a' ≤ s'1 ∧ cons?(s'2)
  ⊃ Mem?(Mem(max_linear_offset) 'type_of)
|- {2} every(λ (e: [Memory_Address_4G, list[Byte]]):
    OK?(linear_write_side_effect_in_page(e)(s')) ⊃
    data(linear_write_side_effect_in_page(e)(s')) = e'2)
    (split(min_page, a', bl'))

```

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `split_linear_write_side_effects_data.1.1.1.1.2`.

split_linear_write_side_effects_data.1.1.1.1.3:

{-1}	every(λ (t_1: [Address, list[Byte]]): cons?(t_1'2) \wedge $a' \leq t_1'1 \wedge$ $t_1'1 + \text{length}(t_1'2) \leq a' + \text{length}(bl') \wedge$ type_of(t_1'1) = type_of(a') \wedge reg_base(min_linear)(type_of(a')) $\leq t_1'1 \wedge$ $t_1'1 < \text{reg_size}(\text{max_linear})(\text{type_of}(a')) \wedge$ $\text{rem}(\text{expt}(2, \text{min_page}))(\text{offset}(t_1'1)) + \text{length}(t_1'2) \leq$ $\text{expt}(2, \text{min_page})$) (split(min_page, a', bl')) \supset every(λ (s: [Memory_Address_4G, list[Byte]]): cons?(s'2) \wedge $a' \leq s'1 \wedge$ $s'1 + \text{length}(s'2) \leq a' + \text{length}(bl') \wedge$ type_of(s'1) = type_of(a') \wedge reg_base(min_linear)(type_of(a')) $\leq s'1 \wedge$ $s'1 < \text{reg_size}(\text{max_linear})(\text{type_of}(a')) \wedge$ $\text{rem}(\text{expt}(2, \text{min_page}))(\text{offset}(s'1)) + \text{length}(s'2) \leq$ $\text{expt}(2, \text{min_page})$) (split(min_page, a', bl')))
{-2}	every(λ (t: [Address, list[Byte]]): (cons?(t'2) \wedge $a' \leq t'1 \wedge$ $t'1 + \text{length}(t'2) \leq a' + \text{length}(bl') \wedge$ type_of(t'1) = type_of(a') \wedge reg_base((#type_of := Mem_, offset := 0#))(type_of(a')) $\leq t'1 \wedge$ $t'1 <$ reg_size((#type_of := Mem_, offset := max_linear_offset#))(type_of(a')))) \wedge $\text{rem}(\text{expt}(2, \text{min_page}))(\text{offset}(t'1)) + \text{length}(t'2) \leq$ $\text{expt}(2, \text{min_page})$) (split(min_page, a', bl')))
{-3}	Mem?(a' type_of)
{-4}	$0 \leq a' \text{ offset}$
{-5}	$a' \text{ offset} < \text{max_linear_offset}$
{-6}	cons?[Byte](bl')
{-7}	is_linear_plain_memory?(pm')
{-8}	(address_block(a', length(bl')) \subseteq pm'rw_addr)
{-9}	pm' states(s')
{1}	\forall (s: [Memory_Address_4G, list[Byte]]): type_of(s'1) = type_of(a') \wedge $s'1 + \text{length}(s'2) \leq a' + \text{length}(bl') \wedge a' \leq s'1 \wedge \text{cons?}(s'2)$ \supset Mem?(Mem(0) type_of))
{2}	every(λ (e: [Memory_Address_4G, list[Byte]]): OK?(linear_write_side_effect_in_page(e)(s')) \supset data(linear_write_side_effect_in_page(e)(s')) = e'2) (split(min_page, a', bl')))

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of split_linear_write_side_effects_data.1.1.1.1.3.

split_linear_write_side_effects_data.1.1.1.2:

<pre> {-1} every(λ (t: [Address, list[Byte]]): (cons?(t'2) ∧ a' ≤ t'1 ∧ t'1 + length(t'2) ≤ a' + length(bl') ∧ type_of(t'1) = type_of(a') ∧ reg_base((#type_of := Mem_, offset := 0#))(type_of(a')) ≤ t'1 ∧ t'1 < reg_size((#type_of := Mem_, offset := max_linear_offset#))(type_of(a'))) ∧ rem(expt(2, min_page))(offset(t'1)) + length(t'2) ≤ expt(2, min_page) (split(min_page, a', bl')) {-2} Mem?(a' 'type_of) {-3} 0 ≤ a' 'offset {-4} a' 'offset < max_linear_offset {-5} cons?[Byte](bl') {-6} is_linear_plain_memory?(pm') {-7} (address_block(a', length(bl')) ⊆ pm' 'rw_addr) {-8} pm' 'states(s') </pre>	<pre> {1} ∀ (t: [Address, list[Byte]]): reg_base(Mem(0))(type_of(a')) ≤ t'1 ∧ type_of(t'1) = type_of(a') ∧ t'1 + length(t'2) ≤ a' + length(bl') ∧ a' ≤ t'1 ∧ cons?(t'2) ⊃ Mem?(Mem(max_linear_offset) 'type_of) {2} every(λ (e: [Memory_Address_4G, list[Byte]]): OK?(linear_write_side_effect_in_page(e)(s')) ⊃ data(linear_write_side_effect_in_page(e)(s')) = e'2 (split(min_page, a', bl')) </pre>
---	---

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `split_linear_write_side_effects_data.1.1.1.2`.

split_linear_write_side_effects_data.1.1.1.3:

<p>{-1} every(λ (t: [Address, list[Byte]]): (cons?($t'2$) \wedge $a' \leq t'1 \wedge$ $t'1 + \text{length}(t'2) \leq a' + \text{length}(bl') \wedge$ type_of($t'1$) = type_of(a') \wedge reg_base((#type_of := Mem_, offset := 0#))(type_of(a')) $\leq t'1 \wedge$ $t'1 <$ reg_size((#type_of := Mem_, offset := max_linear_offset#))(type_of(a'))) \wedge rem(expt(2, min_page))(offset($t'1$)) + length($t'2$) \leq expt(2, min_page)) (split(min_page, a', bl'))</p> <p>{-2} Mem?(a''type_of)</p> <p>{-3} $0 \leq a'$'offset</p> <p>{-4} a''offset < max_linear_offset</p> <p>{-5} cons?[Byte](bl')</p> <p>{-6} is_linear_plain_memory?(pm')</p> <p>{-7} (address_block(a', length(bl'))) \subseteq pm'rw_addr)</p> <p>{-8} pm' states(s')</p>	<hr/> <p>{1} \forall (t: [Address, list[Byte]]): type_of($t'1$) = type_of(a') \wedge $t'1 + \text{length}(t'2) \leq a' + \text{length}(bl') \wedge a' \leq t'1 \wedge$ cons?($t'2$) \supset Mem?(Mem(0)'type_of)</p> <p>{2} every(λ (e: [Memory_Address_4G, list[Byte]]): OK?(linear_write_side_effect_in_page(e)(s')) \supset data(linear_write_side_effect_in_page(e)(s')) = $e'2$) (split(min_page, a', bl'))</p>
--	---

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of split_linear_write_side_effects_data.1.1.1.3.

split_linear_write_side_effects_data.1.1.2:

```

{-1} every(λ (t: [Address, list[Byte]]):
      cons?(t'2) ∧
      a' ≤ t'1 ∧ t'1 + length(t'2) ≤ a' + length(bl')
      (split(min_page, a', bl'))
      ∧
      every(λ (t: [Address, list[Byte]]):
            type_of(t'1) = type_of(a') ∧
            reg_base(min_linear)(type_of(a')) ≤ t'1 ∧
            t'1 < reg_size(max_linear)(type_of(a'))
            (split(min_page, a', bl'))
            ⊃
            every(λ (t_1: [Address, list[Byte]]):
                  (cons?(t_1'2) ∧
                   a' ≤ t_1'1 ∧ t_1'1 + length(t_1'2) ≤ a' + length(bl'))
                  ∧
                  type_of(t_1'1) = type_of(a') ∧
                  reg_base(min_linear)(type_of(a')) ≤ t_1'1 ∧
                  t_1'1 < reg_size(max_linear)(type_of(a'))
                  (split(min_page, a', bl'))
            )
      )
      ⊃
      every(λ (a_2: Address, bl2: list[Byte]): cons?(bl2))(split(min_page, a', bl')) ∧
      every(λ (a_2: Address, bl2: list[Byte]):
            a' ≤ a_2 ∧ a_2 + length(bl2) ≤ a' + length(bl')
            (split(min_page, a', bl'))
            ⊃
            every(λ (t: [Address, list[Byte]]):
                  cons?(t'2) ∧
                  a' ≤ t'1 ∧ t'1 + length(t'2) ≤ a' + length(bl')
                  (split(min_page, a', bl'))
            )
      )
      ⊃
      every(λ (a_2: Address, bl2: list[Byte]):
            rem(expt(2, min_page))(offset(a_2)) + length(bl2) ≤ expt(2, min_page)
            (split(min_page, a', bl'))
      )
      ⊃
      every(λ (t: [Address, list[Byte]]):
            type_of(t'1) = type_of(a') ∧
            reg_base((#type_of := Mem_, offset := 0#))(type_of(a')) ≤ t'1 ∧
            t'1 < reg_size((#type_of := Mem_, offset := max_linear_offset#))(type_of(a'))
            (split(min_page, a', bl'))
      )
      ⊃
      every(λ (a_2: Address, bl2: list[Byte]): cons?(bl2))(split(min_page, a', bl'))
      ⊃
      every(λ (a_2: Address, bl2: list[Byte]):
            a' ≤ a_2 ∧ a_2 + length(bl2) ≤ a' + length(bl')
            (split(min_page, a', bl'))
      )
      ⊃
      Mem?(a' type_of)
      ⊃
      0 ≤ a' offset
      ⊃
      a' offset < max_linear_offset
      ⊃
      every(λ (x: number):
            number_field_pred(x) ∧
            real_pred(x) ∧
            rational_pred(x) ∧ integer_pred(x) ∧ x ≥ 0 ∧ x < max_byte)
      (bl')
      ⊃
      cons?[Byte](bl')
      ⊃
      is_linear_plain_memory?(pm')
      ⊃
      (address_block(a', length(bl')) ⊆ pm' rw_addr)
      ⊃
      pm' states(s')
      ⊃
      {1} ∀ (t_1: [Address, list[Byte]]):
            reg_base(Mem(0))(type_of(a')) ≤ t_1'1 ∧
            type_of(t_1'1) = type_of(a') ∧
            t_1'1 + length(t_1'2) ≤ a' + length(bl') ∧
            a' ≤ t_1'1 ∧ cons?(t_1'2)
            ⊃ Mem?(Mem(max_linear_offset) type_of)
      {2} every(λ (e: [Memory_Address_4G, list[Byte]]):
            OK?(linear_write_side_effect_in_page(e)(s')) ⊃
            data(linear_write_side_effect_in_page(e)(s')) = e'2
            (split(min_page, a', bl'))
      )

```

C.117 Proofs for Linear_Memory_Properties (challenge-linear.pvs)

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `split_linear_write_side_effects_data.1.1.2`.

split_linear_write_side_effects_data.1.1.3:

{-1}	$\text{every}(\lambda (t: [\text{Address}, \text{list}[\text{Byte}]]):$ $\quad \text{cons?}(t'2) \wedge$ $\quad a' \leq t'1 \wedge t'1 + \text{length}(t'2) \leq a' + \text{length}(bl')$ $\quad (\text{split}(\text{min_page}, a', bl'))$ $\quad \wedge$ $\quad \text{every}(\lambda (t: [\text{Address}, \text{list}[\text{Byte}]]):$ $\quad \quad \text{type_of}(t'1) = \text{type_of}(a') \wedge$ $\quad \quad \text{reg_base}(\text{min_linear})(\text{type_of}(a')) \leq t'1 \wedge$ $\quad \quad t'1 < \text{reg_size}(\text{max_linear})(\text{type_of}(a'))$ $\quad \quad (\text{split}(\text{min_page}, a', bl'))$ $\quad \supset$ $\quad \text{every}(\lambda (t_1: [\text{Address}, \text{list}[\text{Byte}]]):$ $\quad \quad (\text{cons?}(t_1'2) \wedge$ $\quad \quad a' \leq t_1'1 \wedge t_1'1 + \text{length}(t_1'2) \leq a' + \text{length}(bl'))$ $\quad \quad \wedge$ $\quad \quad \text{type_of}(t_1'1) = \text{type_of}(a') \wedge$ $\quad \quad \text{reg_base}(\text{min_linear})(\text{type_of}(a')) \leq t_1'1 \wedge$ $\quad \quad t_1'1 < \text{reg_size}(\text{max_linear})(\text{type_of}(a'))$ $\quad \quad (\text{split}(\text{min_page}, a', bl'))$
{-2}	$\text{every}(\lambda (a_2: \text{Address}, bl_2: \text{list}[\text{Byte}]): \text{cons?}(bl_2)(\text{split}(\text{min_page}, a', bl')) \wedge$ $\quad \text{every}(\lambda (a_2: \text{Address}, bl_2: \text{list}[\text{Byte}]):$ $\quad \quad a' \leq a_2 \wedge a_2 + \text{length}(bl_2) \leq a' + \text{length}(bl'))$ $\quad \quad (\text{split}(\text{min_page}, a', bl'))$ $\quad \supset$ $\quad \text{every}(\lambda (t: [\text{Address}, \text{list}[\text{Byte}]]):$ $\quad \quad \text{cons?}(t'2) \wedge$ $\quad \quad a' \leq t'1 \wedge t'1 + \text{length}(t'2) \leq a' + \text{length}(bl'))$ $\quad \quad (\text{split}(\text{min_page}, a', bl'))$
{-3}	$\text{every}(\lambda (a_2: \text{Address}, bl_2: \text{list}[\text{Byte}]):$ $\quad \text{rem}(\text{expt}(2, \text{min_page}))(\text{offset}(a_2)) + \text{length}(bl_2) \leq \text{expt}(2, \text{min_page}))$ $\quad (\text{split}(\text{min_page}, a', bl'))$
{-4}	$\text{every}(\lambda (t: [\text{Address}, \text{list}[\text{Byte}]]):$ $\quad \text{type_of}(t'1) = \text{type_of}(a') \wedge$ $\quad \text{reg_base}((\# \text{type_of} := \text{Mem_}, \text{offset} := 0\#))(\text{type_of}(a')) \leq t'1 \wedge$ $\quad t'1 < \text{reg_size}((\# \text{type_of} := \text{Mem_}, \text{offset} := \text{max_linear_offset}\#))(\text{type_of}(a'))$ $\quad (\text{split}(\text{min_page}, a', bl'))$
{-5}	$\text{every}(\lambda (a_2: \text{Address}, bl_2: \text{list}[\text{Byte}]): \text{cons?}(bl_2)(\text{split}(\text{min_page}, a', bl'))$
{-6}	$\text{every}(\lambda (a_2: \text{Address}, bl_2: \text{list}[\text{Byte}]):$ $\quad a' \leq a_2 \wedge a_2 + \text{length}(bl_2) \leq a' + \text{length}(bl'))$ $\quad (\text{split}(\text{min_page}, a', bl'))$
{-7}	$\text{Mem?}(a' \text{ type_of})$
{-8}	$0 \leq a' \text{ offset}$
{-9}	$a' \text{ offset} < \text{max_linear_offset}$
{-10}	$\text{every}(\lambda (x: \text{number}):$ $\quad \text{number_field_pred}(x) \wedge$ $\quad \text{real_pred}(x) \wedge$ $\quad \text{rational_pred}(x) \wedge \text{integer_pred}(x) \wedge x \geq 0 \wedge x < \text{max_byte})$ $\quad (bl')$
{-11}	$\text{cons?}[\text{Byte}](bl')$
{-12}	$\text{is_linear_plain_memory?}(pm')$
{-13}	$(\text{address_block}(a', \text{length}(bl')) \subseteq pm' \text{ rw_addr})$
{-14}	$pm' \text{ states}(s')$
{1}	$\forall (t_1: [\text{Address}, \text{list}[\text{Byte}]]):$ $\quad \text{type_of}(t_1'1) = \text{type_of}(a') \wedge$ $\quad t_1'1 + \text{length}(t_1'2) \leq a' + \text{length}(bl') \wedge$ $\quad a' \leq t_1'1 \wedge \text{cons?}(t_1'2)$ $\quad \supset \text{Mem?}(\text{Mem}(0) \text{ type_of})$
{2}	$\text{every}(\lambda (e: [\text{Memory_Address_4G}, \text{list}[\text{Byte}]]):$ $\quad \text{OK?}(\text{linear_write_side_effect_in_page}(e)(s')) \supset$ $\quad \text{data}(\text{linear_write_side_effect_in_page}(e)(s')) = e'2)$ $\quad (\text{split}(\text{min_page}, a', bl'))$

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `split_linear_write_side_effects_data.1.1.3`.

`split_linear_write_side_effects_data.1.2`:

{-1}	$\text{every}(\lambda (a_2: \text{Address}, \text{bl2}: \text{list}[\text{Byte}]): \text{cons}^?(\text{bl2}))(\text{split}(\text{min_page}, a', \text{bl}')) \wedge$ $\text{every}(\lambda (a_2: \text{Address}, \text{bl2}: \text{list}[\text{Byte}]):$ $a' \leq a_2 \wedge a_2 + \text{length}(\text{bl2}) \leq a' + \text{length}(\text{bl}'))$ $(\text{split}(\text{min_page}, a', \text{bl}'))$ \supset $\text{every}(\lambda (t: [\text{Address}, \text{list}[\text{Byte}]]):$ $\text{cons}^?(t'2) \wedge$ $a' \leq t'1 \wedge t'1 + \text{length}(t'2) \leq a' + \text{length}(\text{bl}'))$ $(\text{split}(\text{min_page}, a', \text{bl}'))$
{-2}	$\text{every}(\lambda (a_2: \text{Address}, \text{bl2}: \text{list}[\text{Byte}]):$ $\text{rem}(\text{expt}(2, \text{min_page}))(\text{offset}(a_2)) + \text{length}(\text{bl2}) \leq \text{expt}(2, \text{min_page}))$ $(\text{split}(\text{min_page}, a', \text{bl}'))$
{-3}	$\text{every}(\lambda (t: [\text{Address}, \text{list}[\text{Byte}]]):$ $\text{type_of}(t'1) = \text{type_of}(a') \wedge$ $\text{reg_base}(\text{\#type_of} := \text{Mem_}, \text{offset} := 0\#)(\text{type_of}(a')) \leq t'1 \wedge$ $t'1 < \text{reg_size}(\text{\#type_of} := \text{Mem_}, \text{offset} := \text{max_linear_offset}\#)(\text{type_of}(a'))$ $(\text{split}(\text{min_page}, a', \text{bl}'))$
{-4}	$\text{every}(\lambda (a_2: \text{Address}, \text{bl2}: \text{list}[\text{Byte}]): \text{cons}^?(\text{bl2}))(\text{split}(\text{min_page}, a', \text{bl}'))$
{-5}	$\text{every}(\lambda (a_2: \text{Address}, \text{bl2}: \text{list}[\text{Byte}]):$ $a' \leq a_2 \wedge a_2 + \text{length}(\text{bl2}) \leq a' + \text{length}(\text{bl}'))$ $(\text{split}(\text{min_page}, a', \text{bl}'))$
{-6}	$\text{Mem}^?(a' \text{ type_of})$
{-7}	$0 \leq a' \text{ offset}$
{-8}	$a' \text{ offset} < \text{max_linear_offset}$
{-9}	$\text{every}(\lambda (x: \text{number}):$ $\text{number_field_pred}(x) \wedge$ $\text{real_pred}(x) \wedge$ $\text{rational_pred}(x) \wedge \text{integer_pred}(x) \wedge x \geq 0 \wedge x < \text{max_byte})$ (bl')
{-10}	$\text{cons}^?[\text{Byte}](\text{bl}')$
{-11}	$\text{is_linear_plain_memory}^?(\text{pm}')$
{-12}	$(\text{address_block}(a', \text{length}(\text{bl}')) \subseteq \text{pm}' \text{ rw_addr})$
{-13}	$\text{pm}' \text{ states}(s')$
{1}	$\forall (t: [\text{Address}, \text{list}[\text{Byte}]]):$ $\text{reg_base}(\text{Mem}(0))(\text{type_of}(a')) \leq t'1 \wedge \text{type_of}(t'1) = \text{type_of}(a') \supset$ $\text{Mem}^?(\text{Mem}(\text{max_linear_offset}) \text{ type_of})$
{2}	$\text{every}(\lambda (e: [\text{Memory_Address_4G}, \text{list}[\text{Byte}]]):$ $\text{OK}^?(\text{linear_write_side_effect_in_page}(e)(s')) \supset$ $\text{data}(\text{linear_write_side_effect_in_page}(e)(s')) = e'2)$ $(\text{split}(\text{min_page}, a', \text{bl}'))$

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `split_linear_write_side_effects_data.1.2`.

split_linear_write_side_effects_data.1.3:

{-1}	$\text{every}(\lambda (a_2: \text{Address}, \text{bl2}: \text{list}[\text{Byte}]): \text{cons}?(a_2))(\text{split}(\text{min_page}, a', \text{bl}')) \wedge$ $\text{every}(\lambda (a_2: \text{Address}, \text{bl2}: \text{list}[\text{Byte}]):$ $a' \leq a_2 \wedge a_2 + \text{length}(\text{bl2}) \leq a' + \text{length}(\text{bl}'))$ $(\text{split}(\text{min_page}, a', \text{bl}'))$ \supset $\text{every}(\lambda (t: [\text{Address}, \text{list}[\text{Byte}]]):$ $\text{cons}?(t'2) \wedge$ $a' \leq t'1 \wedge t'1 + \text{length}(t'2) \leq a' + \text{length}(\text{bl}'))$ $(\text{split}(\text{min_page}, a', \text{bl}'))$
{-2}	$\text{every}(\lambda (a_2: \text{Address}, \text{bl2}: \text{list}[\text{Byte}]):$ $\text{rem}(\text{expt}(2, \text{min_page}))(\text{offset}(a_2)) + \text{length}(\text{bl2}) \leq \text{expt}(2, \text{min_page}))$ $(\text{split}(\text{min_page}, a', \text{bl}'))$
{-3}	$\text{every}(\lambda (t: [\text{Address}, \text{list}[\text{Byte}]]):$ $\text{type_of}(t'1) = \text{type_of}(a') \wedge$ $\text{reg_base}(\text{\#type_of} := \text{Mem_}, \text{offset} := 0\#)(\text{type_of}(a')) \leq t'1 \wedge$ $t'1 < \text{reg_size}(\text{\#type_of} := \text{Mem_}, \text{offset} := \text{max_linear_offset\#})(\text{type_of}(a')))$ $(\text{split}(\text{min_page}, a', \text{bl}'))$
{-4}	$\text{every}(\lambda (a_2: \text{Address}, \text{bl2}: \text{list}[\text{Byte}]): \text{cons}?(a_2))(\text{split}(\text{min_page}, a', \text{bl}'))$
{-5}	$\text{every}(\lambda (a_2: \text{Address}, \text{bl2}: \text{list}[\text{Byte}]):$ $a' \leq a_2 \wedge a_2 + \text{length}(\text{bl2}) \leq a' + \text{length}(\text{bl}'))$ $(\text{split}(\text{min_page}, a', \text{bl}'))$
{-6}	Mem?(a' type_of)
{-7}	$0 \leq a' \text{offset}$
{-8}	$a' \text{offset} < \text{max_linear_offset}$
{-9}	$\text{every}(\lambda (x: \text{number}):$ $\text{number_field_pred}(x) \wedge$ $\text{real_pred}(x) \wedge$ $\text{rational_pred}(x) \wedge \text{integer_pred}(x) \wedge x \geq 0 \wedge x < \text{max_byte})$ (bl')
{-10}	cons?[Byte](bl')
{-11}	is_linear_plain_memory?(pm')
{-12}	(address_block(a', length(bl')) \subseteq pm'rw_addr)
{-13}	pm' states(s')
{1}	$\forall (t: [\text{Address}, \text{list}[\text{Byte}]]):$ $\text{type_of}(t'1) = \text{type_of}(a') \supset \text{Mem}?(a' \text{type_of})$
{2}	$\text{every}(\lambda (e: [\text{Memory_Address_4G}, \text{list}[\text{Byte}]]):$ $\text{OK}?(a' \text{linear_write_side_effect_in_page}(e)(s')) \supset$ $\text{data}(a' \text{linear_write_side_effect_in_page}(e)(s')) = e'2)$ $(\text{split}(\text{min_page}, a', \text{bl}'))$

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of split_linear_write_side_effects_data.1.3.

split_linear_write_side_effects_data.2:

{-1}	every(λ (a_2 : Address, bl2: list[Byte]): rem(expt(2, min_page))(offset(a_2)) + length(bl2) \leq expt(2, min_page)) (split(min_page, a' , bl'))
{-2}	every(λ (a_2 : Address, bl2: list[Byte]): cons?(bl2))(split(min_page, a' , bl'))
{-3}	every(λ (a_2 : Address, bl2: list[Byte]): $a' \leq a_2 \wedge a_2 + \text{length}(bl2) \leq a' + \text{length}(bl')$ (split(min_page, a' , bl'))
{-4}	Mem?(a' 'type_of)
{-5}	$0 \leq a'$ 'offset
{-6}	a' 'offset < max_linear_offset
{-7}	every(λ (x : number): number_field_pred(x) \wedge real_pred(x) \wedge rational_pred(x) \wedge integer_pred(x) \wedge $x \geq 0 \wedge x < \text{max_byte}$) (bl')
{-8}	cons?[Byte](bl')
{-9}	is_linear_plain_memory?(pm')
{-10}	(address_block(a' , length(bl')) \subseteq pm'rw_addr)
{-11}	pm' states(s')
{1}	$a' + \text{length}(bl') \leq$ reg_size(#type_of := Mem_, offset := max_linear_offset#)(type_of(a'))
{2}	every(λ (e : [Memory_Address_4G, list[Byte]]): OK?(linear_write_side_effect_in_page(e)(s')) \supset data(linear_write_side_effect_in_page(e)(s')) = e '2) (split(min_page, a' , bl'))

Hiding formulas: (-1 -2 -3 -7 2),

Expanding the definition of length,

Using lemma pm_memory_addr,

Simplifying, rewriting, and recording with decision procedures,

Hiding formulas: -6,

Expanding the definition of subset?,

Installing automatic rewrites from: (bus_width!)

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of split_linear_write_side_effects_data.2.

split_linear_write_side_effects_data.3:

{-1}	every(λ (a_2 : Address, bl2: list[Byte]): rem(expt(2, min_page))(offset(a_2)) + length(bl2) \leq expt(2, min_page)) (split(min_page, a' , bl'))
{-2}	every(λ (a_2 : Address, bl2: list[Byte]): cons?(bl2))(split(min_page, a' , bl'))
{-3}	every(λ (a_2 : Address, bl2: list[Byte]): $a' \leq a_2 \wedge a_2 + \text{length}(bl2) \leq a' + \text{length}(bl')$ (split(min_page, a' , bl'))
{-4}	Mem?(a' 'type_of)
{-5}	$0 \leq a'$ 'offset
{-6}	a' 'offset < max_linear_offset
{-7}	every(λ (x : number): number_field_pred(x) \wedge real_pred(x) \wedge rational_pred(x) \wedge integer_pred(x) $\wedge x \geq 0 \wedge x < \text{max_byte}$) (bl')
{-8}	cons?[Byte](bl')
{-9}	is_linear_plain_memory?(pm')
{-10}	(address_block(a' , length(bl')) \subseteq pm'rw_addr)
{-11}	pm'states(s')
{1}	reg_base((#type_of := Mem_, offset := 0#))(type_of(a')) $\leq a'$
{2}	every(λ (e : [Memory_Address_4G, list[Byte]]): OK?(linear_write_side_effect_in_page(e)(s')) \supset data(linear_write_side_effect_in_page(e)(s')) = $e'2$) (split(min_page, a' , bl'))

Expanding the definition of reg_base,

Expanding the definition of \leq ,

which is trivially true.

This completes the proof of split_linear_write_side_effects_data.3.

Q.E.D.

C.117.82 Linear_Memory_Properties.split_linear_read_side_effects_unchanged_memory

Terse proof for split_linear_read_side_effects_unchanged_memory.

split_linear_read_side_effects_unchanged_memory:

{1}	\forall (pm: Plain_Memory[Linear_memory], a : Memory_Address_4G, bl: (cons?[Byte])): is_linear_plain_memory?(pm) \wedge (address_block(a , length(bl)) \subseteq (pm'ro_addr \cup pm'rw_addr)) \supset every(λ (e : [Memory_Address_4G, list[Byte]]): unchanged_memory_invariant?(pm_phy'mem, pm'states, singleton(expr_2_super(linear_read_side_effect_in_page(e), (e))), (pm_phy'ro_addr \cup pm_phy'rw_addr))) (split(min_page, a , bl))
-----	---

Repeatedly Skolemizing and flattening,

Using lemma split_range,

Using lemma split_no_null,

Using lemma split_type,

Using lemma split_pair_cross_size,

Using lemma split_linear_read_side_effects_states,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 3 subgoals:

split_linear_read_side_effects_unchanged_memory.1:

{-1}	is_linear_plain_memory?(pm')
{-2}	(address_block(a', length(bl')) \subseteq (pm'ro_addr \cup pm'rw_addr))
{-3}	every(λ (e: [Memory_Address_4G, list[Byte]]): transformer_invariant?(pm' states, singleton(expr_2_super(linear_read_side_effect_in_page(e)))) (split(min_page, a', bl'))
{-4}	every(λ (a ₂ : Address, bl2: list[Byte]): rem(expt(2, min_page))(offset(a ₂) + length(bl2)) \leq expt(2, min_page)) (split(min_page, a', bl'))
{-5}	every(λ (t: [Address, list[Byte]]): type_of(t'1) = type_of(a') \wedge reg_base(min_linear)(type_of(a')) \leq t'1 \wedge t'1 < reg_size(max_linear)(type_of(a')) (split(min_page, a', bl'))
{-6}	every(λ (a ₂ : Address, bl2: list[Byte]): cons?(bl2))(split(min_page, a', bl'))
{-7}	every(λ (a ₂ : Address, bl2: list[Byte]): a' \leq a ₂ \wedge a ₂ + length(bl2) \leq a' + length(bl') (split(min_page, a', bl'))
{-8}	Mem?(a' type_of)
{-9}	0 \leq a' offset
{-10}	a' offset < max_linear_offset
{-11}	every(λ (x: number): number_field_pred(x) \wedge real_pred(x) \wedge rational_pred(x) \wedge integer_pred(x) \wedge x \geq 0 \wedge x < max_byte) (bl')
{-12}	cons?[Byte](bl')
{1}	every(λ (e: [Memory_Address_4G, list[Byte]]): unchanged_memory_invariant?(pm_phy mem, pm' states, singleton(expr_2_super(linear_read_side_effect_in_page (e))), (pm_phy ro_addr \cup pm_phy rw_addr)) (split(min_page, a', bl'))

Using lemma every_conjunct_left,

Using lemma every_conjunct_left,

we get 3 subgoals:

split_linear_read_side_effects_unchanged_memory.1.1:

```

{-1} every(λ (t: [Address, list[Byte]]):
      cons?(t'2) ∧
      a' ≤ t'1 ∧ t'1 + length(t'2) ≤ a' + length(bl')
      (split(min_page, a', bl'))
    ∧
    every(λ (t: [Address, list[Byte]]):
      type_of(t'1) = type_of(a') ∧
      reg_base(min_linear)(type_of(a')) ≤ t'1 ∧
      t'1 < reg_size(max_linear)(type_of(a'))
      (split(min_page, a', bl'))
    ⊃
    every(λ (t_1: [Address, list[Byte]]):
      (cons?(t_1'2) ∧
      a' ≤ t_1'1 ∧ t_1'1 + length(t_1'2) ≤ a' + length(bl'))
      ∧
      type_of(t_1'1) = type_of(a') ∧
      reg_base(min_linear)(type_of(a')) ≤ t_1'1 ∧
      t_1'1 < reg_size(max_linear)(type_of(a'))
      (split(min_page, a', bl'))
  {-2} every(λ (a_2: Address, bl_2: list[Byte]): cons?(bl_2))(split(min_page, a', bl')) ∧
    every(λ (a_2: Address, bl_2: list[Byte]):
      a' ≤ a_2 ∧ a_2 + length(bl_2) ≤ a' + length(bl')
      (split(min_page, a', bl'))
    ⊃
    every(λ (t: [Address, list[Byte]]):
      cons?(t'2) ∧
      a' ≤ t'1 ∧ t'1 + length(t'2) ≤ a' + length(bl')
      (split(min_page, a', bl'))
  {-3} is_linear_plain_memory?(pm')
  {-4} (address_block(a', length(bl')) ⊆ (pm'ro_addr ∪ pm'rw_addr))
  {-5} every(λ (e: [Memory_Address_4G, list[Byte]]):
      transformer_invariant?(pm' states,
      singleton(expr_2_super(linear_read_side_effect_in_page(e)))
      (split(min_page, a', bl'))
  {-6} every(λ (a_2: Address, bl_2: list[Byte]):
      rem(expt(2, min_page))(offset(a_2)) + length(bl_2) ≤ expt(2, min_page)
      (split(min_page, a', bl'))
  {-7} every(λ (t: [Address, list[Byte]]):
      type_of(t'1) = type_of(a') ∧
      reg_base(min_linear)(type_of(a')) ≤ t'1 ∧
      t'1 < reg_size(max_linear)(type_of(a'))
      (split(min_page, a', bl'))
  {-8} every(λ (a_2: Address, bl_2: list[Byte]): cons?(bl_2))(split(min_page, a', bl'))
  {-9} every(λ (a_2: Address, bl_2: list[Byte]):
      a' ≤ a_2 ∧ a_2 + length(bl_2) ≤ a' + length(bl')
      (split(min_page, a', bl'))
  {-10} Mem?(a' type_of)
  {-11} 0 ≤ a' offset
  {-12} a' offset < max_linear_offset
  {-13} every(λ (x: number):
      number_field_pred(x) ∧
      real_pred(x) ∧
      rational_pred(x) ∧ integer_pred(x) ∧ x ≥ 0 ∧ x < max_byte)
      (bl')
  {-14} cons?[Byte](bl')
2252 {1} every(λ (e: [Memory_Address_4G, list[Byte]]):
      unchanged_memory_invariant?(pm_phy mem, pm' states,
      singleton(expr_2_super(linear_read_side_effect_in_page
      (e))),
      (pm_phy ro_addr ∪ pm_phy rw_addr))
      (split(min_page, a', bl'))

```

Using lemma `every_conjunct_left`,

we get 3 subgoals:

split_linear_read_side_effects_unchanged_memory.1.1.1:

```

{-1}  every(λ (t_1: [Address, list[Byte]]):
      cons?(t_1'2) ∧
      a' ≤ t_1'1 ∧
      t_1'1 + length(t_1'2) ≤ a' + length(bl') ∧
      type_of(t_1'1) = type_of(a') ∧
      reg_base(min_linear)(type_of(a')) ≤ t_1'1 ∧
      t_1'1 < reg_size(max_linear)(type_of(a')))
      (split(min_page, a', bl'))
    ∧
    every(λ (a_2: Address, bl2: list[Byte]):
      rem(expt(2, min_page))(offset(a_2)) + length(bl2) ≤
      expt(2, min_page))
      (split(min_page, a', bl'))
    ⊃
    every(λ (t: [Address, list[Byte]]):
      (cons?(t'2) ∧
      a' ≤ t'1 ∧
      t'1 + length(t'2) ≤ a' + length(bl') ∧
      type_of(t'1) = type_of(a') ∧
      reg_base(min_linear)(type_of(a')) ≤ t'1 ∧
      t'1 < reg_size(max_linear)(type_of(a')))
      ∧
      rem(expt(2, min_page))(offset(t'1)) + length(t'2) ≤
      expt(2, min_page))
      (split(min_page, a', bl'))
{-2}  every(λ (t: [Address, list[Byte]]):
      cons?(t'2) ∧
      a' ≤ t'1 ∧ t'1 + length(t'2) ≤ a' + length(bl'))
      (split(min_page, a', bl'))
    ∧
    every(λ (t: [Address, list[Byte]]):
      type_of(t'1) = type_of(a') ∧
      reg_base(min_linear)(type_of(a')) ≤ t'1 ∧
      t'1 < reg_size(max_linear)(type_of(a')))
      (split(min_page, a', bl'))
    ⊃
    every(λ (t_1: [Address, list[Byte]]):
      (cons?(t_1'2) ∧
      a' ≤ t_1'1 ∧ t_1'1 + length(t_1'2) ≤ a' + length(bl'))
      ∧
      type_of(t_1'1) = type_of(a') ∧
      reg_base(min_linear)(type_of(a')) ≤ t_1'1 ∧
      t_1'1 < reg_size(max_linear)(type_of(a')))
      (split(min_page, a', bl'))
{-3}  every(λ (a_2: Address, bl2: list[Byte]): cons?(bl2))(split(min_page, a', bl')) ∧
    every(λ (a_2: Address, bl2: list[Byte]):
      a' ≤ a_2 ∧ a_2 + length(bl2) ≤ a' + length(bl'))
      (split(min_page, a', bl'))
    ⊃
    every(λ (t: [Address, list[Byte]]):
      cons?(t'2) ∧
      a' ≤ t'1 ∧ t'1 + length(t'2) ≤ a' + length(bl'))
      (split(min_page, a', bl'))
{-4}  is_linear_plain_memory?(pm')
{-5}  (address_block(a', length(bl')) ⊆ (pm'ro_addr ∪ pm'rw_addr))
2254 {-6}  every(λ (e: [Memory_Address_4G, list[Byte]]):
      transformer_invariant?(pm' states,
      singleton(expr_2_super(linear_read_side_effect_in_page(e)))
      (split(min_page, a', bl'))
{-7}  every(λ (a_2: Address, bl2: list[Byte]):
      rem(expt(2, min_page))(offset(a_2)) + length(bl2) ≤ expt(2, min_page))
      (split(min_page, a', bl'))
{-8}  every(λ (t: [Address, list[Byte]]):

```

C.117 Proofs for Linear_Memory_Properties (challenge-linear.pvs)

Using lemma `every_extend` `[[Address, list[Byte]], [Memory_Address_4G, list[Byte]]]`,

we get 3 subgoals:

split_linear_read_side_effects_unchanged_memory.1.1.1.1:

```

{-1} every(λ (t_1: [Address, list[Byte]]):
  cons?(t_1'2) ∧
  a' ≤ t_1'1 ∧
  t_1'1 + length(t_1'2) ≤ a' + length(bl') ∧
  type_of(t_1'1) = type_of(a') ∧
  reg_base(min_linear)(type_of(a')) ≤ t_1'1 ∧
  t_1'1 < reg_size(max_linear)(type_of(a')) ∧
  rem(expt(2, min_page))(offset(t_1'1)) + length(t_1'2) ≤
  expt(2, min_page))
  (split(min_page, a', bl'))
⊃
every(λ (s: [Memory_Address_4G, list[Byte]]):
  cons?(s'2) ∧
  a' ≤ s'1 ∧
  s'1 + length(s'2) ≤ a' + length(bl') ∧
  type_of(s'1) = type_of(a') ∧
  reg_base(min_linear)(type_of(a')) ≤ s'1 ∧
  s'1 < reg_size(max_linear)(type_of(a')) ∧
  rem(expt(2, min_page))(offset(s'1)) + length(s'2) ≤
  expt(2, min_page))
  (split(min_page, a', bl'))
{-2} every(λ (t_1: [Address, list[Byte]]):
  cons?(t_1'2) ∧
  a' ≤ t_1'1 ∧
  t_1'1 + length(t_1'2) ≤ a' + length(bl') ∧
  type_of(t_1'1) = type_of(a') ∧
  reg_base(min_linear)(type_of(a')) ≤ t_1'1 ∧
  t_1'1 < reg_size(max_linear)(type_of(a')))
  (split(min_page, a', bl'))
∧
every(λ (a_2: Address, bl2: list[Byte]):
  rem(expt(2, min_page))(offset(a_2)) + length(bl2) ≤
  expt(2, min_page))
  (split(min_page, a', bl'))
⊃
every(λ (t: [Address, list[Byte]]):
  (cons?(t'2) ∧
  a' ≤ t'1 ∧
  t'1 + length(t'2) ≤ a' + length(bl') ∧
  type_of(t'1) = type_of(a') ∧
  reg_base(min_linear)(type_of(a')) ≤ t'1 ∧
  t'1 < reg_size(max_linear)(type_of(a')))
  ∧
  rem(expt(2, min_page))(offset(t'1)) + length(t'2) ≤
  expt(2, min_page))
  (split(min_page, a', bl'))
{-3} every(λ (t: [Address, list[Byte]]):
  cons?(t'2) ∧
  a' ≤ t'1 ∧ t'1 + length(t'2) ≤ a' + length(bl')
  (split(min_page, a', bl'))
∧
every(λ (t: [Address, list[Byte]]):
  type_of(t'1) = type_of(a') ∧
  reg_base(min_linear)(type_of(a')) ≤ t'1 ∧
  t'1 < reg_size(max_linear)(type_of(a')))
  (split(min_page, a', bl'))
⊃
every(λ (t_1: [Address, list[Byte]]):
  (cons?(t_1'2) ∧
  a' ≤ t_1'1 ∧ t_1'1 + length(t_1'2) ≤ a' + length(bl'))
  ∧
  type_of(t_1'1) = type_of(a') ∧
  reg_base(min_linear)(type_of(a')) ≤ t_1'1 ∧

```

Using lemma `every_conjunct_left`,

we get 3 subgoals:

split_linear_read_side_effects_unchanged_memory.1.1.1.1.1:

```

{-1} every(λ (s: [Memory_Address_4G, list[Byte]]):
  cons?(s'2) ∧
  a' ≤ s'1 ∧
  s'1 + length(s'2) ≤ a' + length(bl') ∧
  type_of(s'1) = type_of(a') ∧
  reg_base(min_linear)(type_of(a')) ≤ s'1 ∧
  s'1 < reg_size(max_linear)(type_of(a')) ∧
  rem(expt(2, min_page))(offset(s'1)) + length(s'2) ≤
  expt(2, min_page))
  (split(min_page, a', bl'))
  ∧
  every(λ (e: [Memory_Address_4G, list[Byte]]):
    transformer_invariant?(pm' states,
      singleton(expr_2_super(linear_read_side_effect_in_page(e))
        (split(min_page, a', bl'))))
    ⊃
    every(λ (t: [Memory_Address_4G, list[Byte]]):
      (cons?(t'2) ∧
        a' ≤ t'1 ∧
        t'1 + length(t'2) ≤ a' + length(bl') ∧
        type_of(t'1) = type_of(a') ∧
        reg_base(min_linear)(type_of(a')) ≤ t'1 ∧
        t'1 < reg_size(max_linear)(type_of(a')) ∧
        rem(expt(2, min_page))(offset(t'1)) + length(t'2) ≤
        expt(2, min_page))
        ∧
        transformer_invariant?(pm' states,
          singleton(expr_2_super(linear_read_side_effect_in_page
            (t))))))
      (split(min_page, a', bl'))
    {-2} every(λ (t_1: [Address, list[Byte]]):
      cons?(t_1'2) ∧
      a' ≤ t_1'1 ∧
      t_1'1 + length(t_1'2) ≤ a' + length(bl') ∧
      type_of(t_1'1) = type_of(a') ∧
      reg_base(min_linear)(type_of(a')) ≤ t_1'1 ∧
      t_1'1 < reg_size(max_linear)(type_of(a')) ∧
      rem(expt(2, min_page))(offset(t_1'1)) + length(t_1'2) ≤
      expt(2, min_page))
      (split(min_page, a', bl'))
      ⊃
      every(λ (s: [Memory_Address_4G, list[Byte]]):
        cons?(s'2) ∧
        a' ≤ s'1 ∧
        s'1 + length(s'2) ≤ a' + length(bl') ∧
        type_of(s'1) = type_of(a') ∧
        reg_base(min_linear)(type_of(a')) ≤ s'1 ∧
        s'1 < reg_size(max_linear)(type_of(a')) ∧
        rem(expt(2, min_page))(offset(s'1)) + length(s'2) ≤
        expt(2, min_page))
        (split(min_page, a', bl'))
      {-3} every(λ (t_1: [Address, list[Byte]]):
        cons?(t_1'2) ∧
        a' ≤ t_1'1 ∧
        t_1'1 + length(t_1'2) ≤ a' + length(bl') ∧
        type_of(t_1'1) = type_of(a') ∧
        reg_base(min_linear)(type_of(a')) ≤ t_1'1 ∧
        t_1'1 < reg_size(max_linear)(type_of(a'))
        (split(min_page, a', bl'))
        ∧
        every(λ (a_2: Address, bl_2: list[Byte]):
          rem(expt(2, min_page))(offset(a_2)) + length(bl_2) ≤
          expt(2, min_page))

```

C.117 Proofs for Linear_Memory_Properties (challenge-linear.pvs)

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Hiding formulas: (-1 -2 -4 -5 -6 -7 -8 -9 -10 -16),

Installing automatic rewrites from: (max_linear! Mem! ##! expr_2_super! expr_2_super_res!)

Using lemma every_implied,

we get 2 subgoals:

split_linear_read_side_effects_unchanged_memory.1.1.1.1.1.1:

2260

```

{-1}  (∀ (t: [Memory_Address_4G, list[Byte]]):
      cons?(t'2) ∧
      a' ≤ t'1 ∧
      t'1 + length(t'2) ≤ a' + length(bl') ∧
      type_of(t'1) = type_of(a') ∧
      reg_base(min_linear)(type_of(a')) ≤ t'1 ∧
      t'1 < reg_size(max_linear)(type_of(a')) ∧
      rem(expt(2, min_page))(offset(t'1)) + length(t'2) ≤
      expt(2, min_page)
      ∧
      transformer_invariant?(pm' states,
                             singleton(expr_2_super(linear_read_side_effect_in_page
                                                       (t))))
      ⊃
      unchanged_memory_invariant?(pm_phy' mem, pm' states,
                                   singleton(expr_2_super(linear_read_side_effect_in_page(t))
                                             (pm_phy'ro_addr ∪ pm_phy'rw_addr)))
      ∧
      every(λ (s: [Memory_Address_4G, list[Byte]]):
            cons?(s'2) ∧
            a' ≤ s'1 ∧
            s'1 + length(s'2) ≤ a' + length(bl') ∧
            type_of(s'1) = type_of(a') ∧
            reg_base(min_linear)(type_of(a')) ≤ s'1 ∧
            s'1 < reg_size(max_linear)(type_of(a')) ∧
            rem(expt(2, min_page))(offset(s'1)) + length(s'2) ≤
            expt(2, min_page)
            ∧
            transformer_invariant?(pm' states,
                                   singleton(expr_2_super(linear_read_side_effect_in_page
                                                           (s))))
            (split(min_page, a', bl')))
      ⊃
      every(λ (e: [Memory_Address_4G, list[Byte]]):
            unchanged_memory_invariant?(pm_phy' mem, pm' states,
                                         singleton(expr_2_super(linear_read_side_effect_in_page
                                                                   (e)),
                                                   (pm_phy'ro_addr ∪ pm_phy'rw_addr)))
            (split(min_page, a', bl')))
{-2}  every(λ (t: [Memory_Address_4G, list[Byte]]):
      (cons?(t'2) ∧
       a' ≤ t'1 ∧
       t'1 + length(t'2) ≤ a' + length(bl') ∧
       type_of(t'1) = type_of(a') ∧
       reg_base(min_linear)(type_of(a')) ≤ t'1 ∧
       t'1 < reg_size(max_linear)(type_of(a')) ∧
       rem(expt(2, min_page))(offset(t'1)) + length(t'2) ≤
       expt(2, min_page))
      ∧
      transformer_invariant?(pm' states,
                             singleton(expr_2_super(linear_read_side_effect_in_page(t))
                                             (pm_phy'ro_addr ∪ pm_phy'rw_addr)))
      (split(min_page, a', bl')))
{-3}  is_linear_plain_memory?(pm')
{-4}  (address_block(a', length(bl')) ⊆ (pm'ro_addr ∪ pm'rw_addr))
{-5}  Mem?(a' type_of)
{-6}  0 ≤ a' offset
{-7}  a' offset < max_linear_offset
{-8}  cons?[Byte](bl')
{1}  every(λ (e: [Memory_Address_4G, list[Byte]]):
      unchanged_memory_invariant?(pm_phy' mem, pm' states,
                                   singleton(expr_2_super(linear_read_side_effect_in_page
                                                           (e)),
                                                   (pm_phy'ro_addr ∪ pm_phy'rw_addr)))
      )

```

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Hiding formulas: (-1 -2 2),

Expanding the definition of linear_read_side_effect_in_page,

Repeatedly Skolemizing and flattening,

Case splitting on union(pm!1'ro_addr, pm!1'rw_addr)(t!1'1),

we get 2 subgoals:

split_linear_read_side_effects_unchanged_memory.1.1.1.1.1.1.1.1:

<pre> {-1} union(pm'ro_addr, pm'rw_addr)(t'1) {-2} cons?(t'2) {-3} a' ≤ t'1 {-4} t'1 + length(t'2) ≤ a' + length(bl') {-5} type_of(t'1) = type_of(a') {-6} reg_base(min_linear)(type_of(a')) ≤ t'1 {-7} t'1 < reg_size((#type_of := Mem_, offset := max_linear_offset#))(type_of(a')) {-8} rem(expt(2, min_page))(offset(t'1)) + length(t'2) ≤ expt(2, min_page) {-9} transformer_invariant?(pm'states, singleton(expr_2_super(λ (s: Lin- ear_memory[Physical_memory, pm_phy]): ory_Address_4G) ory_read_side_effect(pm_phy'mem) (s)))) {-10} (address_block(a', length(bl')) ⊆ (pm'ro_addr ∪ pm'rw_addr)) {-11} Mem?(a' type_of) {-12} 0 ≤ a'offset {-13} a'offset < max_linear_offset {-14} cons?[Byte](bl')</pre>	<pre> (linear_resolve(t'1, Read) ## (λ (pa: Mem- (ns: Linear_memory [Physical_memory, pm_phy]): mem- (pa, t'2, TRUE)(s))) (s))))</pre>
<pre> {1} unchanged_memory_invariant?(pm_phy'mem, pm'states, singleton(expr_2_super(λ (s: Linear_memory [Physical_memory, pm_phy]): (linear_resolve(t'1, Read) ## (λ (pa: Mem- (ns: Linear_memory [Physical_memory, pm_phy]): mem- (pa, t'2, TRUE)(s))) (s))), (pm_phy'ro_addr ∪ pm_phy'rw_addr))</pre>	<pre> (s))))</pre>

Using lemma pm_states,

C Proof scripts

Using lemma `pm_plain_phy`,

Expanding the definition of `unchanged_memory_invariant?`,

Simplifying, rewriting, and recording with decision procedures,

Repeatedly Skolemizing and flattening,

Expanding the definition of `singleton`,

Replacing using formula -13,

Hiding formulas: -13,

Using lemma `pm_linear_resolve_read_ok`,

Case splitting on `subset?(address_block(data(linear_resolve(t!1, Read)(s!1)), length(t!1'2)), union(pm_phy'ro_addr, pm_phy'rw_addr))`,

we get 3 subgoals:

split_linear_read_side_effects_unchanged_memory.1.1.1.1.1.1.1.1:

```

{-1}  (address_block(data(linear_resolve(t'1, Read)(s')), length(t'2)) ⊆ (pm_phy'ro_addr ∪ pm_phy'rw_addr))
{-2}  is_linear_plain_memory?(pm') ⊃ OK?(linear_resolve(t'1, Read)(s'))
{-3}  plain_memory?(pm_phy)
{-4}  pm' states = pm_phy states
{-5}  union(pm' ro_addr, pm' rw_addr)(t'1)
{-6}  cons?(t'2)
{-7}  a' ≤ t'1
{-8}  t'1 + length(t'2) ≤ a' + length(bl')
{-9}  type_of(t'1) = type_of(a')
{-10} reg_base(min_linear)(type_of(a')) ≤ t'1
{-11} t'1 < reg_size((#type_of := Mem_, offset := max_linear_offset#))(type_of(a'))
{-12} rem(expt(2, min_page))(offset(t'1) + length(t'2)) ≤ expt(2, min_page)
{-13} transformer_invariant?(pm' states,
      {y |
        y =
          expr_2_super(λ (s: Linear_memory[Physical_memory, pm_phy]):
            CASES lin-
ear_resolve(t'1, Read)(s) OF
          OK(state, data):
            mem-
ory_read_side_effect(pm_phy mem)
            (data, t'2, TRUE)(s),
          Exception(ex_type, state):
            Exception(ex_type, state),
          Fatal: Fatal,
          Hang: Hang
          ENDCASES)})
{-14} pm' states(s')
{-15} union(pm_phy' ro_addr, pm_phy' rw_addr)(a'')
{-16} OK?(expr_2_super(λ (s: Linear_memory[Physical_memory, pm_phy]):
      CASES linear_resolve(t'1, Read)(s) OF
        OK(state, data):
          memory_read_side_effect(pm_phy mem)(data, t'2, TRUE)(s),
          Exception(ex_type, state): Exception(ex_type, state),
          Fatal: Fatal,
          Hang: Hang
          ENDCASES)
      (s'))
{-17} OK?(memory_read(pm_phy mem)(a'')(s'))
{-18} OK?(memory_read(pm_phy mem)
      (a'')
      (state(expr_2_super(λ (s: Linear_memory[Physical_memory, pm_phy]):
        CASES linear_resolve(t'1, Read)(s) OF
          OK(state, data):
            memory_read_side_effect(pm_phy mem)
              (data, t'2, TRUE)(s),
          Exception(ex_type, state): Excep-
tion(ex_type, state),
          Fatal: Fatal,
          Hang: Hang
          ENDCASES)
      (s')))))
{-19} (address_block(a', length(bl')) ⊆ (pm' ro_addr ∪ pm' rw_addr))
{-20} Mem?(a' type_of)
{-21} 0 ≤ a' offset
{-22} a' offset < max_linear_offset
{-23} cons?[Byte](bl')
{1}  data(memory_read(pm_phy mem)
      (a'')
      (state(expr_2_super(λ (s: Linear_memory[Physical_memory, pm_phy]):
        CASES linear_resolve(t'1, Read)(s) OF
          OK(state, data):
            memory_read_side_effect(pm_phy mem)

```


C Proof scripts

Using lemma `linear_resolve_states`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Using lemma `plain_memory_unchanged_invariant`,

Using lemma `unchanged_memory_invariant_unchanged`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Case splitting on `memory_read_side_effect_super_transformers(pm_phy'mem, union (pm_phy'ro_addr, pm_phy'rw_addr)) (expr_2_super (memory_read_side_effect (pm_phy'mem) (data (linear_resolve (t!1'1, Read)(s!1)), t!1'2, TRUE)))`,

we get 2 subgoals:

split_linear_read_side_effects_unchanged_memory.1.1.1.1.1.1.1.1.1:

```

{-1}  memory_read_side_effect_super_transformers(pm_phy' mem, (pm_phy'ro_addr ∪ pm_phy'rw_addr))
                                             (expr_2_super(memory_read_side_effect(pm_phy' mem)
                                             (data
                                             (linear_resolve
                                             (t'1, Read)(s')),
                                             t'2,
                                             TRUE)))
{-2}  unchanged_memory_invariant?(pm_phy' mem, pm_phy' states,
                                   ((pm_phy'other_actions ∪ memory_read_transformers(pm_phy' mem, (pm_phy'ro_addr ∪ pm_phy'rw_addr))
                                   (pm_phy'ro_addr ∪ pm_phy'rw_addr))
{-3}  union(pm_phy'ro_addr, pm_phy'rw_addr)(a'')
{-4}  OK?(memory_read(pm_phy' mem)(a'')(s'))
{-5}  plain_memory?(pm_phy)
{-6}  union(pm'ro_addr, pm'rw_addr)(t'1)
{-7}  pm' states(s')
{-8}  OK?(linear_resolve(t'1, Read)(s'))
{-9}  pm_phy' states(state(linear_resolve(t'1, Read)(s')))
{-10} (address_block(data(linear_resolve(t'1, Read)(s')), length(t'2)) ⊆ (pm_phy'ro_addr ∪ pm_phy'rw_addr))
{-11} pm' states = pm_phy' states
{-12} cons?(t'2)
{-13} a' ≤ t'1
{-14} t'1 + length(t'2) ≤ a' + length(bl')
{-15} type_of(t'1) = type_of(a')
{-16} reg_base(min_linear)(type_of(a')) ≤ t'1
{-17} t'1 < reg_size((#type_of := Mem_, offset := max_linear_offset#))(type_of(a'))
{-18} rem(expt(2, min_page))(offset(t'1)) + length(t'2) ≤ expt(2, min_page)
{-19} transformer_invariant?(pm' states,
                              {y |
                              y =
                              expr_2_super(λ (s : Linear_memory[Physical_memory, pm_phy]):
                              CASES lin-
ear_resolve(t'1, Read)(s) OF
                              OK(state, data):
                              mem-
ory_read_side_effect(pm_phy' mem)
                              (data, t'2, TRUE)(s),
                              Exception(ex_type, state):
                              Exception(ex_type, state),
                              Fatal: Fatal,
                              Hang: Hang
                              ENDCASES)})
{-20} OK?(memory_read_side_effect(pm_phy' mem)
          (data(linear_resolve(t'1, Read)(s')), t'2, TRUE)(s'))
{-21} OK?(OK(state(memory_read_side_effect(pm_phy' mem)
          (data(linear_resolve(t'1, Read)(s')), t'2, TRUE)(s'))))
{-22} OK?(memory_read(pm_phy' mem)
          (a'')
          (state(memory_read_side_effect(pm_phy' mem)
          (data(linear_resolve(t'1, Read)(s')), t'2, TRUE)(s'))))
{-23} (address_block(a', length(bl')) ⊆ (pm'ro_addr ∪ pm'rw_addr))
{-24} Mem?(a' type_of)
{-25} 0 ≤ a' offset
{-26} a' offset < max_linear_offset
{-27} cons?[Byte](bl')
-----
{1}  union((pm_phy'other_actions ∪ memory_read_transformers(pm_phy' mem, (pm_phy'ro_addr ∪ pm_phy'rw_addr))
          (memory_read_side_effect_super_transformers(pm_phy' mem, (pm_phy'ro_addr ∪ pm_phy'rw_addr))
          (expr_2_super(memory_read_side_effect(pm_phy' mem)
          (data(linear_resolve(t'1, Read)(s')), t'2, TRUE)))
{2}  data(memory_read(pm_phy' mem)
          (a'')
          (state(memory_read_side_effect(pm_phy' mem)
          (data(linear_resolve(t'1, Read)(s')), t'2, TRUE)(s'))))
      = data(memory_read(pm_phy' mem)(a'')(s'))

```

C Proof scripts

Keeping (-1 1) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `split_linear_read_side_effects_unchanged_memory.1.1.1.1.1.1.1.1.1.`

split_linear_read_side_effects_unchanged_memory.1.1.1.1.1.1.1.1.2:

```

{-1}  unchanged_memory_invariant?(pm_phy' mem, pm_phy' states,
                                     ((pm_phy'other_actions ∪ memory_read_transformers(pm_phy' mem, (pm_phy'ro_addr ∪ pm_phy'rw_addr))
                                     (pm_phy'ro_addr ∪ pm_phy'rw_addr))
{-2}  union(pm_phy'ro_addr, pm_phy'rw_addr)(a'')
{-3}  OK?(memory_read(pm_phy' mem)(a'')(s'))
{-4}  plain_memory?(pm_phy)
{-5}  union(pm'ro_addr, pm'rw_addr)(t'1)
{-6}  pm'states(s')
{-7}  OK?(linear_resolve(t'1, Read)(s'))
{-8}  pm_phy'states(state(linear_resolve(t'1, Read)(s')))
{-9}  (address_block(data(linear_resolve(t'1, Read)(s')), length(t'2)) ⊆ (pm_phy'ro_addr ∪ pm_phy'rw_addr))
{-10} pm'states = pm_phy'states
{-11} cons?(t'2)
{-12} a' ≤ t'1
{-13} t'1 + length(t'2) ≤ a' + length(bl')
{-14} type_of(t'1) = type_of(a')
{-15} reg_base(min_linear)(type_of(a')) ≤ t'1
{-16} t'1 < reg_size((#type_of := Mem_, offset := max_linear_offset#))(type_of(a'))
{-17} rem(expt(2, min_page))(offset(t'1)) + length(t'2) ≤ expt(2, min_page)
{-18} transformer_invariant?(pm'states,
                               {y |
                                 y =
                                   expr_2_super(λ (s: Linear_memory[Physical_memory, pm_phy]):
                                                 CASES lin-
ear_resolve(t'1, Read)(s) OF
                                     OK(state, data):
                                     mem-
emory_read_side_effect(pm_phy' mem)
                                     (data, t'2, TRUE)(s),
                                     Exception(ex_type, state):
                                     Exception(ex_type, state),
                                     Fatal: Fatal,
                                     Hang: Hang
                                     ENDCASES)})
{-19} OK?(memory_read_side_effect(pm_phy' mem)
          (data(linear_resolve(t'1, Read)(s')), t'2, TRUE)(s'))
{-20} OK?(OK(state(memory_read_side_effect(pm_phy' mem)
          (data(linear_resolve(t'1, Read)(s')), t'2, TRUE)(s'))))
{-21} OK?(memory_read(pm_phy' mem)
          (a'')
          (state(memory_read_side_effect(pm_phy' mem)
          (data(linear_resolve(t'1, Read)(s')), t'2, TRUE)(s'))))
{-22} (address_block(a', length(bl')) ⊆ (pm'ro_addr ∪ pm'rw_addr))
{-23} Mem?(a' type_of)
{-24} 0 ≤ a'offset
{-25} a'offset < max_linear_offset
{-26} cons?[Byte](bl')
-----
{1}  memory_read_side_effect_super_transformers(pm_phy' mem, (pm_phy'ro_addr ∪ pm_phy'rw_addr))
      (expr_2_super(memory_read_side_effect(pm_phy' mem)
                    (data
                      (linear_resolve
                        (t'1, Read)(s')),
                        t'2,
                        TRUE)))
{2}  union((pm_phy'other_actions ∪ memory_read_transformers(pm_phy' mem, (pm_phy'ro_addr ∪ pm_phy'rw_addr)) ∪
          (memory_read_side_effect_super_transformers(pm_phy' mem, (pm_phy'ro_addr ∪ pm_phy'rw_addr)) ∪
          (expr_2_super(memory_read_side_effect(pm_phy' mem)
                    (data(linear_resolve(t'1, Read)(s')), t'2, TRUE))))))
{3}  data(memory_read(pm_phy' mem)
          (a'')
          (state(memory_read_side_effect(pm_phy' mem)
          (data(linear_resolve(t'1, Read)(s')), t'2, TRUE)(s'))))
      = data(memory_read(pm_phy' mem)(a'')(s'))

```

C Proof scripts

Hiding formulas: (-1 -11 2 3),

Expanding the definition of `memory_read_side_effect_super_transformers`,

Instantiating quantified variables,

This completes the proof of `split_linear_read_side_effects_unchanged_memory.1.1.1.1.1.1.1.1.2`.

split_linear_read_side_effects_unchanged_memory.1.1.1.1.1.1.2:

```

{-1}  is_linear_plain_memory?(pm') ⊃ OK?(linear_resolve(t'1, Read)(s'))
{-2}  plain_memory?(pm_phy)
{-3}  pm' states = pm_phy states
{-4}  union(pm' ro_addr, pm' rw_addr)(t'1)
{-5}  cons?(t'2)
{-6}  a' ≤ t'1
{-7}  t'1 + length(t'2) ≤ a' + length(bl')
{-8}  type_of(t'1) = type_of(a')
{-9}  reg_base(min_linear)(type_of(a')) ≤ t'1
{-10} t'1 < reg_size((#type_of := Mem_, offset := max_linear_offset#))(type_of(a'))
{-11} rem(expt(2, min_page))(offset(t'1) + length(t'2)) ≤ expt(2, min_page)
{-12} transformer_invariant?(pm' states,
      {y |
        y =
          expr_2_super(λ (s: Linear_memory[Physical_memory, pm_phy]):
            CASES lin-
ear_resolve(t'1, Read)(s) OF
      OK(state, data):
        mem-
ory_read_side_effect(pm_phy mem)
          (data, t'2, TRUE)(s),
        Exception(ex_type, state):
          Exception(ex_type, state),
        Fatal: Fatal,
        Hang: Hang
        ENDCASES})
{-13} pm' states(s')
{-14} union(pm_phy ro_addr, pm_phy rw_addr)(a'')
{-15} OK?(expr_2_super(λ (s: Linear_memory[Physical_memory, pm_phy]):
      CASES linear_resolve(t'1, Read)(s) OF
        OK(state, data):
          memory_read_side_effect(pm_phy mem)(data, t'2, TRUE)(s),
          Exception(ex_type, state): Exception(ex_type, state),
          Fatal: Fatal,
          Hang: Hang
          ENDCASES)
      (s'))
{-16} OK?(memory_read(pm_phy mem)(a'')(s'))
{-17} OK?(memory_read(pm_phy mem)
      (a'')
      (state(expr_2_super(λ (s: Linear_memory[Physical_memory, pm_phy]):
        CASES linear_resolve(t'1, Read)(s) OF
          OK(state, data):
            memory_read_side_effect(pm_phy mem)
              (data, t'2, TRUE)(s),
            Exception(ex_type, state): Excep-
tion(ex_type, state),
          Fatal: Fatal,
          Hang: Hang
          ENDCASES)
        (s'))))
{-18} (address_block(a', length(bl')) ⊆ (pm' ro_addr ∪ pm' rw_addr))
{-19} Mem?(a' type_of)
{-20} 0 ≤ a' offset
{-21} a' offset < max_linear_offset
{-22} cons?[Byte](bl')

```

```

{1}  (address_block(data(linear_resolve(t'1, Read)(s')), length(t'2)) ⊆ (pm_phy ro_addr ∪ pm_phy rw_addr))
{2}  data(memory_read(pm_phy mem)
      (a'')
      (state(expr_2_super(λ (s: Linear_memory[Physical_memory, pm_phy]):
        CASES linear_resolve(t'1, Read)(s) OF
          OK(state, data):
            memory_read_side_effect(pm_phy mem)

```

C Proof scripts

Hiding formulas: 2,

Using lemma `same_page_address_block_read`,

we get 3 subgoals:

2270

split_linear_read_side_effects_unchanged_memory.1.1.1.1.1.1.1.2.1:

```

{-1}  is_linear_plain_memory?(pm') ∧
      pm' states(s') ∧
      union(pm' ro_addr, pm' rw_addr)(t'1) ∧
      a' ≤ t'1 ∧
      t'1 + length(t'2) ≤ a' + length(bl') ∧
      rem(expt(2, min_page))(offset(t'1)) + length(t'2) ≤ expt(2, min_page)
      ∧ (address_block(a', length(bl')) ⊆ (pm' ro_addr ∪ pm' rw_addr))
      ⊃
      (address_block(data(linear_resolve(t'1, Read)(s')), length(t'2)) ⊆ (pm_phy ro_addr ∪ pm_phy rw_addr))
{-2}  is_linear_plain_memory?(pm') ⊃ OK?(linear_resolve(t'1, Read)(s'))
{-3}  plain_memory?(pm_phy)
{-4}  pm' states = pm_phy states
{-5}  union(pm' ro_addr, pm' rw_addr)(t'1)
{-6}  cons?(t'2)
{-7}  a' ≤ t'1
{-8}  t'1 + length(t'2) ≤ a' + length(bl')
{-9}  type_of(t'1) = type_of(a')
{-10} reg_base(min_linear)(type_of(a')) ≤ t'1
{-11} t'1 < reg_size((#type_of := Mem_, offset := max_linear_offset#))(type_of(a'))
{-12} rem(expt(2, min_page))(offset(t'1)) + length(t'2) ≤ expt(2, min_page)
{-13} transformer_invariant?(pm' states,
                               {y |
                                 y =
                                   expr_2_super(λ (s: Linear_memory[Physical_memory, pm_phy]):
                                                 CASES lin-
ear_resolve(t'1, Read)(s) OF
                                     OK(state, data):
                                     mem-
ory_read_side_effect(pm_phy mem)
                                     (data, t'2, TRUE)(s),
                                     Exception(ex_type, state):
                                     Exception(ex_type, state),
                                     Fatal: Fatal,
                                     Hang: Hang
                                     ENDCASES)})
{-14} pm' states(s')
{-15} union(pm_phy ro_addr, pm_phy rw_addr)(a'')
{-16} OK?(expr_2_super(λ (s: Linear_memory[Physical_memory, pm_phy]):
                       CASES linear_resolve(t'1, Read)(s) OF
                               OK(state, data):
                               memory_read_side_effect(pm_phy mem)(data, t'2, TRUE)(s),
                               Exception(ex_type, state): Exception(ex_type, state),
                               Fatal: Fatal,
                               Hang: Hang
                               ENDCASES)
                       (s'))
{-17} OK?(memory_read(pm_phy mem)(a'')(s'))
{-18} OK?(memory_read(pm_phy mem)
                    (a'')
                    (state(expr_2_super(λ (s: Linear_memory[Physical_memory, pm_phy]):
                                         CASES linear_resolve(t'1, Read)(s) OF
                                                 OK(state, data):
                                                 memory_read_side_effect(pm_phy mem)
                                                 (data, t'2, TRUE)(s),
                                                 Exception(ex_type, state): Excep-
tion(ex_type, state),
                                         Fatal: Fatal,
                                         Hang: Hang
                                         ENDCASES)
                                         (s'))))
{-19} (address_block(a', length(bl')) ⊆ (pm' ro_addr ∪ pm' rw_addr))
{-20} Mem?(a' type_of)
{-21} 0 ≤ a' offset

```


C Proof scripts

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `split_linear_read_side_effects_unchanged_memory.1.1.1.1.1.1.1.2.1`.

split_linear_read_side_effects_unchanged_memory.1.1.1.1.1.1.2.2:

```

{-1}  is_linear_plain_memory?(pm')  $\supset$  OK?(linear_resolve(t'1, Read)(s'))
{-2}  plain_memory?(pm_phy)
{-3}  pm' states = pm_phy states
{-4}  union(pm' ro_addr, pm' rw_addr)(t'1)
{-5}  cons?(t'2)
{-6}   $a' \leq t'1$ 
{-7}   $t'1 + \text{length}(t'2) \leq a' + \text{length}(bl')$ 
{-8}  type_of(t'1) = type_of(a')
{-9}  reg_base(min_linear)(type_of(a'))  $\leq t'1$ 
{-10}  $t'1 < \text{reg\_size}(\# \text{type\_of} := \text{Mem\_}, \text{offset} := \text{max\_linear\_offset\#})(\text{type\_of}(a'))$ 
{-11}  $\text{rem}(\text{expt}(2, \text{min\_page}))(\text{offset}(t'1) + \text{length}(t'2)) \leq \text{expt}(2, \text{min\_page})$ 
{-12} transformer_invariant?(pm' states,
      {y |
        y =
          expr_2_super( $\lambda$  (s: Linear_memory[Physical_memory, pm_phy]):
            CASES lin-
ear_resolve(t'1, Read)(s) OF
      OK(state, data):
        mem-
ory_read_side_effect(pm_phy mem)
          (data, t'2, TRUE)(s),
        Exception(ex_type, state):
          Exception(ex_type, state),
        Fatal: Fatal,
        Hang: Hang
        ENDCASES})
{-13} pm' states(s')
{-14} union(pm_phy ro_addr, pm_phy rw_addr)(a'')
{-15} OK?(expr_2_super( $\lambda$  (s: Linear_memory[Physical_memory, pm_phy]):
      CASES linear_resolve(t'1, Read)(s) OF
        OK(state, data):
          memory_read_side_effect(pm_phy mem)(data, t'2, TRUE)(s),
        Exception(ex_type, state): Exception(ex_type, state),
        Fatal: Fatal,
        Hang: Hang
        ENDCASES)
      (s'))
{-16} OK?(memory_read(pm_phy mem)(a'')(s'))
{-17} OK?(memory_read(pm_phy mem)
      (a'')
      (state(expr_2_super( $\lambda$  (s: Linear_memory[Physical_memory, pm_phy]):
        CASES linear_resolve(t'1, Read)(s) OF
          OK(state, data):
            memory_read_side_effect(pm_phy mem)
              (data, t'2, TRUE)(s),
          Exception(ex_type, state): Excep-
tion(ex_type, state),
          Fatal: Fatal,
          Hang: Hang
          ENDCASES)
        (s'))))
{-18} (address_block(a', length(bl'))  $\subseteq$  (pm' ro_addr  $\cup$  pm' rw_addr))
{-19} Mem?(a' type_of)
{-20}  $0 \leq a'$  offset
{-21}  $a'$  offset < max_linear_offset
{-22} cons?[Byte](bl')

```

```

{1}  length[Byte](t'2) > 0
{2}  (address_block(data(linear_resolve(t'1, Read)(s')), length(t'2))  $\subseteq$  (pm_phy ro_addr  $\cup$  pm_phy rw_addr))

```

C Proof scripts

Expanding the definition of length,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `split_linear_read_side_effects_unchanged_memory.1.1.1.1.1.1.2.2`.

split_linear_read_side_effects_unchanged_memory.1.1.1.1.1.1.1.2.3:

```

{-1}  is_linear_plain_memory?(pm')  $\supset$  OK?(linear_resolve(t'1, Read)(s'))
{-2}  plain_memory?(pm_phy)
{-3}  pm' states = pm_phy states
{-4}  union(pm' ro_addr, pm' rw_addr)(t'1)
{-5}  cons?(t'2)
{-6}   $a' \leq t'1$ 
{-7}   $t'1 + \text{length}(t'2) \leq a' + \text{length}(bl')$ 
{-8}  type_of(t'1) = type_of(a')
{-9}  reg_base(min_linear)(type_of(a'))  $\leq t'1$ 
{-10}  $t'1 < \text{reg\_size}(\# \text{type\_of} := \text{Mem\_}, \text{offset} := \text{max\_linear\_offset\#})(\text{type\_of}(a'))$ 
{-11}  $\text{rem}(\text{expt}(2, \text{min\_page}))(\text{offset}(t'1) + \text{length}(t'2)) \leq \text{expt}(2, \text{min\_page})$ 
{-12} transformer_invariant?(pm' states,
      {y |
        y =
          expr_2_super( $\lambda$  (s: Linear_memory[Physical_memory, pm_phy]):
            CASES lin-
ear_resolve(t'1, Read)(s) OF
      OK(state, data):
        mem-
ory_read_side_effect(pm_phy mem)
          (data, t'2, TRUE)(s),
        Exception(ex_type, state):
          Exception(ex_type, state),
        Fatal: Fatal,
        Hang: Hang
        ENDCASES)})
{-13} pm' states(s')
{-14} union(pm_phy ro_addr, pm_phy rw_addr)(a'')
{-15} OK?(expr_2_super( $\lambda$  (s: Linear_memory[Physical_memory, pm_phy]):
      CASES linear_resolve(t'1, Read)(s) OF
        OK(state, data):
          memory_read_side_effect(pm_phy mem)(data, t'2, TRUE)(s),
        Exception(ex_type, state): Exception(ex_type, state),
        Fatal: Fatal,
        Hang: Hang
        ENDCASES)
      (s'))
{-16} OK?(memory_read(pm_phy mem)(a'')(s'))
{-17} OK?(memory_read(pm_phy mem)
      (a'')
      (state(expr_2_super( $\lambda$  (s: Linear_memory[Physical_memory, pm_phy]):
        CASES linear_resolve(t'1, Read)(s) OF
          OK(state, data):
            memory_read_side_effect(pm_phy mem)
              (data, t'2, TRUE)(s),
          Exception(ex_type, state): Excep-
tion(ex_type, state),
          Fatal: Fatal,
          Hang: Hang
          ENDCASES)
        (s'))))
{-18} (address_block(a', length(bl'))  $\subseteq$  (pm' ro_addr  $\cup$  pm' rw_addr))
{-19} Mem?(a' type_of)
{-20}  $0 \leq a'$  offset
{-21}  $a'$  offset  $<$  max_linear_offset
{-22} cons?[Byte](bl')

```

```

{1}  length[Byte](bl')  $>$  0
{2}  (address_block(data(linear_resolve(t'1, Read)(s')), length(t'2))  $\subseteq$  (pm_phy ro_addr  $\cup$  pm_phy rw_addr))

```

C Proof scripts

Expanding the definition of length,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `split_linear_read_side_effects_unchanged_memory.1.1.1.1.1.1.2.3`.

split_linear_read_side_effects_unchanged_memory.1.1.1.1.1.1.1.3:

```

{-1} is_linear_plain_memory?(pm')  $\supset$  OK?(linear_resolve(t'1, Read)(s'))
{-2} plain_memory?(pm_phy)
{-3} pm' states = pm_phy states
{-4} union(pm' ro_addr, pm' rw_addr)(t'1)
{-5} cons?(t'2)
{-6}  $a' \leq t'1$ 
{-7}  $t'1 + \text{length}(t'2) \leq a' + \text{length}(bl')$ 
{-8} type_of(t'1) = type_of(a')
{-9} reg_base(min_linear)(type_of(a'))  $\leq t'1$ 
{-10}  $t'1 < \text{reg\_size}(\# \text{type\_of} := \text{Mem\_}, \text{offset} := \text{max\_linear\_offset\#})(\text{type\_of}(a'))$ 
{-11}  $\text{rem}(\text{expt}(2, \text{min\_page}))(\text{offset}(t'1) + \text{length}(t'2)) \leq \text{expt}(2, \text{min\_page})$ 
{-12} transformer_invariant?(pm' states,
      {y |
        y =
          expr_2_super( $\lambda$  (s: Linear_memory[Physical_memory, pm_phy]):
            CASES lin-
ear_resolve(t'1, Read)(s) OF
      OK(state, data):
        mem-
ory_read_side_effect(pm_phy mem)
          (data, t'2, TRUE)(s),
        Exception(ex_type, state):
          Exception(ex_type, state),
        Fatal: Fatal,
        Hang: Hang
        ENDCASES)})
{-13} pm' states(s')
{-14} union(pm_phy ro_addr, pm_phy rw_addr)(a'')
{-15} OK?(expr_2_super( $\lambda$  (s: Linear_memory[Physical_memory, pm_phy]):
      CASES linear_resolve(t'1, Read)(s) OF
        OK(state, data):
          memory_read_side_effect(pm_phy mem)(data, t'2, TRUE)(s),
        Exception(ex_type, state): Exception(ex_type, state),
        Fatal: Fatal,
        Hang: Hang
        ENDCASES)
      (s'))
{-16} OK?(memory_read(pm_phy mem)(a'')(s'))
{-17} OK?(memory_read(pm_phy mem)
      (a'')
      (state(expr_2_super( $\lambda$  (s: Linear_memory[Physical_memory, pm_phy]):
        CASES linear_resolve(t'1, Read)(s) OF
          OK(state, data):
            memory_read_side_effect(pm_phy mem)
              (data, t'2, TRUE)(s),
          Exception(ex_type, state): Excep-
tion(ex_type, state),
          Fatal: Fatal,
          Hang: Hang
          ENDCASES)
        (s'))))
{-18} (address_block(a', length(bl'))  $\subseteq$  (pm' ro_addr  $\cup$  pm' rw_addr))
{-19} Mem?(a' type_of)
{-20}  $0 \leq a' \text{ offset}$ 
{-21}  $a' \text{ offset} < \text{max\_linear\_offset}$ 
{-22} cons?[Byte](bl')

```

```

{1} OK?[Linear_memory[Physical_memory, pm_phy], Address]
      (linear_resolve[Physical_memory, pm_phy](t'1, Read)(s'))
{2} data(memory_read(pm_phy mem)
      (a'')
      (state(expr_2_super( $\lambda$  (s: Linear_memory[Physical_memory, pm_phy]):
        CASES linear_resolve(t'1, Read)(s) OF
          OK(state, data):

```

C Proof scripts

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `split_linear_read_side_effects_unchanged_memory.1.1.1.1.1.1.1.3`.

`split_linear_read_side_effects_unchanged_memory.1.1.1.1.1.1.2`:

<pre> {-1} cons?(t'2) {-2} a' ≤ t'1 {-3} t'1 + length(t'2) ≤ a' + length(bl') {-4} type_of(t'1) = type_of(a') {-5} reg_base(min_linear)(type_of(a')) ≤ t'1 {-6} t'1 < reg_size((#type_of := Mem_, offset := max_linear_offset#))(type_of(a')) {-7} rem(expt(2, min_page))(offset(t'1)) + length(t'2) ≤ expt(2, min_page) {-8} transformer_invariant?(pm' states, singleton(expr_2_super(λ (s: </pre>	<pre> Lin- ear_memory[Physical_memory, pm_phy]: (linear_resolve(t'1, Read) ## (λ (pa: Mem- </pre>
<pre> ory_Address_4G) </pre>	<pre> (ns: Linear_memory [Physical_memory, pm_ mem- </pre>
<pre> ory_read_side_effect(pm_phy mem) </pre>	<pre> (pa, t'2, TRUE)(s))) (s))) </pre>
<pre> {-9} (address_block(a', length(bl')) ⊆ (pm'ro_addr ∪ pm'rw_addr)) {-10} Mem?(a' type_of) {-11} 0 ≤ a' offset {-12} a' offset < max_linear_offset {-13} cons?[Byte](bl')</pre>	
<pre> {1} union(pm'ro_addr, pm'rw_addr)(t'1) {2} unchanged_memory_invariant?(pm_phy mem, pm' states, singleton(expr_2_super(λ (s: </pre>	<pre> Linear_memory [Physical_memory, pm_ (linear_resolve(t'1, Read) ## (λ (pa: Mem- </pre>
<pre> ory_Address_4G) </pre>	<pre> (ns: Linear_memory [Physical_memory, pm_ mem- </pre>
<pre> ory_read_side_effect(pm_phy mem) </pre>	<pre> (pa, t'2, TRUE)(s))) (s))), (pm_phy ro_addr ∪ pm_phy rw_addr)) </pre>

Hiding formulas: (-8 -7 2),

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `split_linear_read_side_effects_unchanged_memory.1.1.1.1.1.1.2`.

split_linear_read_side_effects_unchanged_memory.1.1.1.1.1.2:

<p>{-1}</p>	$\begin{aligned} & \text{every}(\lambda (t: [\text{Memory_Address_4G}, \text{list}[\text{Byte}]]): \\ & \quad (\text{cons?}(t'2) \wedge \\ & \quad \quad a' \leq t'1 \wedge \\ & \quad \quad t'1 + \text{length}(t'2) \leq a' + \text{length}(bl') \wedge \\ & \quad \quad \text{type_of}(t'1) = \text{type_of}(a') \wedge \\ & \quad \quad \text{reg_base}(\text{min_linear})(\text{type_of}(a')) \leq t'1 \wedge \\ & \quad \quad t'1 < \text{reg_size}(\text{max_linear})(\text{type_of}(a')) \wedge \\ & \quad \quad \text{rem}(\text{expt}(2, \text{min_page}))(\text{offset}(t'1)) + \text{length}(t'2) \leq \\ & \quad \quad \text{expt}(2, \text{min_page})) \\ & \quad \wedge \\ & \quad \text{transformer_invariant?}(\text{pm}'\text{states}, \\ & \quad \quad \text{singleton}(\text{expr_2_super}(\text{linear_read_side_effect_in_page}(t)))) \\ & \quad (\text{split}(\text{min_page}, a', bl')) \end{aligned}$
<p>{-2}</p>	$\text{is_linear_plain_memory?}(\text{pm}')$
<p>{-3}</p>	$(\text{address_block}(a', \text{length}(bl')) \subseteq (\text{pm}'\text{ro_addr} \cup \text{pm}'\text{rw_addr}))$
<p>{-4}</p>	$\text{Mem?}(a'\text{'type_of})$
<p>{-5}</p>	$0 \leq a'\text{'offset}$
<p>{-6}</p>	$a'\text{'offset} < \text{max_linear_offset}$
<p>{-7}</p>	$\text{cons?}[\text{Byte}](bl')$
<p>{1}</p>	$\begin{aligned} & \forall (s: [\text{Memory_Address_4G}, \text{list}[\text{Byte}]]): \\ & \quad \text{type_of}(s'1) = \text{type_of}(a') \wedge \\ & \quad s'1 + \text{length}(s'2) \leq a' + \text{length}(bl') \wedge a' \leq s'1 \wedge \text{cons?}(s'2) \\ & \quad \supset \text{Mem?}(\text{min_linear}'\text{type_of}) \end{aligned}$
<p>{2}</p>	$\begin{aligned} & \text{every}(\lambda (e: [\text{Memory_Address_4G}, \text{list}[\text{Byte}]]): \\ & \quad \text{unchanged_memory_invariant?}(\text{pm_phy}'\text{mem}, \text{pm}'\text{states}, \\ & \quad \quad \text{singleton}(\text{expr_2_super}(\text{linear_read_side_effect_in_page} \\ & \quad \quad \quad (e))), \\ & \quad \quad (\text{pm_phy}'\text{ro_addr} \cup \text{pm_phy}'\text{rw_addr}))) \\ & \quad (\text{split}(\text{min_page}, a', bl')) \end{aligned}$

Expanding the definition of min_linear,

Expanding the definition of Mem,

which is trivially true.

This completes the proof of split_linear_read_side_effects_unchanged_memory.1.1.1.1.1.2.

split_linear_read_side_effects_unchanged_memory.1.1.1.1.2:

```

{-1} every(λ (t_1: [Address, list[Byte]]):
  cons?(t_1'2) ∧
  a' ≤ t_1'1 ∧
  t_1'1 + length(t_1'2) ≤ a' + length(bl') ∧
  type_of(t_1'1) = type_of(a') ∧
  reg_base(min_linear)(type_of(a')) ≤ t_1'1 ∧
  t_1'1 < reg_size(max_linear)(type_of(a')) ∧
  rem(expt(2, min_page))(offset(t_1'1)) + length(t_1'2) ≤
  expt(2, min_page))
  (split(min_page, a', bl'))
⊃
every(λ (s: [Memory_Address_4G, list[Byte]]):
  cons?(s'2) ∧
  a' ≤ s'1 ∧
  s'1 + length(s'2) ≤ a' + length(bl') ∧
  type_of(s'1) = type_of(a') ∧
  reg_base(min_linear)(type_of(a')) ≤ s'1 ∧
  s'1 < reg_size(max_linear)(type_of(a')) ∧
  rem(expt(2, min_page))(offset(s'1)) + length(s'2) ≤
  expt(2, min_page))
  (split(min_page, a', bl'))
{-2} every(λ (t_1: [Address, list[Byte]]):
  cons?(t_1'2) ∧
  a' ≤ t_1'1 ∧
  t_1'1 + length(t_1'2) ≤ a' + length(bl') ∧
  type_of(t_1'1) = type_of(a') ∧
  reg_base(min_linear)(type_of(a')) ≤ t_1'1 ∧
  t_1'1 < reg_size(max_linear)(type_of(a')))
  (split(min_page, a', bl'))
∧
every(λ (a_2: Address, bl2: list[Byte]):
  rem(expt(2, min_page))(offset(a_2)) + length(bl2) ≤
  expt(2, min_page))
  (split(min_page, a', bl'))
⊃
every(λ (t: [Address, list[Byte]]):
  (cons?(t'2) ∧
  a' ≤ t'1 ∧
  t'1 + length(t'2) ≤ a' + length(bl') ∧
  type_of(t'1) = type_of(a') ∧
  reg_base(min_linear)(type_of(a')) ≤ t'1 ∧
  t'1 < reg_size(max_linear)(type_of(a')))
  ∧
  rem(expt(2, min_page))(offset(t'1)) + length(t'2) ≤
  expt(2, min_page))
  (split(min_page, a', bl'))
{-3} every(λ (t: [Address, list[Byte]]):
  cons?(t'2) ∧
  a' ≤ t'1 ∧ t'1 + length(t'2) ≤ a' + length(bl')
  (split(min_page, a', bl'))
∧
every(λ (t: [Address, list[Byte]]):
  type_of(t'1) = type_of(a') ∧
  reg_base(min_linear)(type_of(a')) ≤ t'1 ∧
  t'1 < reg_size(max_linear)(type_of(a')))
  (split(min_page, a', bl'))
⊃
every(λ (t_1: [Address, list[Byte]]):
  (cons?(t_1'2) ∧
  a' ≤ t_1'1 ∧ t_1'1 + length(t_1'2) ≤ a' + length(bl'))
  ∧
  type_of(t_1'1) = type_of(a') ∧
  reg_base(min_linear)(type_of(a')) ≤ t_1'1 ∧

```

Expanding the definition of `max_linear`,

Expanding the definition of `Mem`,

which is trivially true.

This completes the proof of `split_linear_read_side_effects_unchanged_memory.1.1.1.1.2`.

split_linear_read_side_effects_unchanged_memory.1.1.1.1.3:

```

{-1} every(λ (t_1: [Address, list[Byte]]):
  cons?(t_1'2) ∧
  a' ≤ t_1'1 ∧
  t_1'1 + length(t_1'2) ≤ a' + length(bl') ∧
  type_of(t_1'1) = type_of(a') ∧
  reg_base(min_linear)(type_of(a')) ≤ t_1'1 ∧
  t_1'1 < reg_size(max_linear)(type_of(a')) ∧
  rem(expt(2, min_page))(offset(t_1'1)) + length(t_1'2) ≤
  expt(2, min_page))
  (split(min_page, a', bl'))
⊃
every(λ (s: [Memory_Address_4G, list[Byte]]):
  cons?(s'2) ∧
  a' ≤ s'1 ∧
  s'1 + length(s'2) ≤ a' + length(bl') ∧
  type_of(s'1) = type_of(a') ∧
  reg_base(min_linear)(type_of(a')) ≤ s'1 ∧
  s'1 < reg_size(max_linear)(type_of(a')) ∧
  rem(expt(2, min_page))(offset(s'1)) + length(s'2) ≤
  expt(2, min_page))
  (split(min_page, a', bl'))
{-2} every(λ (t_1: [Address, list[Byte]]):
  cons?(t_1'2) ∧
  a' ≤ t_1'1 ∧
  t_1'1 + length(t_1'2) ≤ a' + length(bl') ∧
  type_of(t_1'1) = type_of(a') ∧
  reg_base(min_linear)(type_of(a')) ≤ t_1'1 ∧
  t_1'1 < reg_size(max_linear)(type_of(a')))
  (split(min_page, a', bl'))
∧
every(λ (a_2: Address, bl2: list[Byte]):
  rem(expt(2, min_page))(offset(a_2)) + length(bl2) ≤
  expt(2, min_page))
  (split(min_page, a', bl'))
⊃
every(λ (t: [Address, list[Byte]]):
  (cons?(t'2) ∧
  a' ≤ t'1 ∧
  t'1 + length(t'2) ≤ a' + length(bl') ∧
  type_of(t'1) = type_of(a') ∧
  reg_base(min_linear)(type_of(a')) ≤ t'1 ∧
  t'1 < reg_size(max_linear)(type_of(a')))
  ∧
  rem(expt(2, min_page))(offset(t'1)) + length(t'2) ≤
  expt(2, min_page))
  (split(min_page, a', bl'))
{-3} every(λ (t: [Address, list[Byte]]):
  cons?(t'2) ∧
  a' ≤ t'1 ∧ t'1 + length(t'2) ≤ a' + length(bl')
  (split(min_page, a', bl'))
∧
every(λ (t: [Address, list[Byte]]):
  type_of(t'1) = type_of(a') ∧
  reg_base(min_linear)(type_of(a')) ≤ t'1 ∧
  t'1 < reg_size(max_linear)(type_of(a')))
  (split(min_page, a', bl'))
⊃
every(λ (t_1: [Address, list[Byte]]):
  (cons?(t_1'2) ∧
  a' ≤ t_1'1 ∧ t_1'1 + length(t_1'2) ≤ a' + length(bl'))
  ∧
  type_of(t_1'1) = type_of(a') ∧
  reg_base(min_linear)(type_of(a')) ≤ t_1'1 ∧

```

Expanding the definition of `min_linear`,

Expanding the definition of `Mem`,

which is trivially true.

This completes the proof of `split_linear_read_side_effects_unchanged_memory.1.1.1.1.3`.

split_linear_read_side_effects_unchanged_memory.1.1.1.2:

```

{-1} every(λ (t_1: [Address, list[Byte]]):
      cons?(t_1'2) ∧
      a' ≤ t_1'1 ∧
      t_1'1 + length(t_1'2) ≤ a' + length(bl') ∧
      type_of(t_1'1) = type_of(a') ∧
      reg_base(min_linear)(type_of(a')) ≤ t_1'1 ∧
      t_1'1 < reg_size(max_linear)(type_of(a')))
      (split(min_page, a', bl'))
  ∧
  every(λ (a_2: Address, bl2: list[Byte]):
      rem(expt(2, min_page))(offset(a_2)) + length(bl2) ≤
      expt(2, min_page))
      (split(min_page, a', bl'))
  ⊃
  every(λ (t: [Address, list[Byte]]):
      (cons?(t'2) ∧
      a' ≤ t'1 ∧
      t'1 + length(t'2) ≤ a' + length(bl') ∧
      type_of(t'1) = type_of(a') ∧
      reg_base(min_linear)(type_of(a')) ≤ t'1 ∧
      t'1 < reg_size(max_linear)(type_of(a')))
      ∧
      rem(expt(2, min_page))(offset(t'1)) + length(t'2) ≤
      expt(2, min_page))
      (split(min_page, a', bl'))
{-2} every(λ (t: [Address, list[Byte]]):
      cons?(t'2) ∧
      a' ≤ t'1 ∧ t'1 + length(t'2) ≤ a' + length(bl'))
      (split(min_page, a', bl'))
  ∧
  every(λ (t: [Address, list[Byte]]):
      type_of(t'1) = type_of(a') ∧
      reg_base(min_linear)(type_of(a')) ≤ t'1 ∧
      t'1 < reg_size(max_linear)(type_of(a')))
      (split(min_page, a', bl'))
  ⊃
  every(λ (t_1: [Address, list[Byte]]):
      (cons?(t_1'2) ∧
      a' ≤ t_1'1 ∧ t_1'1 + length(t_1'2) ≤ a' + length(bl'))
      ∧
      type_of(t_1'1) = type_of(a') ∧
      reg_base(min_linear)(type_of(a')) ≤ t_1'1 ∧
      t_1'1 < reg_size(max_linear)(type_of(a')))
      (split(min_page, a', bl'))
{-3} every(λ (a_2: Address, bl2: list[Byte]): cons?(bl2))(split(min_page, a', bl')) ∧
  every(λ (a_2: Address, bl2: list[Byte]):
      a' ≤ a_2 ∧ a_2 + length(bl2) ≤ a' + length(bl'))
      (split(min_page, a', bl'))
  ⊃
  every(λ (t: [Address, list[Byte]]):
      cons?(t'2) ∧
      a' ≤ t'1 ∧ t'1 + length(t'2) ≤ a' + length(bl'))
      (split(min_page, a', bl'))
{-4} is_linear_plain_memory?(pm')
{-5} (address_block(a', length(bl')) ⊆ (pm'ro_addr ∪ pm'rw_addr))
2284 {-6} every(λ (e: [Memory_Address_4G, list[Byte]]):
      transformer_invariant?(pm' states,
      singleton(expr_2_super(linear_read_side_effect_in_page(e)))
      (split(min_page, a', bl'))
{-7} every(λ (a_2: Address, bl2: list[Byte]):
      rem(expt(2, min_page))(offset(a_2)) + length(bl2) ≤ expt(2, min_page))
      (split(min_page, a', bl'))
{-8} every(λ (t: [Address, list[Byte]]):

```

Expanding the definition of `max_linear`,

Expanding the definition of `Mem`,

which is trivially true.

This completes the proof of `split_linear_read_side_effects_unchanged_memory.1.1.1.2`.

split_linear_read_side_effects_unchanged_memory.1.1.1.3:

```

{-1} every(λ (t_1: [Address, list[Byte]]):
      cons?(t_1'2) ∧
      a' ≤ t_1'1 ∧
      t_1'1 + length(t_1'2) ≤ a' + length(bl') ∧
      type_of(t_1'1) = type_of(a') ∧
      reg_base(min_linear)(type_of(a')) ≤ t_1'1 ∧
      t_1'1 < reg_size(max_linear)(type_of(a')))
      (split(min_page, a', bl'))
  ∧
  every(λ (a_2: Address, bl2: list[Byte]):
      rem(expt(2, min_page))(offset(a_2)) + length(bl2) ≤
      expt(2, min_page))
      (split(min_page, a', bl'))
  ⊃
  every(λ (t: [Address, list[Byte]]):
      (cons?(t'2) ∧
      a' ≤ t'1 ∧
      t'1 + length(t'2) ≤ a' + length(bl') ∧
      type_of(t'1) = type_of(a') ∧
      reg_base(min_linear)(type_of(a')) ≤ t'1 ∧
      t'1 < reg_size(max_linear)(type_of(a')))
      ∧
      rem(expt(2, min_page))(offset(t'1)) + length(t'2) ≤
      expt(2, min_page))
      (split(min_page, a', bl'))
{-2} every(λ (t: [Address, list[Byte]]):
      cons?(t'2) ∧
      a' ≤ t'1 ∧ t'1 + length(t'2) ≤ a' + length(bl'))
      (split(min_page, a', bl'))
  ∧
  every(λ (t: [Address, list[Byte]]):
      type_of(t'1) = type_of(a') ∧
      reg_base(min_linear)(type_of(a')) ≤ t'1 ∧
      t'1 < reg_size(max_linear)(type_of(a')))
      (split(min_page, a', bl'))
  ⊃
  every(λ (t_1: [Address, list[Byte]]):
      (cons?(t_1'2) ∧
      a' ≤ t_1'1 ∧ t_1'1 + length(t_1'2) ≤ a' + length(bl'))
      ∧
      type_of(t_1'1) = type_of(a') ∧
      reg_base(min_linear)(type_of(a')) ≤ t_1'1 ∧
      t_1'1 < reg_size(max_linear)(type_of(a')))
      (split(min_page, a', bl'))
{-3} every(λ (a_2: Address, bl2: list[Byte]): cons?(bl2))(split(min_page, a', bl')) ∧
  every(λ (a_2: Address, bl2: list[Byte]):
      a' ≤ a_2 ∧ a_2 + length(bl2) ≤ a' + length(bl'))
      (split(min_page, a', bl'))
  ⊃
  every(λ (t: [Address, list[Byte]]):
      cons?(t'2) ∧
      a' ≤ t'1 ∧ t'1 + length(t'2) ≤ a' + length(bl'))
      (split(min_page, a', bl'))
{-4} is_linear_plain_memory?(pm')
{-5} (address_block(a', length(bl')) ⊆ (pm'ro_addr ∪ pm'rw_addr))
2286 {-6} every(λ (e: [Memory_Address_4G, list[Byte]]):
      transformer_invariant?(pm' states,
      singleton(expr_2_super(linear_read_side_effect_in_page(e)))
      (split(min_page, a', bl'))
{-7} every(λ (a_2: Address, bl2: list[Byte]):
      rem(expt(2, min_page))(offset(a_2)) + length(bl2) ≤ expt(2, min_page))
      (split(min_page, a', bl'))
{-8} every(λ (t: [Address, list[Byte]]):

```

Expanding the definition of `min_linear`,

Expanding the definition of `Mem`,

which is trivially true.

This completes the proof of `split_linear_read_side_effects_unchanged_memory.1.1.1.3`.

split_linear_read_side_effects_unchanged_memory.1.1.2:

{-1}	$\text{every}(\lambda (t: [\text{Address}, \text{list}[\text{Byte}]]):$ $\text{cons?}(t'2) \wedge$ $a' \leq t'1 \wedge t'1 + \text{length}(t'2) \leq a' + \text{length}(bl')$ $(\text{split}(\text{min_page}, a', bl'))$ \wedge $\text{every}(\lambda (t: [\text{Address}, \text{list}[\text{Byte}]]):$ $\text{type_of}(t'1) = \text{type_of}(a') \wedge$ $\text{reg_base}(\text{min_linear})(\text{type_of}(a')) \leq t'1 \wedge$ $t'1 < \text{reg_size}(\text{max_linear})(\text{type_of}(a'))$ $(\text{split}(\text{min_page}, a', bl'))$ \supset $\text{every}(\lambda (t_1: [\text{Address}, \text{list}[\text{Byte}]]):$ $\text{cons?}(t_1'2) \wedge$ $a' \leq t_1'1 \wedge t_1'1 + \text{length}(t_1'2) \leq a' + \text{length}(bl')$ \wedge $\text{type_of}(t_1'1) = \text{type_of}(a') \wedge$ $\text{reg_base}(\text{min_linear})(\text{type_of}(a')) \leq t_1'1 \wedge$ $t_1'1 < \text{reg_size}(\text{max_linear})(\text{type_of}(a'))$ $(\text{split}(\text{min_page}, a', bl'))$
{-2}	$\text{every}(\lambda (a_2: \text{Address}, bl_2: \text{list}[\text{Byte}]): \text{cons?}(bl_2)(\text{split}(\text{min_page}, a', bl')) \wedge$ $\text{every}(\lambda (a_2: \text{Address}, bl_2: \text{list}[\text{Byte}]):$ $a' \leq a_2 \wedge a_2 + \text{length}(bl_2) \leq a' + \text{length}(bl')$ $(\text{split}(\text{min_page}, a', bl'))$ \supset $\text{every}(\lambda (t: [\text{Address}, \text{list}[\text{Byte}]]):$ $\text{cons?}(t'2) \wedge$ $a' \leq t'1 \wedge t'1 + \text{length}(t'2) \leq a' + \text{length}(bl')$ $(\text{split}(\text{min_page}, a', bl'))$
{-3}	$\text{is_linear_plain_memory?}(pm')$
{-4}	$(\text{address_block}(a', \text{length}(bl')) \subseteq (pm' \text{ro_addr} \cup pm' \text{rw_addr}))$
{-5}	$\text{every}(\lambda (e: [\text{Memory_Address_4G}, \text{list}[\text{Byte}]]):$ $\text{transformer_invariant?}(pm' \text{states},$ $\text{singleton}(\text{expr_2_super}(\text{linear_read_side_effect_in_page}(e)))$
{-6}	$\text{every}(\lambda (a_2: \text{Address}, bl_2: \text{list}[\text{Byte}]):$ $\text{rem}(\text{expt}(2, \text{min_page}))(\text{offset}(a_2)) + \text{length}(bl_2) \leq \text{expt}(2, \text{min_page}))$ $(\text{split}(\text{min_page}, a', bl'))$
{-7}	$\text{every}(\lambda (t: [\text{Address}, \text{list}[\text{Byte}]]):$ $\text{type_of}(t'1) = \text{type_of}(a') \wedge$ $\text{reg_base}(\text{min_linear})(\text{type_of}(a')) \leq t'1 \wedge$ $t'1 < \text{reg_size}(\text{max_linear})(\text{type_of}(a'))$ $(\text{split}(\text{min_page}, a', bl'))$
{-8}	$\text{every}(\lambda (a_2: \text{Address}, bl_2: \text{list}[\text{Byte}]): \text{cons?}(bl_2)(\text{split}(\text{min_page}, a', bl'))$
{-9}	$\text{every}(\lambda (a_2: \text{Address}, bl_2: \text{list}[\text{Byte}]):$ $a' \leq a_2 \wedge a_2 + \text{length}(bl_2) \leq a' + \text{length}(bl')$ $(\text{split}(\text{min_page}, a', bl'))$
{-10}	$\text{Mem?}(a' \text{type_of})$
{-11}	$0 \leq a' \text{offset}$
{-12}	$a' \text{offset} < \text{max_linear_offset}$
{-13}	$\text{every}(\lambda (x: \text{number}):$ $\text{number_field_pred}(x) \wedge$ $\text{real_pred}(x) \wedge$ $\text{rational_pred}(x) \wedge \text{integer_pred}(x) \wedge x \geq 0 \wedge x < \text{max_byte})$ (bl')
{-14}	$\text{cons?}[\text{Byte}](bl')$
2288	$\{1\} \quad \forall (t_1: [\text{Address}, \text{list}[\text{Byte}]]):$ $\text{reg_base}(\text{min_linear})(\text{type_of}(a')) \leq t_1'1 \wedge$ $\text{type_of}(t_1'1) = \text{type_of}(a') \wedge$ $t_1'1 + \text{length}(t_1'2) \leq a' + \text{length}(bl') \wedge$ $a' \leq t_1'1 \wedge \text{cons?}(t_1'2)$ $\supset \text{Mem?}(\text{max_linear} \text{type_of})$
{2}	$\text{every}(\lambda (e: [\text{Memory_Address_4G}, \text{list}[\text{Byte}]]):$ $\text{unchanged_memory_invariant?}(pm_phy \text{mem}, pm' \text{states},$

Expanding the definition of `max_linear`,

Expanding the definition of `Mem`,

which is trivially true.

This completes the proof of `split_linear_read_side_effects_unchanged_memory.1.1.2`.

split_linear_read_side_effects_unchanged_memory.1.1.3:

```

{-1} every(λ (t: [Address, list[Byte]]):
      cons?(t'2) ∧
      a' ≤ t'1 ∧ t'1 + length(t'2) ≤ a' + length(bl')
      (split(min_page, a', bl'))
    ∧
    every(λ (t: [Address, list[Byte]]):
      type_of(t'1) = type_of(a') ∧
      reg_base(min_linear)(type_of(a')) ≤ t'1 ∧
      t'1 < reg_size(max_linear)(type_of(a'))
      (split(min_page, a', bl'))
    ⊃
    every(λ (t_1: [Address, list[Byte]]):
      cons?(t_1'2) ∧
      a' ≤ t_1'1 ∧ t_1'1 + length(t_1'2) ≤ a' + length(bl')
      ∧
      type_of(t_1'1) = type_of(a') ∧
      reg_base(min_linear)(type_of(a')) ≤ t_1'1 ∧
      t_1'1 < reg_size(max_linear)(type_of(a'))
      (split(min_page, a', bl'))
  {-2} every(λ (a_2: Address, bl_2: list[Byte]): cons?(bl_2))(split(min_page, a', bl')) ∧
    every(λ (a_2: Address, bl_2: list[Byte]):
      a' ≤ a_2 ∧ a_2 + length(bl_2) ≤ a' + length(bl')
      (split(min_page, a', bl'))
    ⊃
    every(λ (t: [Address, list[Byte]]):
      cons?(t'2) ∧
      a' ≤ t'1 ∧ t'1 + length(t'2) ≤ a' + length(bl')
      (split(min_page, a', bl'))
  {-3} is_linear_plain_memory?(pm')
  {-4} (address_block(a', length(bl')) ⊆ (pm'ro_addr ∪ pm'rw_addr))
  {-5} every(λ (e: [Memory_Address_4G, list[Byte]]):
      transformer_invariant?(pm' states,
      singleton(expr_2_super(linear_read_side_effect_in_page(e)))
      (split(min_page, a', bl'))
  {-6} every(λ (a_2: Address, bl_2: list[Byte]):
      rem(expt(2, min_page))(offset(a_2)) + length(bl_2) ≤ expt(2, min_page)
      (split(min_page, a', bl'))
  {-7} every(λ (t: [Address, list[Byte]]):
      type_of(t'1) = type_of(a') ∧
      reg_base(min_linear)(type_of(a')) ≤ t'1 ∧
      t'1 < reg_size(max_linear)(type_of(a'))
      (split(min_page, a', bl'))
  {-8} every(λ (a_2: Address, bl_2: list[Byte]): cons?(bl_2))(split(min_page, a', bl'))
  {-9} every(λ (a_2: Address, bl_2: list[Byte]):
      a' ≤ a_2 ∧ a_2 + length(bl_2) ≤ a' + length(bl')
      (split(min_page, a', bl'))
  {-10} Mem?(a' type_of)
  {-11} 0 ≤ a' offset
  {-12} a' offset < max_linear_offset
  {-13} every(λ (x: number):
      number_field_pred(x) ∧
      real_pred(x) ∧
      rational_pred(x) ∧ integer_pred(x) ∧ x ≥ 0 ∧ x < max_byte)
      (bl')
  {-14} cons?[Byte](bl')
2290 {1} ∀ (t_1: [Address, list[Byte]]):
      type_of(t_1'1) = type_of(a') ∧
      t_1'1 + length(t_1'2) ≤ a' + length(bl') ∧
      a' ≤ t_1'1 ∧ cons?(t_1'2)
      ⊃ Mem?(min_linear type_of)
  {2} every(λ (e: [Memory_Address_4G, list[Byte]]):
      unchanged_memory_invariant?(pm_phy mem, pm' states,
      singleton(expr_2_super(linear_read_side_effect_in_page

```

Expanding the definition of min_linear,
 Expanding the definition of Mem,
 which is trivially true.

This completes the proof of split_linear_read_side_effects_unchanged_memory.1.1.3.

split_linear_read_side_effects_unchanged_memory.1.2:

{-1}	every(λ (a_2 : Address, bl2: list[Byte]): cons?(bl2))(split(min_page, a' , bl')) \wedge every(λ (a_2 : Address, bl2: list[Byte]): $a' \leq a_2 \wedge a_2 + \text{length}(bl2) \leq a' + \text{length}(bl')$) (split(min_page, a' , bl')) \supset every(λ (t : [Address, list[Byte]]): cons?($t'2$) \wedge $a' \leq t'1 \wedge t'1 + \text{length}(t'2) \leq a' + \text{length}(bl')$) (split(min_page, a' , bl'))
{-2}	is_linear_plain_memory?(pm')
{-3}	(address_block(a' , length(bl')) \subseteq (pm'ro_addr \cup pm'rw_addr))
{-4}	every(λ (e : [Memory_Address_4G, list[Byte]]): transformer_invariant?(pm'states, singleton(expr_2_super(linear_read_side_effect_in_page(e)))) (split(min_page, a' , bl'))
{-5}	every(λ (a_2 : Address, bl2: list[Byte]): rem(expt(2, min_page))(offset(a_2)) + length(bl2) \leq expt(2, min_page)) (split(min_page, a' , bl'))
{-6}	every(λ (t : [Address, list[Byte]]): type_of($t'1$) = type_of(a') \wedge reg_base(min_linear)(type_of(a')) \leq $t'1 \wedge$ $t'1 <$ reg_size(max_linear)(type_of(a')) (split(min_page, a' , bl'))
{-7}	every(λ (a_2 : Address, bl2: list[Byte]): cons?(bl2))(split(min_page, a' , bl'))
{-8}	every(λ (a_2 : Address, bl2: list[Byte]): $a' \leq a_2 \wedge a_2 + \text{length}(bl2) \leq a' + \text{length}(bl')$) (split(min_page, a' , bl'))
{-9}	Mem?(a' 'type_of)
{-10}	$0 \leq a'$ 'offset
{-11}	a' 'offset $<$ max_linear_offset
{-12}	every(λ (x : number): number_field_pred(x) \wedge real_pred(x) \wedge rational_pred(x) \wedge integer_pred(x) \wedge $x \geq 0 \wedge x <$ max_byte) (bl')
{-13}	cons?[Byte](bl')
{1}	\forall (t : [Address, list[Byte]]): reg_base(min_linear)(type_of(a')) \leq $t'1 \wedge$ type_of($t'1$) = type_of(a') \supset Mem?(max_linear'type_of)
{2}	every(λ (e : [Memory_Address_4G, list[Byte]]): unchanged_memory_invariant?(pm_phy'mem, pm'states, singleton(expr_2_super(linear_read_side_effect_in_page (e))), (pm_phy'ro_addr \cup pm_phy'rw_addr)) (split(min_page, a' , bl'))

Expanding the definition of max_linear,
 Expanding the definition of Mem,

which is trivially true.

This completes the proof of `split_linear_read_side_effects_unchanged_memory.1.2`.

`split_linear_read_side_effects_unchanged_memory.1.3`:

{-1}	$\text{every}(\lambda (a_2: \text{Address}, \text{bl2}: \text{list}[\text{Byte}]): \text{cons?}(\text{bl2}))(\text{split}(\text{min_page}, a', \text{bl}')) \wedge$ $\text{every}(\lambda (a_2: \text{Address}, \text{bl2}: \text{list}[\text{Byte}]):$ $a' \leq a_2 \wedge a_2 + \text{length}(\text{bl2}) \leq a' + \text{length}(\text{bl}'))$ $(\text{split}(\text{min_page}, a', \text{bl}'))$ \supset $\text{every}(\lambda (t: [\text{Address}, \text{list}[\text{Byte}]]):$ $\text{cons?}(t'2) \wedge$ $a' \leq t'1 \wedge t'1 + \text{length}(t'2) \leq a' + \text{length}(\text{bl}'))$ $(\text{split}(\text{min_page}, a', \text{bl}'))$
{-2}	<code>is_linear_plain_memory?(pm')</code>
{-3}	<code>(address_block(a', length(bl')) \subseteq (pm'ro_addr \cup pm'rw_addr))</code>
{-4}	$\text{every}(\lambda (e: [\text{Memory_Address_4G}, \text{list}[\text{Byte}]]):$ $\text{transformer_invariant?}(\text{pm}'\text{states},$ $\text{singleton}(\text{expr_2_super}(\text{linear_read_side_effect_in_page}(e)))$ $(\text{split}(\text{min_page}, a', \text{bl}'))$
{-5}	$\text{every}(\lambda (a_2: \text{Address}, \text{bl2}: \text{list}[\text{Byte}]):$ $\text{rem}(\text{expt}(2, \text{min_page}))(\text{offset}(a_2)) + \text{length}(\text{bl2}) \leq \text{expt}(2, \text{min_page}))$ $(\text{split}(\text{min_page}, a', \text{bl}'))$
{-6}	$\text{every}(\lambda (t: [\text{Address}, \text{list}[\text{Byte}]]):$ $\text{type_of}(t'1) = \text{type_of}(a') \wedge$ $\text{reg_base}(\text{min_linear})(\text{type_of}(a')) \leq t'1 \wedge$ $t'1 < \text{reg_size}(\text{max_linear})(\text{type_of}(a')))$ $(\text{split}(\text{min_page}, a', \text{bl}'))$
{-7}	$\text{every}(\lambda (a_2: \text{Address}, \text{bl2}: \text{list}[\text{Byte}]): \text{cons?}(\text{bl2}))(\text{split}(\text{min_page}, a', \text{bl}'))$
{-8}	$\text{every}(\lambda (a_2: \text{Address}, \text{bl2}: \text{list}[\text{Byte}]):$ $a' \leq a_2 \wedge a_2 + \text{length}(\text{bl2}) \leq a' + \text{length}(\text{bl}'))$ $(\text{split}(\text{min_page}, a', \text{bl}'))$
{-9}	<code>Mem?(a'type_of)</code>
{-10}	<code>0 \leq a'offset</code>
{-11}	<code>a'offset < max_linear_offset</code>
{-12}	$\text{every}(\lambda (x: \text{number}):$ $\text{number_field_pred}(x) \wedge$ $\text{real_pred}(x) \wedge$ $\text{rational_pred}(x) \wedge \text{integer_pred}(x) \wedge x \geq 0 \wedge x < \text{max_byte})$ (bl')
{-13}	<code>cons?[Byte](bl')</code>
{1}	$\forall (t: [\text{Address}, \text{list}[\text{Byte}]]):$ $\text{type_of}(t'1) = \text{type_of}(a') \supset \text{Mem?}(\text{min_linear}'\text{type_of})$
{2}	$\text{every}(\lambda (e: [\text{Memory_Address_4G}, \text{list}[\text{Byte}]]):$ $\text{unchanged_memory_invariant?}(\text{pm_phy}'\text{mem}, \text{pm}'\text{states},$ $\text{singleton}(\text{expr_2_super}(\text{linear_read_side_effect_in_page}$ $(e))),$ $(\text{pm_phy}'\text{ro_addr} \cup \text{pm_phy}'\text{rw_addr})))$ $(\text{split}(\text{min_page}, a', \text{bl}'))$

Expanding the definition of `min_linear`,

Expanding the definition of `Mem`,

which is trivially true.

This completes the proof of `split_linear_read_side_effects_unchanged_memory.1.3`.

split_linear_read_side_effects_unchanged_memory.2:

{-1}	is_linear_plain_memory?(pm')
{-2}	(address_block(a', length(bl')) \subseteq (pm'ro_addr \cup pm'rw_addr))
{-3}	every(λ (e: [Memory_Address_4G, list[Byte]]): transformer_invariant?(pm' states, singleton(expr_2_super(linear_read_side_effect_in_page(e)))) (split(min_page, a', bl'))
{-4}	every(λ (a ₂ : Address, bl2: list[Byte]): rem(expt(2, min_page))(offset(a ₂)) + length(bl2) \leq expt(2, min_page)) (split(min_page, a', bl'))
{-5}	every(λ (a ₂ : Address, bl2: list[Byte]): cons?(bl2))(split(min_page, a', bl'))
{-6}	every(λ (a ₂ : Address, bl2: list[Byte]): a' \leq a ₂ \wedge a ₂ + length(bl2) \leq a' + length(bl')) (split(min_page, a', bl'))
{-7}	Mem?(a' type_of)
{-8}	0 \leq a' offset
{-9}	a' offset < max_linear_offset
{-10}	every(λ (x: number): number_field_pred(x) \wedge real_pred(x) \wedge rational_pred(x) \wedge integer_pred(x) \wedge x \geq 0 \wedge x < max_byte) (bl')
{-11}	cons?[Byte](bl')
{1}	a' + length(bl') \leq reg_size(max_linear)(type_of(a'))
{2}	every(λ (e: [Memory_Address_4G, list[Byte]]): unchanged_memory_invariant?(pm_phy mem, pm' states, singleton(expr_2_super(linear_read_side_effect_in_page (e))), (pm_phy ro_addr \cup pm_phy rw_addr))) (split(min_page, a', bl'))

Hiding formulas: (-3 -4 -5 -6 -10 2),

Expanding the definition of length,

Using lemma pm_memory_addr,

Simplifying, rewriting, and recording with decision procedures,

Hiding formulas: -2,

Expanding the definition of subset?,

Installing automatic rewrites from: (Mem! max_linear! bus_width!)

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of split_linear_read_side_effects_unchanged_memory.2.

split_linear_read_side_effects_unchanged_memory.3:

<pre> {-1} is_linear_plain_memory?(pm') {-2} (address_block(a', length(bl')) ⊆ (pm'ro_addr ∪ pm'rw_addr)) {-3} every(λ (e: [Memory_Address_4G, list[Byte]]): transformer_invariant?(pm' states, singleton(expr_2_super(linear_read_side_effect_in_page(e))) (split(min_page, a', bl'))) {-4} every(λ (a₂: Address, bl₂: list[Byte]): rem(expt(2, min_page))(offset(a₂)) + length(bl₂) ≤ expt(2, min_page)) (split(min_page, a', bl')) {-5} every(λ (a₂: Address, bl₂: list[Byte]): cons?(bl₂))(split(min_page, a', bl')) {-6} every(λ (a₂: Address, bl₂: list[Byte]): a' ≤ a₂ ∧ a₂ + length(bl₂) ≤ a' + length(bl')) (split(min_page, a', bl')) {-7} Mem?(a' type_of) {-8} 0 ≤ a' offset {-9} a' offset < max_linear_offset {-10} every(λ (x: number): number_field_pred(x) ∧ real_pred(x) ∧ rational_pred(x) ∧ integer_pred(x) ∧ x ≥ 0 ∧ x < max_byte) (bl') {-11} cons?[Byte](bl') </pre>	<pre> {1} reg_base(min_linear)(type_of(a')) ≤ a' {2} every(λ (e: [Memory_Address_4G, list[Byte]]): unchanged_memory_invariant?(pm_phy mem, pm' states, singleton(expr_2_super(linear_read_side_effect_in_page (e))), (pm_phy ro_addr ∪ pm_phy rw_addr))) (split(min_page, a', bl')) </pre>
--	---

Expanding the definition of reg_base,

Expanding the definition of min_linear,

Expanding the definition of Mem,

Expanding the definition of <=,

which is trivially true.

This completes the proof of split_linear_read_side_effects_unchanged_memory.3.

Q.E.D.

C.117.83 Linear_Memory_Properties.split_linear_write_side_effects_unchanged_memory

Terse proof for split_linear_write_side_effects_unchanged_memory.

split_linear_write_side_effects_unchanged_memory:

$\{1\} \quad \forall (pm: \text{Plain_Memory}[\text{Linear_memory}], a: \text{Memory_Address_4G}, bl: (\text{cons?}[\text{Byte}])): \\ \text{is_linear_plain_memory?}(pm) \wedge (\text{address_block}(a, \text{length}(bl)) \subseteq pm\text{'rw_addr}) \supset \\ \text{every}(\lambda (e: [\text{Memory_Address_4G}, \text{list}[\text{Byte}]]): \\ \text{unchanged_memory_invariant?}(pm_phy\text{'mem}, pm\text{'states}, \\ \text{singleton}(\text{expr_2_super}(\text{linear_write_side_effect_in_page} \\ (e))), \\ (pm_phy\text{'ro_addr} \cup pm_phy\text{'rw_addr}))) \\ (\text{split}(\text{min_page}, a, bl))$
--

Repeatedly Skolemizing and flattening,

Installing automatic rewrites from: (Mem! max_linear! min_linear!)

Using lemma split_range,

Using lemma split_no_null,

Using lemma split_type,

Using lemma split_pair_cross_size,

Using lemma split_linear_write_side_effects_states,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 3 subgoals:


```
split_linear_write_side_effects_unchanged_memory.1:
```

{-1}	is_linear_plain_memory?(pm')
{-2}	(address_block(a', length(bl')) \subseteq pm'rw_addr)
{-3}	every(λ (e: [Memory_Address_4G, list[Byte]]): transformer_invariant?(pm' states, singleton(expr_2_super(linear_write_side_effect_in_page(e)) (split(min_page, a', bl'))
{-4}	every(λ (a ₂ : Address, bl2: list[Byte]): rem(expt(2, min_page))(offset(a ₂)) + length(bl2) \leq expt(2, min_page)) (split(min_page, a', bl'))
{-5}	every(λ (t: [Address, list[Byte]]): type_of(t'1) = type_of(a') \wedge reg_base((#type_of := Mem_, offset := 0#))(type_of(a')) \leq t'1 \wedge t'1 < reg_size((#type_of := Mem_, offset := max_linear_offset#))(type_of(a')) (split(min_page, a', bl'))
{-6}	every(λ (a ₂ : Address, bl2: list[Byte]): cons?(bl2))(split(min_page, a', bl'))
{-7}	every(λ (a ₂ : Address, bl2: list[Byte]): a' \leq a ₂ \wedge a ₂ + length(bl2) \leq a' + length(bl')) (split(min_page, a', bl'))
{-8}	Mem?(a' type_of)
{-9}	0 \leq a' offset
{-10}	a' offset < max_linear_offset
{-11}	every(λ (x: number): number_field_pred(x) \wedge real_pred(x) \wedge rational_pred(x) \wedge integer_pred(x) \wedge x \geq 0 \wedge x < max_byte)
{-12}	cons?[Byte](bl')
{1}	every(λ (e: [Memory_Address_4G, list[Byte]]): unchanged_memory_invariant?(pm_phy mem, pm' states, singleton(expr_2_super(linear_write_side_effect_in_page (e))), (pm_phy ro_addr \cup pm_phy rw_addr))) (split(min_page, a', bl'))

Using lemma every_conjunct_left,

Using lemma every_conjunct_left,

Using lemma every_conjunct_left,

Using lemma every_extend[[Address, list[Byte]], [Memory_Address_4G, list[Byte]]],

Using lemma every_conjunct_left,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Hiding formulas: (-1 -2 -4 -5 -6 -7 -8 -9 -12 -16),

Installing automatic rewrites from: (##! expr_2_super! expr_2_super_res!)

Using lemma every_implied,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Hiding formulas: (-1 -2 2),

Expanding the definition of linear_write_side_effect_in_page,

Repeatedly Skolemizing and flattening,

Case splitting on pm!1'rw_addr(t!1'1),

we get 2 subgoals:

split_linear_write_side_effects_unchanged_memory.1.1:

<pre> {-1} pm'rw_addr(t'1) {-2} cons?(t'2) {-3} a' ≤ t'1 {-4} t'1 + length(t'2) ≤ a' + length(bl') {-5} type_of(t'1) = type_of(a') {-6} reg_base((#type_of := Mem_, offset := 0#))(type_of(a')) ≤ t'1 {-7} t'1 < reg_size((#type_of := Mem_, offset := max_linear_offset#))(type_of(a')) {-8} rem(expt(2, min_page))(offset(t'1)) + length(t'2) ≤ expt(2, min_page) {-9} transformer_invariant?(pm'states, singleton(expr_2_super(λ (s: ear_memory[Physical_memory, pm_phy]): ory_Address_4G) ory_write_side_effect(pm_phy'mem) {-10} (address_block(a', length(bl')) ⊆ pm'rw_addr) {-11} Mem?(a'type_of) {-12} 0 ≤ a'offset {-13} a'offset < max_linear_offset {-14} cons?[Byte](bl') </pre>	<pre> Lin- (linear_resolve(t'1, Write) ## (λ (pa: Mem- (ns: Linear_memory [Physical_memory, pm_phy]): mem- (pa, t'2, TRUE)(s))) (s))) </pre>
<pre> {1} unchanged_memory_invariant?(pm_phy'mem, pm'states, singleton(expr_2_super(λ (s: </pre>	<pre> Linear_memory [Physical_memory, pm_phy]): (linear_resolve(t'1, Write) ## (λ (pa: Mem- (ns: Linear_memory [Physical_memory, pm_phy]): mem- (pa, t'2, TRUE)(s))) (s))), (pm_phy'ro_addr ∪ pm_phy'rw_addr)) </pre>

Using lemma pm_states,
Using lemma pm_plain_phy,
Expanding the definition of unchanged_memory_invariant?,
Simplifying, rewriting, and recording with decision procedures,
Repeatedly Skolemizing and flattening,
Expanding the definition of singleton,
Replacing using formula -13,

C Proof scripts

Hiding formulas: -13,

Using lemma `pm_linear_resolve_write_ok`,

Case splitting on `subset?(address_block(data(linear_resolve(t!1'1, Write)(s1)), length(t!1'2)), pm_phy'rw_addr)`,

we get 3 subgoals:

split_linear_write_side_effects_unchanged_memory.1.1.1:

```

{-1}  (address_block(data(linear_resolve(t'1, Write)(s')), length(t'2)) ⊆ pm_phy'rw_addr)
{-2}  is_linear_plain_memory?(pm') ⊃ OK?(linear_resolve(t'1, Write)(s'))
{-3}  plain_memory?(pm_phy)
{-4}  pm' states = pm_phy' states
{-5}  pm' rw_addr(t'1)
{-6}  cons?(t'2)
{-7}  a' ≤ t'1
{-8}  t'1 + length(t'2) ≤ a' + length(bl')
{-9}  type_of(t'1) = type_of(a')
{-10} reg_base((#type_of := Mem_, offset := 0#))(type_of(a')) ≤ t'1
{-11} t'1 < reg_size((#type_of := Mem_, offset := max_linear_offset#))(type_of(a'))
{-12} rem(expt(2, min_page))(offset(t'1) + length(t'2)) ≤ expt(2, min_page)
{-13} transformer_invariant?(pm' states,
      {y |
        y =
          expr_2_super(λ (s: Linear_memory[Physical_memory, pm_phy]):
            CASES lin-
ear_resolve(t'1, Write)(s) OF
          OK(state, data):
            mem-
ory_write_side_effect(pm_phy'mem)
            (data, t'2, TRUE)(s),
          Exception(ex_type, state):
            Exception(ex_type, state),
          Fatal: Fatal,
          Hang: Hang
          ENDCASES})
{-14} pm' states(s')
{-15} union(pm_phy'ro_addr, pm_phy'rw_addr)(a'')
{-16} OK?(expr_2_super(λ (s: Linear_memory[Physical_memory, pm_phy]):
      CASES linear_resolve(t'1, Write)(s) OF
        OK(state, data):
          memory_write_side_effect(pm_phy'mem)(data, t'2, TRUE)(s),
          Exception(ex_type, state): Exception(ex_type, state),
          Fatal: Fatal,
          Hang: Hang
          ENDCASES)
      (s'))
{-17} OK?(memory_read(pm_phy'mem)(a'')(s'))
{-18} OK?(memory_read(pm_phy'mem)
      (a'')
      (state(expr_2_super(λ (s: Linear_memory[Physical_memory, pm_phy]):
        CASES linear_resolve(t'1, Write)(s) OF
          OK(state, data):
            memory_write_side_effect(pm_phy'mem)
            (data, t'2, TRUE)(s),
          Exception(ex_type, state): Excep-
tion(ex_type, state),
          Fatal: Fatal,
          Hang: Hang
          ENDCASES)
      (s'))))
{-19} (address_block(a', length(bl')) ⊆ pm'rw_addr)
{-20} Mem?(a' type_of)
{-21} 0 ≤ a' offset
{-22} a' offset < max_linear_offset
{-23} cons?[Byte](bl')
{1}  data(memory_read(pm_phy'mem)
      (a'')
      (state(expr_2_super(λ (s: Linear_memory[Physical_memory, pm_phy]):
        CASES linear_resolve(t'1, Write)(s) OF
          OK(state, data):
            memory_write_side_effect(pm_phy'mem)

```

C Proof scripts

Using lemma `linear_resolve_states`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 2 subgoals:

split_linear_write_side_effects_unchanged_memory.1.1.1.1:

<pre> {-1} pm' states(s') {-2} OK?(linear_resolve(t'1, Write)(s')) {-3} pm_phy' states(state(linear_resolve(t'1, Write)(s'))) {-4} (address_block(data(linear_resolve(t'1, Write)(s')), length(t'2)) ⊆ pm_phy'rw_addr) {-5} plain_memory?(pm_phy) {-6} pm' states = pm_phy' states {-7} pm' rw_addr(t'1) {-8} cons?(t'2) {-9} a' ≤ t'1 {-10} t'1 + length(t'2) ≤ a' + length(bl') {-11} type_of(t'1) = type_of(a') {-12} reg_base((#type_of := Mem_, offset := 0#))(type_of(a')) ≤ t'1 {-13} t'1 < reg_size((#type_of := Mem_, offset := max_linear_offset#))(type_of(a')) {-14} rem(expt(2, min_page))(offset(t'1) + length(t'2)) ≤ expt(2, min_page) {-15} transformer_invariant?(pm' states, {y y = expr_2_super(λ (s : Linear_memory [Physical_memory, pm_phy]): CASES lin- ear_resolve(t'1, Write)(s) OF OK(state, data): mem- ory_write_side_effect(pm_phy' mem) (data, t'2, TRUE)(s), Exception(ex_type, state): Exception(ex_type, state), Fatal: Fatal, Hang: Hang ENDCASES)}))) </pre>	<pre> (data, t'2, TRUE)(s), Exception(ex_type, state): Exception(ex_type, state), Fatal: Fatal, Hang: Hang ENDCASES)}) </pre>
<pre> {-16} union(pm_phy' ro_addr, pm_phy' rw_addr)(a'') {-17} OK?(memory_write_side_effect(pm_phy' mem) (data(linear_resolve(t'1, Write)(s')), t'2, TRUE)(s')) {-18} OK?(OK(state(memory_write_side_effect(pm_phy' mem) (data(linear_resolve(t'1, Write)(s')), t'2, TRUE)(s')))) {-19} OK?(memory_read(pm_phy' mem)(a'')(s')) {-20} OK?(memory_read(pm_phy' mem) (a'') (state(memory_write_side_effect(pm_phy' mem) (data(linear_resolve(t'1, Write)(s')), t'2, TRUE)(s')))) {-21} (address_block(a', length(bl')) ⊆ pm'rw_addr) {-22} Mem?(a' type_of) {-23} 0 ≤ a' offset {-24} a' offset < max_linear_offset {-25} cons?[Byte](bl') </pre>	
<pre> {1} data(memory_read(pm_phy' mem) (a'') (state(memory_write_side_effect(pm_phy' mem) (data(linear_resolve(t'1, Write)(s')), t'2, TRUE) (s')))) = data(memory_read(pm_phy' mem)(a'')(s')) </pre>	

Using lemma plain_memory_unchanged_invariant,

C Proof scripts

Using lemma `unchanged_memory_invariant_unchanged`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Case splitting on `memory_write_side_effect_super_transformers(pm_phy'mem, pm_phy'rw_addr)`
`(expr_2_super (memory_write_side_effect (pm_phy'mem) (data (linear_resolve (t!1'1, Write)(s!1)), t!1'2,`
`TRUE)))`,

we get 2 subgoals:

split_linear_write_side_effects_unchanged_memory.1.1.1.1.1:

```

{-1}  memory_write_side_effect_super_transformers(pm_phy'mem, pm_phy'rw_addr)
                                             (expr_2_super(memory_write_side_effect(pm_phy'mem)
                                                         (data
                                                           (linear_resolve
                                                            (t'1, Write)(s')),
                                                            t'2,
                                                            TRUE)))
{-2}  unchanged_memory_invariant?(pm_phy'mem, pm_phy'states,
                                   ((pm_phy'other_actions ∪ memory_read_transformers(pm_phy'mem, (pm_phy'ro_addr ∪ pm_phy'rw_addr))
                                   (pm_phy'ro_addr ∪ pm_phy'rw_addr))
{-3}  union(pm_phy'ro_addr, pm_phy'rw_addr)(a'')
{-4}  OK?(memory_read(pm_phy'mem)(a'')(s'))
{-5}  plain_memory?(pm_phy)
{-6}  pm'states(s')
{-7}  OK?(linear_resolve(t'1, Write)(s'))
{-8}  pm_phy'states(state(linear_resolve(t'1, Write)(s')))
{-9}  (address_block(data(linear_resolve(t'1, Write)(s')), length(t'2)) ⊆ pm_phy'rw_addr)
{-10} pm'states = pm_phy'states
{-11} pm'rw_addr(t'1)
{-12} cons?(t'2)
{-13} a' ≤ t'1
{-14} t'1 + length(t'2) ≤ a' + length(bl')
{-15} type_of(t'1) = type_of(a')
{-16} reg_base((#type_of := Mem_, offset := 0#))(type_of(a')) ≤ t'1
{-17} t'1 < reg_size((#type_of := Mem_, offset := max_linear_offset#))(type_of(a'))
{-18} rem(expt(2, min_page))(offset(t'1)) + length(t'2) ≤ expt(2, min_page)
{-19} transformer_invariant?(pm'states,
                              {y |
                                y =
                                  expr_2_super(λ (s : Linear_memory[Physical_memory, pm_phy]):
                                                CASES lin-
ear_resolve(t'1, Write)(s) OF
                                                OK(state, data):
                                                mem-
ory_write_side_effect(pm_phy'mem)
                                                (data, t'2, TRUE)(s),
                                                Exception(ex_type, state):
                                                Exception(ex_type, state),
                                                Fatal: Fatal,
                                                Hang: Hang
                                                ENDCASES)})
{-20} OK?(memory_write_side_effect(pm_phy'mem)
        (data(linear_resolve(t'1, Write)(s')), t'2, TRUE)(s'))
{-21} OK?(OK(state(memory_write_side_effect(pm_phy'mem)
        (data(linear_resolve(t'1, Write)(s')), t'2, TRUE)(s'))))
{-22} OK?(memory_read(pm_phy'mem)
        (a'')
        (state(memory_write_side_effect(pm_phy'mem)
        (data(linear_resolve(t'1, Write)(s')), t'2, TRUE)(s'))))
{-23} (address_block(a', length(bl')) ⊆ pm'rw_addr)
{-24} Mem?(a'type_of)
{-25} 0 ≤ a'offset
{-26} a'offset < max_linear_offset
{-27} cons?[Byte](bl')
-----
{1}  union((pm_phy'other_actions ∪ memory_read_transformers(pm_phy'mem, (pm_phy'ro_addr ∪ pm_phy'rw_addr) ∪
        (memory_read_side_effect_super_transformers(pm_phy'mem, (pm_phy'ro_addr ∪ pm_phy'rw_addr)) ∪
        (expr_2_super(memory_write_side_effect(pm_phy'mem)
        (data(linear_resolve(t'1, Write)(s')), t'2, TRUE)))
{2}  data(memory_read(pm_phy'mem)
        (a'')
        (state(memory_write_side_effect(pm_phy'mem)
        (data(linear_resolve(t'1, Write)(s')), t'2, TRUE)
        (s'))))

```


C Proof scripts

Keeping (-1 1) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `split_linear_write_side_effects_unchanged_memory.1.1.1.1.1.`

split_linear_write_side_effects_unchanged_memory.1.1.1.1.2:

```

{-1}  unchanged_memory_invariant?(pm_phy' mem, pm_phy' states,
      ((pm_phy'other_actions ∪ memory_read_transformers(pm_phy' mem, (pm_phy'ro_addr ∪ pm_phy'rw_addr))
{-2}  union(pm_phy'ro_addr, pm_phy'rw_addr)(a'')
{-3}  OK?(memory_read(pm_phy' mem)(a'')(s'))
{-4}  plain_memory?(pm_phy)
{-5}  pm' states(s')
{-6}  OK?(linear_resolve(t'1, Write)(s'))
{-7}  pm_phy' states(state(linear_resolve(t'1, Write)(s')))
{-8}  (address_block(data(linear_resolve(t'1, Write)(s')), length(t'2)) ⊆ pm_phy'rw_addr)
{-9}  pm' states = pm_phy' states
{-10} pm' rw_addr(t'1)
{-11} cons?(t'2)
{-12} a' ≤ t'1
{-13} t'1 + length(t'2) ≤ a' + length(bl')
{-14} type_of(t'1) = type_of(a')
{-15} reg_base((#type_of := Mem_, offset := 0#))(type_of(a')) ≤ t'1
{-16} t'1 < reg_size((#type_of := Mem_, offset := max_linear_offset#))(type_of(a'))
{-17} rem(expt(2, min_page))(offset(t'1) + length(t'2)) ≤ expt(2, min_page)
{-18} transformer_invariant?(pm' states,
      {y |
        y =
          expr_2_super(λ (s: Linear_memory[Physical_memory, pm_phy]):
            CASES lin-
              ear_resolve(t'1, Write)(s) OF
                OK(state, data):
                  mem-
                    (data, t'2, TRUE)(s),
                  Exception(ex_type, state):
                    Exception(ex_type, state),
                  Fatal: Fatal,
                  Hang: Hang
                ENDCASES)})
{-19} OK?(memory_write_side_effect(pm_phy' mem)
      (data(linear_resolve(t'1, Write)(s')), t'2, TRUE)(s'))
{-20} OK?(OK(state(memory_write_side_effect(pm_phy' mem)
      (data(linear_resolve(t'1, Write)(s')), t'2, TRUE)(s'))))
{-21} OK?(memory_read(pm_phy' mem)
      (a'')
      (state(memory_write_side_effect(pm_phy' mem)
      (data(linear_resolve(t'1, Write)(s')), t'2, TRUE)(s'))))
{-22} (address_block(a', length(bl')) ⊆ pm'rw_addr)
{-23} Mem?(a' type_of)
{-24} 0 ≤ a' offset
{-25} a' offset < max_linear_offset
{-26} cons?[Byte](bl')
-----
{1}  memory_write_side_effect_super_transformers(pm_phy' mem, pm_phy'rw_addr)
      (expr_2_super(memory_write_side_effect(pm_phy' mem)
      (data
        (linear_resolve
          (t'1, Write)(s')),
          t'2,
          TRUE)))
{2}  union((pm_phy'other_actions ∪ memory_read_transformers(pm_phy' mem, (pm_phy'ro_addr ∪ pm_phy'rw_addr) ∪
      (memory_read_side_effect_super_transformers(pm_phy' mem, (pm_phy'ro_addr ∪ pm_phy'rw_addr)) ∪
      (expr_2_super(memory_write_side_effect(pm_phy' mem)
      (data(linear_resolve(t'1, Write)(s')), t'2, TRUE)))
{3}  data(memory_read(pm_phy' mem)
      (a'')
      (state(memory_write_side_effect(pm_phy' mem)
      (data(linear_resolve(t'1, Write)(s')), t'2, TRUE)
      (s'))))

```

C Proof scripts

Hiding formulas: (-1 -18 2 3),

Expanding the definition of `memory_write_side_effect_super_transformers`,

Instantiating quantified variables,

This completes the proof of `split_linear_write_side_effects_unchanged_memory.1.1.1.1.2`.

split_linear_write_side_effects_unchanged_memory.1.1.1.2:

```

{-1} pm' states(s')
{-2} OK?(linear_resolve(t'1, Write)(s'))
{-3} (address_block(data(linear_resolve(t'1, Write)(s')), length(t'2)) ⊆ pm_phy'rw_addr)
{-4} plain_memory?(pm_phy)
{-5} pm' states = pm_phy' states
{-6} pm' rw_addr(t'1)
{-7} cons?(t'2)
{-8} a' ≤ t'1
{-9} t'1 + length(t'2) ≤ a' + length(bl')
{-10} type_of(t'1) = type_of(a')
{-11} reg_base((#type_of := Mem_, offset := 0#))(type_of(a')) ≤ t'1
{-12} t'1 < reg_size((#type_of := Mem_, offset := max_linear_offset#))(type_of(a'))
{-13} rem(expt(2, min_page))(offset(t'1) + length(t'2)) ≤ expt(2, min_page)
{-14} transformer_invariant?(pm' states,
    {y |
      y =
        expr_2_super(λ (s: Linear_memory [Physical_memory, pm_phy]):
          CASES lin-
            ear_resolve(t'1, Write)(s) OF
              OK(state, data):
                mem-
                  ory_write_side_effect(pm_phy' mem)
                    (data, t'2, TRUE)(s),
                    Exception(ex_type, state):
                      Exception(ex_type, state),
                      Fatal: Fatal,
                      Hang: Hang
                    ENDCASES)})
{-15} union(pm_phy' ro_addr, pm_phy' rw_addr)(a'')
{-16} OK?(memory_write_side_effect(pm_phy' mem)
    (data(linear_resolve(t'1, Write)(s')), t'2, TRUE)(s'))
{-17} OK?(OK(state(memory_write_side_effect(pm_phy' mem)
    (data(linear_resolve(t'1, Write)(s')), t'2, TRUE)(s'))))
{-18} OK?(memory_read(pm_phy' mem)(a'')(s'))
{-19} OK?(memory_read(pm_phy' mem)
    (a'')
    (state(memory_write_side_effect(pm_phy' mem)
    (data(linear_resolve(t'1, Write)(s')), t'2, TRUE)(s'))))
{-20} (address_block(a', length(bl')) ⊆ pm'rw_addr)
{-21} Mem?(a' type_of)
{-22} 0 ≤ a' offset
{-23} a' offset < max_linear_offset
{-24} cons?[Byte](bl')
-----
{1} union(pm' ro_addr, pm' rw_addr)(t'1)
{2} data(memory_read(pm_phy' mem)
    (a'')
    (state(memory_write_side_effect(pm_phy' mem)
    (data(linear_resolve(t'1, Write)(s')), t'2, TRUE)
    (s'))))
    = data(memory_read(pm_phy' mem)(a'')(s'))

```

Keeping (-6 1) and hiding *,

C Proof scripts

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `split_linear_write_side_effects_unchanged_memory.1.1.1.2`.

split_linear_write_side_effects_unchanged_memory.1.1.2:

```

{-1}  is_linear_plain_memory?(pm')  $\supset$  OK?(linear_resolve(t'1, Write)(s'))
{-2}  plain_memory?(pm_phy)
{-3}  pm' states = pm_phy states
{-4}  pm' rw_addr(t'1)
{-5}  cons?(t'2)
{-6}   $a' \leq t'1$ 
{-7}   $t'1 + \text{length}(t'2) \leq a' + \text{length}(bl')$ 
{-8}  type_of(t'1) = type_of(a')
{-9}  reg_base((#type_of := Mem_, offset := 0#))(type_of(a'))  $\leq t'1$ 
{-10}  $t'1 < \text{reg\_size}((\#type\_of := \text{Mem\_}, \text{offset} := \text{max\_linear\_offset\#}))(type\_of(a'))$ 
{-11}  $\text{rem}(\text{expt}(2, \text{min\_page}))(\text{offset}(t'1) + \text{length}(t'2)) \leq \text{expt}(2, \text{min\_page})$ 
{-12} transformer_invariant?(pm' states,
      {y |
        y =
          expr_2_super( $\lambda$  (s: Linear_memory[Physical_memory, pm_phy]):
            CASES lin-
ear_resolve(t'1, Write)(s) OF
      OK(state, data):
        mem-
ory_write_side_effect(pm_phy mem)
          (data, t'2, TRUE)(s),
        Exception(ex_type, state):
          Exception(ex_type, state),
        Fatal: Fatal,
        Hang: Hang
        ENDCASES})
{-13} pm' states(s')
{-14} union(pm_phy ro_addr, pm_phy rw_addr)(a'')
{-15} OK?(expr_2_super( $\lambda$  (s: Linear_memory[Physical_memory, pm_phy]):
      CASES linear_resolve(t'1, Write)(s) OF
        OK(state, data):
          memory_write_side_effect(pm_phy mem)(data, t'2, TRUE)(s),
          Exception(ex_type, state): Exception(ex_type, state),
          Fatal: Fatal,
          Hang: Hang
          ENDCASES)
      (s'))
{-16} OK?(memory_read(pm_phy mem)(a'')(s'))
{-17} OK?(memory_read(pm_phy mem)
      (a'')
      (state(expr_2_super( $\lambda$  (s: Linear_memory[Physical_memory, pm_phy]):
        CASES linear_resolve(t'1, Write)(s) OF
          OK(state, data):
            memory_write_side_effect(pm_phy mem)
              (data, t'2, TRUE)(s),
            Exception(ex_type, state): Excep-
tion(ex_type, state),
          Fatal: Fatal,
          Hang: Hang
          ENDCASES)
      (s'))))
{-18} (address_block(a', length(bl'))  $\subseteq$  pm' rw_addr)
{-19} Mem?(a' type_of)
{-20}  $0 \leq a'$  offset
{-21}  $a'$  offset < max_linear_offset
{-22} cons?[Byte](bl')

```

```

{1}  (address_block(data(linear_resolve(t'1, Write)(s')), length(t'2))  $\subseteq$  pm_phy rw_addr)
{2}  data(memory_read(pm_phy mem)
      (a'')
      (state(expr_2_super( $\lambda$  (s: Linear_memory[Physical_memory, pm_phy]):
        CASES linear_resolve(t'1, Write)(s) OF
          OK(state, data):
            memory_write_side_effect(pm_phy mem)

```

C Proof scripts

Hiding formulas: 2,

Using lemma `same_page_address_block_write`,

we get 3 subgoals:

split_linear_write_side_effects_unchanged_memory.1.1.2.1:

```

{-1}  is_linear_plain_memory?(pm') ∧
      pm' states(s') ∧
      pm' rw_addr(t'1) ∧
      a' ≤ t'1 ∧
      t'1 + length(t'2) ≤ a' + length(bl') ∧
      rem(expt(2, min_page))(offset(t'1)) + length(t'2) ≤ expt(2, min_page)
      ∧ (address_block(a', length(bl')) ⊆ pm'rw_addr)
      ⊃
      (address_block(data(linear_resolve(t'1, Write)(s')), length(t'2)) ⊆ pm_phy'rw_addr)
{-2}  is_linear_plain_memory?(pm') ⊃ OK?(linear_resolve(t'1, Write)(s'))
{-3}  plain_memory?(pm_phy)
{-4}  pm' states = pm_phy states
{-5}  pm' rw_addr(t'1)
{-6}  cons?(t'2)
{-7}  a' ≤ t'1
{-8}  t'1 + length(t'2) ≤ a' + length(bl')
{-9}  type_of(t'1) = type_of(a')
{-10} reg_base((#type_of := Mem_, offset := 0#))(type_of(a')) ≤ t'1
{-11} t'1 < reg_size((#type_of := Mem_, offset := max_linear_offset#))(type_of(a'))
{-12} rem(expt(2, min_page))(offset(t'1)) + length(t'2) ≤ expt(2, min_page)
{-13} transformer_invariant?(pm' states,
                               {y |
                                 y =
                                   expr_2_super(λ (s: Linear_memory[Physical_memory, pm_phy]):
                                                 CASES lin-
ear_resolve(t'1, Write)(s) OF
                                     OK(state, data):
                                     mem-
ory_write_side_effect(pm_phy mem)
                                     (data, t'2, TRUE)(s),
                                     Exception(ex_type, state):
                                     Exception(ex_type, state),
                                     Fatal: Fatal,
                                     Hang: Hang
                                     ENDCASES)})
{-14} pm' states(s')
{-15} union(pm_phy'ro_addr, pm_phy'rw_addr)(a'')
{-16} OK?(expr_2_super(λ (s: Linear_memory[Physical_memory, pm_phy]):
                       CASES linear_resolve(t'1, Write)(s) OF
                               OK(state, data):
                               memory_write_side_effect(pm_phy mem)(data, t'2, TRUE)(s),
                               Exception(ex_type, state): Exception(ex_type, state),
                               Fatal: Fatal,
                               Hang: Hang
                               ENDCASES)
                       (s'))
{-17} OK?(memory_read(pm_phy mem)(a'')(s'))
{-18} OK?(memory_read(pm_phy mem)
                    (a'')
                    (state(expr_2_super(λ (s: Linear_memory[Physical_memory, pm_phy]):
                                         CASES linear_resolve(t'1, Write)(s) OF
                                                 OK(state, data):
                                                 memory_write_side_effect(pm_phy mem)
                                                 (data, t'2, TRUE)(s),
                                                 Exception(ex_type, state): Excep-
tion(ex_type, state),
                                         Fatal: Fatal,
                                         Hang: Hang
                                         ENDCASES)
                                         (s'))))
{-19} (address_block(a', length(bl')) ⊆ pm'rw_addr)
{-20} Mem?(a' type_of)
{-21} 0 ≤ a' offset

```


C Proof scripts

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `split_linear_write_side_effects_unchanged_memory.1.1.2.1`.

split_linear_write_side_effects_unchanged_memory.1.1.2.2:

```

{-1}  is_linear_plain_memory?(pm')  $\supset$  OK?(linear_resolve(t'1, Write)(s'))
{-2}  plain_memory?(pm_phy)
{-3}  pm' states = pm_phy states
{-4}  pm' rw_addr(t'1)
{-5}  cons?(t'2)
{-6}  a'  $\leq$  t'1
{-7}  t'1 + length(t'2)  $\leq$  a' + length(bl')
{-8}  type_of(t'1) = type_of(a')
{-9}  reg_base((#type_of := Mem_, offset := 0#))(type_of(a'))  $\leq$  t'1
{-10} t'1 < reg_size((#type_of := Mem_, offset := max_linear_offset#))(type_of(a'))
{-11} rem(expt(2, min_page))(offset(t'1) + length(t'2))  $\leq$  expt(2, min_page)
{-12} transformer_invariant?(pm' states,
      {y |
        y =
          expr_2_super( $\lambda$  (s: Linear_memory[Physical_memory, pm_phy]):
            CASES lin-
ear_resolve(t'1, Write)(s) OF
      OK(state, data):
        mem-
ory_write_side_effect(pm_phy mem)
          (data, t'2, TRUE)(s),
        Exception(ex_type, state):
          Exception(ex_type, state),
        Fatal: Fatal,
        Hang: Hang
        ENDCASES)})
{-13} pm' states(s')
{-14} union(pm_phy ro_addr, pm_phy rw_addr)(a'')
{-15} OK?(expr_2_super( $\lambda$  (s: Linear_memory[Physical_memory, pm_phy]):
      CASES linear_resolve(t'1, Write)(s) OF
        OK(state, data):
          memory_write_side_effect(pm_phy mem)(data, t'2, TRUE)(s),
        Exception(ex_type, state): Exception(ex_type, state),
        Fatal: Fatal,
        Hang: Hang
        ENDCASES)
      (s'))
{-16} OK?(memory_read(pm_phy mem)(a'')(s'))
{-17} OK?(memory_read(pm_phy mem)
      (a'')
      (state(expr_2_super( $\lambda$  (s: Linear_memory[Physical_memory, pm_phy]):
        CASES linear_resolve(t'1, Write)(s) OF
          OK(state, data):
            memory_write_side_effect(pm_phy mem)
              (data, t'2, TRUE)(s),
          Exception(ex_type, state): Excep-
tion(ex_type, state),
          Fatal: Fatal,
          Hang: Hang
          ENDCASES)
        (s'))))
{-18} (address_block(a', length(bl'))  $\subseteq$  pm' rw_addr)
{-19} Mem?(a' type_of)
{-20} 0  $\leq$  a' offset
{-21} a' offset < max_linear_offset
{-22} cons?[Byte](bl')

```

C Proof scripts

Expanding the definition of length,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `split_linear_write_side_effects_unchanged_memory.1.1.2.2`.

split_linear_write_side_effects_unchanged_memory.1.1.2.3:

```

{-1}  is_linear_plain_memory?(pm') ⊃ OK?(linear_resolve(t'1, Write)(s'))
{-2}  plain_memory?(pm_phy)
{-3}  pm' states = pm_phy states
{-4}  pm' rw_addr(t'1)
{-5}  cons?(t'2)
{-6}  a' ≤ t'1
{-7}  t'1 + length(t'2) ≤ a' + length(bl')
{-8}  type_of(t'1) = type_of(a')
{-9}  reg_base((#type_of := Mem_, offset := 0#))(type_of(a')) ≤ t'1
{-10} t'1 < reg_size((#type_of := Mem_, offset := max_linear_offset#))(type_of(a'))
{-11} rem(expt(2, min_page))(offset(t'1) + length(t'2)) ≤ expt(2, min_page)
{-12} transformer_invariant?(pm' states,
      {y |
        y =
          expr_2_super(λ (s: Linear_memory[Physical_memory, pm_phy]):
            CASES lin-
ear_resolve(t'1, Write)(s) OF
      OK(state, data):
        mem-
ory_write_side_effect(pm_phy mem)
          (data, t'2, TRUE)(s),
        Exception(ex_type, state):
          Exception(ex_type, state),
        Fatal: Fatal,
        Hang: Hang
        ENDCASES})
{-13} pm' states(s')
{-14} union(pm_phy ro_addr, pm_phy rw_addr)(a'')
{-15} OK?(expr_2_super(λ (s: Linear_memory[Physical_memory, pm_phy]):
      CASES linear_resolve(t'1, Write)(s) OF
        OK(state, data):
          memory_write_side_effect(pm_phy mem)(data, t'2, TRUE)(s),
        Exception(ex_type, state): Exception(ex_type, state),
        Fatal: Fatal,
        Hang: Hang
        ENDCASES)
      (s'))
{-16} OK?(memory_read(pm_phy mem)(a'')(s'))
{-17} OK?(memory_read(pm_phy mem)
      (a'')
      (state(expr_2_super(λ (s: Linear_memory[Physical_memory, pm_phy]):
        CASES linear_resolve(t'1, Write)(s) OF
          OK(state, data):
            memory_write_side_effect(pm_phy mem)
              (data, t'2, TRUE)(s),
          Exception(ex_type, state): Excep-
tion(ex_type, state),
          Fatal: Fatal,
          Hang: Hang
          ENDCASES)
        (s'))))
{-18} (address_block(a', length(bl')) ⊆ pm' rw_addr)
{-19} Mem?(a' type_of)
{-20} 0 ≤ a' offset
{-21} a' offset < max_linear_offset
{-22} cons?[Byte](bl')
      2315
{1}  length[Byte](bl') > 0
{2}  (address_block(data(linear_resolve(t'1, Write)(s')), length(t'2)) ⊆ pm_phy rw_addr)

```

C Proof scripts

Expanding the definition of length,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `split_linear_write_side_effects_unchanged_memory.1.1.2.3`.

split_linear_write_side_effects_unchanged_memory.1.1.3:

```

{-1}  is_linear_plain_memory?(pm') ⊃ OK?(linear_resolve(t'1, Write)(s'))
{-2}  plain_memory?(pm_phy)
{-3}  pm' states = pm_phy states
{-4}  pm' rw_addr(t'1)
{-5}  cons?(t'2)
{-6}  a' ≤ t'1
{-7}  t'1 + length(t'2) ≤ a' + length(bl')
{-8}  type_of(t'1) = type_of(a')
{-9}  reg_base((#type_of := Mem_, offset := 0#))(type_of(a')) ≤ t'1
{-10} t'1 < reg_size((#type_of := Mem_, offset := max_linear_offset#))(type_of(a'))
{-11} rem(expt(2, min_page))(offset(t'1) + length(t'2)) ≤ expt(2, min_page)
{-12} transformer_invariant?(pm' states,
      {y |
        y =
          expr_2_super(λ (s: Linear_memory[Physical_memory, pm_phy]):
            CASES lin-
ear_resolve(t'1, Write)(s) OF
      OK(state, data):
        mem-
ory_write_side_effect(pm_phy mem)
          (data, t'2, TRUE)(s),
        Exception(ex_type, state):
          Exception(ex_type, state),
        Fatal: Fatal,
        Hang: Hang
        ENDCASES})
{-13} pm' states(s')
{-14} union(pm_phy ro_addr, pm_phy rw_addr)(a'')
{-15} OK?(expr_2_super(λ (s: Linear_memory[Physical_memory, pm_phy]):
      CASES linear_resolve(t'1, Write)(s) OF
        OK(state, data):
          memory_write_side_effect(pm_phy mem)(data, t'2, TRUE)(s),
          Exception(ex_type, state): Exception(ex_type, state),
          Fatal: Fatal,
          Hang: Hang
          ENDCASES)
      (s'))
{-16} OK?(memory_read(pm_phy mem)(a'')(s'))
{-17} OK?(memory_read(pm_phy mem)
      (a'')
      (state(expr_2_super(λ (s: Linear_memory[Physical_memory, pm_phy]):
        CASES linear_resolve(t'1, Write)(s) OF
          OK(state, data):
            memory_write_side_effect(pm_phy mem)
              (data, t'2, TRUE)(s),
            Exception(ex_type, state): Excep-
tion(ex_type, state),
          Fatal: Fatal,
          Hang: Hang
          ENDCASES)
      (s'))))
{-18} (address_block(a', length(bl')) ⊆ pm' rw_addr)
{-19} Mem?(a' type_of)
{-20} 0 ≤ a' offset
{-21} a' offset < max_linear_offset
{-22} cons?[Byte](bl')

```

```

{1}  OK?[Linear_memory[Physical_memory, pm_phy], Address]
      (linear_resolve[Physical_memory, pm_phy](t'1, Write)(s'))
{2}  data(memory_read(pm_phy mem)
      (a'')
      (state(expr_2_super(λ (s: Linear_memory[Physical_memory, pm_phy]):
        CASES linear_resolve(t'1, Write)(s) OF
          OK(state, data):

```

C Proof scripts

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `split_linear_write_side_effects_unchanged_memory.1.1.3`.

`split_linear_write_side_effects_unchanged_memory.1.2`:

<pre> {-1} cons?(t'2) {-2} a' ≤ t'1 {-3} t'1 + length(t'2) ≤ a' + length(bl') {-4} type_of(t'1) = type_of(a') {-5} reg_base(#type_of := Mem_, offset := 0#)(type_of(a')) ≤ t'1 {-6} t'1 < reg_size(#type_of := Mem_, offset := max_linear_offset#)(type_of(a')) {-7} rem(expt(2, min_page))(offset(t'1)) + length(t'2) ≤ expt(2, min_page) {-8} transformer_invariant?(pm' states, singleton(expr_2_super(λ (s: ear_memory[Physical_memory, pm_phy]: ory_Address_4G) ory_write_side_effect(pm_phy mem) {-9} (address_block(a', length(bl')) ⊆ pm'rw_addr) {-10} Mem?(a' type_of) {-11} 0 ≤ a' offset {-12} a' offset < max_linear_offset {-13} cons?[Byte](bl') </pre>	<pre> Lin- (linear_resolve(t'1, Write) ## (λ (pa: Mem- (ns: Linear_memory [Physical_memory, pm_ mem- (pa, t'2, TRUE)(s))) (s)))) </pre>
<pre> {1} pm'rw_addr(t'1) {2} unchanged_memory_invariant?(pm_phy mem, pm' states, singleton(expr_2_super(λ (s: </pre>	<pre> Linear_memory [Physical_memory, pm_ (linear_resolve(t'1, Write) # (λ (pa: Mem- (ns: Linear_memory [Physical_memory, pm_ mem- (pa, t'2, TRUE)(s))) (s))), (pm_phy ro_addr ∪ pm_phy rw_addr) </pre>

Hiding formulas: (-8 -7 2),

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `split_linear_write_side_effects_unchanged_memory.1.2`.

split_linear_write_side_effects_unchanged_memory.2:

{-1}	is_linear_plain_memory?(pm')
{-2}	(address_block(a', length(bl')) \subseteq pm'rw_addr)
{-3}	every(λ (e: [Memory_Address_4G, list[Byte]]): transformer_invariant?(pm' states, singleton(expr_2_super(linear_write_side_effect_in_page(e)))) (split(min_page, a', bl'))
{-4}	every(λ (a2: Address, bl2: list[Byte]): rem(expt(2, min_page))(offset(a2)) + length(bl2) \leq expt(2, min_page)) (split(min_page, a', bl'))
{-5}	every(λ (a2: Address, bl2: list[Byte]): cons?(bl2))(split(min_page, a', bl'))
{-6}	every(λ (a2: Address, bl2: list[Byte]): a' \leq a2 \wedge a2 + length(bl2) \leq a' + length(bl')) (split(min_page, a', bl'))
{-7}	Mem?(a' type_of)
{-8}	0 \leq a' offset
{-9}	a' offset < max_linear_offset
{-10}	every(λ (x: number): number_field_pred(x) \wedge real_pred(x) \wedge rational_pred(x) \wedge integer_pred(x) \wedge x \geq 0 \wedge x < max_byte) (bl')
{-11}	cons?[Byte](bl')
{1}	a' + length(bl') \leq reg_size((#type_of := Mem_, offset := max_linear_offset#))(type_of(a'))
{2}	every(λ (e: [Memory_Address_4G, list[Byte]]): unchanged_memory_invariant?(pm_phy mem, pm' states, singleton(expr_2_super(linear_write_side_effect_in_page (e))), (pm_phy ro_addr \cup pm_phy rw_addr))) (split(min_page, a', bl'))

Hiding formulas: (-3 -4 -5 -6 -10 2),

Expanding the definition of length,

Using lemma pm_memory_addr,

Simplifying, rewriting, and recording with decision procedures,

Hiding formulas: -2,

Expanding the definition of subset?,

Installing automatic rewrites from: (Mem! max_linear! bus_width!)

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of split_linear_write_side_effects_unchanged_memory.2.

split_linear_write_side_effects_unchanged_memory.3:

<pre> {-1} is_linear_plain_memory?(pm') {-2} (address_block(a', length(bl')) ⊆ pm'rw_addr) {-3} every(λ (e: [Memory_Address_4G, list[Byte]]): transformer_invariant?(pm' states, singleton(expr_2_super(linear_write_side_effect_in_page(e)) (split(min_page, a', bl')))) {-4} every(λ (a2: Address, bl2: list[Byte]): rem(expt(2, min_page))(offset(a2)) + length(bl2) ≤ expt(2, min_page)) (split(min_page, a', bl')) {-5} every(λ (a2: Address, bl2: list[Byte]): cons?(bl2))(split(min_page, a', bl')) {-6} every(λ (a2: Address, bl2: list[Byte]): a' ≤ a2 ∧ a2 + length(bl2) ≤ a' + length(bl')) (split(min_page, a', bl')) {-7} Mem?(a' type_of) {-8} 0 ≤ a' offset {-9} a' offset < max_linear_offset {-10} every(λ (x: number): number_field_pred(x) ∧ real_pred(x) ∧ rational_pred(x) ∧ integer_pred(x) ∧ x ≥ 0 ∧ x < max_byte) (bl') {-11} cons?[Byte](bl') </pre>	<pre> {1} reg_base((#type_of := Mem_, offset := 0#))(type_of(a')) ≤ a' {2} every(λ (e: [Memory_Address_4G, list[Byte]]): unchanged_memory_invariant?(pm_phy mem, pm' states, singleton(expr_2_super(linear_write_side_effect_in_page (e))), (pm_phy ro_addr ∪ pm_phy rw_addr))) (split(min_page, a', bl')) </pre>
--	--

Expanding the definition of reg_base,

Expanding the definition of ≤,

which is trivially true.

This completes the proof of split_linear_write_side_effects_unchanged_memory.3.

Q.E.D.

C.117.84 Linear_Memory_Properties.linear_unchanged_invariant

Terse proof for linear_unchanged_invariant.

linear_unchanged_invariant.1:

```

{-1} transformer_invariant?(pm' states, transformers')
{-2} is_linear_plain_memory?(pm')  $\supset$  pm' states = pm_phy states
{-3} is_linear_plain_memory?(pm')
{-4} (addresses'  $\subseteq$  restrict[Address, Memory_Address_4G, boolean]((pm' ro_addr  $\cup$  pm' rw_addr)))
{-5}  $\forall$  (s: (pm' states)):
      transformer_invariant?(pm_phy states, transformers')  $\wedge$ 
      ( $\forall$  (s_1: Physical_memory, q: [Physical_memory  $\rightarrow$  SuperResult[Physical_memory]],
        a: Address):
        pm_phy states(s_1)  $\wedge$ 
        transformers'(q)  $\wedge$ 
        extend[Address, Memory_Address_4G, bool, FALSE]
          (virt_to_phys_range(s, addresses',
            (singleton(Read)  $\cup$  singleton(Execute))))
          (a)
           $\wedge$ 
          OK?(q(s_1))  $\wedge$ 
          OK?(memory_read(pm_phy mem)(a)(s_1))  $\wedge$ 
          OK?(memory_read(pm_phy mem)(a)(state(q(s_1))))
           $\supset$ 
          data(memory_read(pm_phy mem)(a)(state(q(s_1)))) =
          data(memory_read(pm_phy mem)(a)(s_1)))


---


{1} transformer_invariant?(pm' states, transformers')  $\wedge$ 
      ( $\forall$  (s: Linear_memory[Physical_memory, pm_phy],
        q: [Linear_memory[Physical_memory, pm_phy]  $\rightarrow$  SuperResult[Physical_memory]],
        a: Address):
        pm' states(s)  $\wedge$ 
        transformers'(q)  $\wedge$ 
        extend[Address, Memory_Address_4G, bool, FALSE](addresses')(a)  $\wedge$ 
        OK?(q(s))  $\wedge$ 
        OK?(memory_read(pm' mem)(a)(s))  $\wedge$ 
        OK?(memory_read(pm' mem)(a)(state(q(s))))
         $\supset$ 
        data(memory_read(pm' mem)(a)(state(q(s)))) =
        data(memory_read(pm' mem)(a)(s)))

```

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Repeatedly Skolemizing and flattening,

Rewriting using pm_read_linear, matching in *,

Expanding the definition of linear_read,

Installing automatic rewrites from: (expr_2_super! expr_2_super_res! ##! singleton!)

Case splitting on union(pm!1'ro_addr, pm!1'rw_addr)(a!1),

we get 2 subgoals:

linear_unchanged_invariant.1.1:

```

{-1} union(pm'ro_addr, pm'rw_addr)(a')
{-2} transformer_invariant?(pm'states, transformers')
{-3} is_linear_plain_memory?(pm')
{-4} pm'states = pm_phy'states
{-5} (addresses' ⊆ restrict[Address, Memory_Address_4G, boolean]((pm'ro_addr ∪ pm'rw_addr)))
{-6} ∀ (s: (pm'states)):
    ∀ (s_1: Physical_memory, q: [Physical_memory → SuperResult[Physical_memory]],
        a: Address):
        pm_phy'states(s_1) ∧
        transformers'(q) ∧
        extend[Address, Memory_Address_4G, bool, FALSE]
            (virt_to_phys_range(s, addresses', (singleton(Read) ∪ singleton(Execute)))
            (a)
            ∧
            OK?(q(s_1)) ∧
            OK?(memory_read(pm_phy'mem)(a)(s_1)) ∧
            OK?(memory_read(pm_phy'mem)(a)(state(q(s_1))))
        ⊃
        data(memory_read(pm_phy'mem)(a)(state(q(s_1)))) =
        data(memory_read(pm_phy'mem)(a)(s_1))
{-7} pm'states(s')
{-8} transformers'(q')
{-9} extend[Address, Memory_Address_4G, bool, FALSE](addresses')(a')
{-10} OK?(q'(s'))
{-11} OK?(IF in_memory(min_linear, max_linear)(a')
    THEN IF Mem?(type_of(a'))
        THEN (linear_resolve(a', Read) ##
            (λ (pa: Address): memory_read(pm_phy'mem)(pa)))
            (s')
        ELSE memory_read(pm_phy'mem)(a')(s')
    ENDIF
    ELSE fatal_result(s')
    ENDIF)
{-12} OK?(IF in_memory(min_linear, max_linear)(a')
    THEN IF Mem?(type_of(a'))
        THEN (linear_resolve(a', Read) ##
            (λ (pa: Address): memory_read(pm_phy'mem)(pa)))
            (state(q'(s')))
        ELSE memory_read(pm_phy'mem)(a')(state(q'(s')))
    ENDIF
    ELSE fatal_result(state(q'(s')))
    ENDIF)


---


{1} IF in_memory(min_linear, max_linear)(a')
    THEN IF Mem?(type_of(a'))
        THEN data((linear_resolve(a', Read) ##
            (λ (pa: Address): memory_read(pm_phy'mem)(pa)))
            (state(q'(s'))))
        ELSE data(memory_read(pm_phy'mem)(a')(state(q'(s'))))
    ENDIF
    ELSE data(fatal_result(state(q'(s'))))
    ENDIF
    =
    IF in_memory(min_linear, max_linear)(a')
        THEN IF Mem?(type_of(a'))
            THEN data((linear_resolve(a', Read) ##
                (λ (pa: Address): memory_read(pm_phy'mem)(pa)))
                (s'))
            ELSE data(memory_read(pm_phy'mem)(a')(s'))
        ENDIF
    ELSE data(fatal_result(s'))
    ENDIF

```

C Proof scripts

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting, we get 2 subgoals:

`linear_unchanged_invariant.1.1.1:`

<pre> {-1} union(pm'ro_addr, pm'rw_addr)(a') {-2} transformer_invariant?(pm'states, transformers') {-3} is_linear_plain_memory?(pm') {-4} pm'states = pm_phy'states {-5} (addresses' ⊆ restrict[Address, Memory_Address_4G, boolean]((pm'ro_addr ∪ pm'rw_addr))) {-6} ∀ (s: (pm'states)): ∀ (s_1: Physical_memory, q: [Physical_memory → SuperResult[Physical_memory]], a: Address): pm_phy'states(s_1) ∧ transformers'(q) ∧ extend[Address, Memory_Address_4G, bool, FALSE] (virt_to_phys_range(s, addresses', (singleton(Read) ∪ singleton(Execute))) (a)) ∧ OK?(q(s_1)) ∧ OK?(memory_read(pm_phy'mem)(a)(s_1)) ∧ OK?(memory_read(pm_phy'mem)(a)(state(q(s_1)))) ⊃ data(memory_read(pm_phy'mem)(a)(state(q(s_1)))) = data(memory_read(pm_phy'mem)(a)(s_1)) </pre>	<pre> {-7} pm'states(s') {-8} transformers'(q') {-9} extend[Address, Memory_Address_4G, bool, FALSE](addresses')(a') {-10} OK?(q'(s')) {-11} in_memory(min_linear, max_linear)(a') {-12} Mem?(type_of(a')) {-13} OK?(linear_resolve(a', Read)(s')) {-14} OK?(memory_read(pm_phy'mem) (data(linear_resolve(a', Read)(s'))) (state(linear_resolve(a', Read)(s')))) {-15} OK?(linear_resolve(a', Read)(state(q'(s')))) {-16} OK?(memory_read(pm_phy'mem) (data(linear_resolve(a', Read)(state(q'(s')))) (state(linear_resolve(a', Read)(state(q'(s')))))) </pre> <hr style="border: 0.5px solid black;"/> <pre> {1} data(memory_read(pm_phy'mem) (data(linear_resolve(a', Read)(state(q'(s')))) (state(linear_resolve(a', Read)(state(q'(s')))))) = data(memory_read(pm_phy'mem) (data(linear_resolve(a', Read)(s')) (state(linear_resolve(a', Read)(s')))) </pre>
--	---

Using lemma `super_transformer_invariant_next_ok`,

Using lemma `linear_resolve_same_result`,

Using lemma `linear_resolve_unchanged_pm_phy`,

Using lemma `unchanged_memory_invariant_unchanged`,

Using lemma `unchanged_memory_invariant_unchanged`,

Case splitting on `union(pm_phy'ro_addr, pm_phy'rw_addr) (data(linear_resolve(a!1, Read)(s!1)))`,

we get 2 subgoals:

linear_unchanged_invariant.1.1.1.1.1:

```

{-1} union(pm_phy'ro_addr, pm_phy'rw_addr)(data(linear_resolve(a', Read)(s')))
{-2} unchanged_memory_invariant?(pm_phy' mem, pm' states,
                                singleton(expr_2_super[Physical_memory, Address]
                                           (linear_resolve(a', Read))),
                                ((pm_phy'ro_addr ∪ pm_phy'rw_addr) \ extend [Address, Memory_Address
                                ^
                                pm' states(state(q'(s'))) ∧
                                singleton(expr_2_super[Physical_memory, Address](linear_resolve(a', Read)))
                                (expr_2_super(linear_resolve(a', Read)))
                                ^
                                difference((pm_phy'ro_addr ∪ pm_phy'rw_addr),
                                extend[Address, Memory_Address_4G, bool, FALSE]
                                (address_in_pt_range?(s',
                                restrict[Address, Mem-
                                ory_Address_4G, boolean]
                                ((pm'ro_addr ∪ pm'rw_addr))))
                                (data(linear_resolve(a', Read)(s')))
                                ^
                                OK?(expr_2_super(linear_resolve(a', Read))(state(q'(s')))) ∧
                                OK?(memory_read(pm_phy' mem)
                                (data(linear_resolve(a', Read)(s')))(state(q'(s'))))
                                ^
                                OK?(memory_read(pm_phy' mem)
                                (data(linear_resolve(a', Read)(s')))
                                (state(expr_2_super(linear_resolve(a', Read))(state(q'(s'))))))
                                ⊃
                                data(memory_read(pm_phy' mem)
                                (data(linear_resolve(a', Read)(s')))
                                (state(expr_2_super(linear_resolve(a', Read))(state(q'(s'))))))
                                =
                                data(memory_read(pm_phy' mem)
                                (data(linear_resolve(a', Read)(s')))(state(q'(s'))))
{-3} unchanged_memory_invariant?(pm_phy' mem, pm' states,
                                singleton(expr_2_super[Physical_memory, Address]
                                           (linear_resolve(a', Read))),
                                ((pm_phy'ro_addr ∪ pm_phy'rw_addr) \ extend [Address, Memory_Address
                                ^
                                pm' states(s') ∧
                                singleton(expr_2_super[Physical_memory, Address](linear_resolve(a', Read)))
                                (expr_2_super(linear_resolve(a', Read)))
                                ^
                                difference((pm_phy'ro_addr ∪ pm_phy'rw_addr),
                                extend[Address, Memory_Address_4G, bool, FALSE]
                                (address_in_pt_range?(s',
                                restrict[Address, Mem-
                                ory_Address_4G, boolean]
                                ((pm'ro_addr ∪ pm'rw_addr))))
                                (data(linear_resolve(a', Read)(s')))
                                ^
                                OK?(expr_2_super(linear_resolve(a', Read))(s')) ∧
                                OK?(memory_read(pm_phy' mem)(data(linear_resolve(a', Read)(s')))(s')) ∧
                                OK?(memory_read(pm_phy' mem)
                                (data(linear_resolve(a', Read)(s')))
                                (state(expr_2_super(linear_resolve(a', Read))(s'))))
                                ⊃
                                data(memory_read(pm_phy' mem)
                                (data(linear_resolve(a', Read)(s')))
                                (state(expr_2_super(linear_resolve(a', Read))(s'))))
                                = data(memory_read(pm_phy' mem)(data(linear_resolve(a', Read)(s')))(s'))
{-4} union(pm'ro_addr, pm'rw_addr)(a') ∧ is_linear_plain_memory?(pm') ∧ pm' states(s')
                                ⊃
                                unchanged_memory_invariant?[Physical_memory]
                                (pm_phy' mem, pm' states,

```

C Proof scripts

Case splitting on $\text{OK?}(\text{memory_read}(\text{pm_phy' mem})(\text{data}(\text{linear_resolve}(\text{a!1}, \text{Read})(\text{s!1}))))(\text{s!1}),$
 $\text{OK?}(\text{memory_read}(\text{pm_phy' mem})(\text{data}(\text{linear_resolve}(\text{a!1}, \text{Read})(\text{s!1}))))(\text{state}(\text{q!1}(\text{s!1}))))),$

we get 3 subgoals:

linear_unchanged_invariant.1.1.1.1.1:

```

{-1} OK?(memory_read(pm_phy' mem)
      (data(linear_resolve(a', Read)(s'))(state(q'(s')))))
{-2} OK?(memory_read(pm_phy' mem)(data(linear_resolve(a', Read)(s'))(s')))
{-3} union(pm_phy' ro_addr, pm_phy' rw_addr)(data(linear_resolve(a', Read)(s')))
{-4} unchanged_memory_invariant?(pm_phy' mem, pm' states,
      singleton(expr_2_super[Physical_memory, Address]
                (linear_resolve(a', Read))),
      ((pm_phy' ro_addr ∪ pm_phy' rw_addr) \ extend [Address, Memory_Address
      ^
      pm' states(state(q'(s'))) ∧
      singleton(expr_2_super[Physical_memory, Address](linear_resolve(a', Read))
                (expr_2_super(linear_resolve(a', Read))))
      ^
      difference((pm_phy' ro_addr ∪ pm_phy' rw_addr),
                extend[Address, Memory_Address_4G, bool, FALSE]
                (address_in_pt_range?(s',
                restrict[Address, Mem-
memory_Address_4G, boolean]
                ((pm' ro_addr ∪ pm' rw_addr))))
                (data(linear_resolve(a', Read)(s'))))
      ^
      OK?(expr_2_super(linear_resolve(a', Read))(state(q'(s')))) ∧
      OK?(memory_read(pm_phy' mem)
                (data(linear_resolve(a', Read)(s'))(state(q'(s')))))
      ^
      OK?(memory_read(pm_phy' mem)
                (data(linear_resolve(a', Read)(s'))
                (state(expr_2_super(linear_resolve(a', Read))(state(q'(s'))))))))
      ⊃
      data(memory_read(pm_phy' mem)
            (data(linear_resolve(a', Read)(s'))
            (state(expr_2_super(linear_resolve(a', Read))(state(q'(s'))))))))
      =
      data(memory_read(pm_phy' mem)
            (data(linear_resolve(a', Read)(s'))(state(q'(s')))))
{-5} unchanged_memory_invariant?(pm_phy' mem, pm' states,
      singleton(expr_2_super[Physical_memory, Address]
                (linear_resolve(a', Read))),
      ((pm_phy' ro_addr ∪ pm_phy' rw_addr) \ extend [Address, Memory_Address
      ^
      pm' states(s') ∧
      singleton(expr_2_super[Physical_memory, Address](linear_resolve(a', Read))
                (expr_2_super(linear_resolve(a', Read))))
      ^
      difference((pm_phy' ro_addr ∪ pm_phy' rw_addr),
                extend[Address, Memory_Address_4G, bool, FALSE]
                (address_in_pt_range?(s',
                restrict[Address, Mem-
memory_Address_4G, boolean]
                ((pm' ro_addr ∪ pm' rw_addr))))
                (data(linear_resolve(a', Read)(s'))))
      ^
      OK?(expr_2_super(linear_resolve(a', Read))(s')) ∧
      OK?(memory_read(pm_phy' mem)(data(linear_resolve(a', Read)(s'))(s'))) ∧
      OK?(memory_read(pm_phy' mem)
                (data(linear_resolve(a', Read)(s'))
                (state(expr_2_super(linear_resolve(a', Read))(s')))))
      ⊃
      data(memory_read(pm_phy' mem)
            (data(linear_resolve(a', Read)(s'))
            (state(expr_2_super(linear_resolve(a', Read))(s')))))
      = data(memory_read(pm_phy' mem)(data(linear_resolve(a', Read)(s'))(s')))
{-6} union(pm' ro_addr, pm' rw_addr)(a') ∧ is_linear_plain_memory?(pm') ∧ pm' states(s')

```


C Proof scripts

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 3 subgoals:

linear_unchanged_invariant.1.1.1.1.1.1:

```

{-1} OK?(memory_read(pm_phy' mem)
      (data(linear_resolve(a', Read)(s'))(state(q'(s')))))
{-2} OK?(memory_read(pm_phy' mem)(data(linear_resolve(a', Read)(s'))(s')))
{-3} union(pm_phy' ro_addr, pm_phy' rw_addr)(data(linear_resolve(a', Read)(s')))
{-4} unchanged_memory_invariant?(pm_phy' mem, pm' states,
      singleton(expr_2_super[Physical_memory, Address]
      (linear_resolve(a', Read))),
      ((pm_phy' ro_addr ∪ pm_phy' rw_addr) \ extend [Address, Memory_Address
{-5} pm' states(state(q'(s')))
{-6} data(memory_read(pm_phy' mem)
      (data(linear_resolve(a', Read)(s'))
      (state(linear_resolve(a', Read)(state(q'(s'))))))))
      =
      data(memory_read(pm_phy' mem)
      (data(linear_resolve(a', Read)(s'))(state(q'(s')))))
{-7} pm' states(s')
{-8} data(memory_read(pm_phy' mem)
      (data(linear_resolve(a', Read)(s'))
      (state(linear_resolve(a', Read)(s')))))
      = data(memory_read(pm_phy' mem)(data(linear_resolve(a', Read)(s'))(s')))
{-9} union(pm' ro_addr, pm' rw_addr)(a')
{-10} is_linear_plain_memory?(pm')
{-11} OK?(linear_resolve(a', Read)(state(q'(s'))))
{-12} OK?(linear_resolve(a', Read)(s'))
{-13} data(linear_resolve(a', Read)(state(q'(s')))) =
      data(linear_resolve(a', Read)(s'))
{-14} transformer_invariant?(pm' states, transformers')
{-15} transformers'(q')
{-16} pm' states = pm_phy' states
{-17} (addresses' ⊆ restrict[Address, Memory_Address_4G, boolean]((pm' ro_addr ∪ pm' rw_addr)))
{-18} ∀ (s: (pm' states)):
      ∀ (s_1: Physical_memory, q: [Physical_memory → SuperResult[Physical_memory]],
      a: Address):
      pm_phy' states(s_1) ∧
      transformers'(q) ∧
      extend[Address, Memory_Address_4G, bool, FALSE]
      (virt_to_phys_range(s, addresses', (singleton(Read) ∪ singleton(Execute)))
      (a)
      ∧
      OK?(q(s_1)) ∧
      OK?(memory_read(pm_phy' mem)(a)(s_1)) ∧
      OK?(memory_read(pm_phy' mem)(a)(state(q(s_1))))
      ⊃
      data(memory_read(pm_phy' mem)(a)(state(q(s_1)))) =
      data(memory_read(pm_phy' mem)(a)(s_1))
{-19} extend[Address, Memory_Address_4G, bool, FALSE](addresses')(a')
{-20} OK?(q'(s'))
{-21} in_memory(min_linear, max_linear)(a')
{-22} Mem?(type_of(a'))
{-23} OK?(memory_read(pm_phy' mem)
      (data(linear_resolve(a', Read)(s'))
      (state(linear_resolve(a', Read)(s')))))
{-24} OK?(memory_read(pm_phy' mem)
      (data(linear_resolve(a', Read)(state(q'(s'))))
      (state(linear_resolve(a', Read)(state(q'(s'))))))))
-----
{1} data(memory_read(pm_phy' mem)
      (data(linear_resolve(a', Read)(state(q'(s'))))
      (state(linear_resolve(a', Read)(state(q'(s'))))))))
      =
      data(memory_read(pm_phy' mem)
      (data(linear_resolve(a', Read)(s'))
      (state(linear_resolve(a', Read)(s')))))

```

C Proof scripts

Replacing using formula -8,

Replacing using formula -13,

Instantiating the top quantifier in -18 with the terms: (s!1),

Instantiating quantified variables,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Hiding formulas: (-4 2),

Expanding the definition of extend,

Expanding the definition of virt_to_phys_range,

Using lemma linear_resolve_memory_address,

Expanding the definition of every,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Instantiating quantified variables,

Keeping (1) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `linear_unchanged_invariant.1.1.1.1.1.1`.

linear_unchanged_invariant.1.1.1.1.1.2:

```

{-1} OK?(memory_read(pm_phy' mem)
      (data(linear_resolve(a', Read)(s'))(state(q'(s'))))
{-2} OK?(memory_read(pm_phy' mem)(data(linear_resolve(a', Read)(s'))(s'))
{-3} union(pm_phy' ro_addr, pm_phy' rw_addr)(data(linear_resolve(a', Read)(s'))
{-4} unchanged_memory_invariant?(pm_phy' mem, pm' states,
      singleton(expr_2_super[Physical_memory, Address]
      (linear_resolve(a', Read))),
      ((pm_phy' ro_addr ∪ pm_phy' rw_addr) \ extend [Address, Memory_Address

{-5} pm' states(state(q'(s')))
{-6} pm' states(s')
{-7} union(pm' ro_addr, pm' rw_addr)(a')
{-8} is_linear_plain_memory?(pm')
{-9} OK?(linear_resolve(a', Read)(state(q'(s'))))
{-10} OK?(linear_resolve(a', Read)(s'))
{-11} data(linear_resolve(a', Read)(state(q'(s')))) =
      data(linear_resolve(a', Read)(s'))
{-12} transformer_invariant?(pm' states, transformers')
{-13} transformers'(q')
{-14} pm' states = pm_phy' states
{-15} (addresses' ⊆ restrict[Address, Memory_Address_4G, boolean]((pm' ro_addr ∪ pm' rw_addr)))
{-16} ∀ (s: (pm' states)):
      ∀ (s_1: Physical_memory, q: [Physical_memory → SuperResult[Physical_memory]],
      a: Address):
      pm_phy' states(s_1) ∧
      transformers'(q) ∧
      extend[Address, Memory_Address_4G, bool, FALSE]
      (virt_to_phys_range(s, addresses', (singleton(Read) ∪ singleton(Execute))))
      (a)
      ∧
      OK?(q(s_1)) ∧
      OK?(memory_read(pm_phy' mem)(a)(s_1)) ∧
      OK?(memory_read(pm_phy' mem)(a)(state(q(s_1))))
      ⊃
      data(memory_read(pm_phy' mem)(a)(state(q(s_1)))) =
      data(memory_read(pm_phy' mem)(a)(s_1))
{-17} extend[Address, Memory_Address_4G, bool, FALSE](addresses')(a')
{-18} OK?(q'(s'))
{-19} in_memory(min_linear, max_linear)(a')
{-20} Mem?(type_of(a'))
{-21} OK?(memory_read(pm_phy' mem)
      (data(linear_resolve(a', Read)(s'))
      (state(linear_resolve(a', Read)(s'))))
{-22} OK?(memory_read(pm_phy' mem)
      (data(linear_resolve(a', Read)(state(q'(s'))))
      (state(linear_resolve(a', Read)(state(q'(s'))))))
-----
{1} difference((pm_phy' ro_addr ∪ pm_phy' rw_addr),
      extend[Address, Memory_Address_4G, bool, FALSE]
      (address_in_pt_range?(s',
      restrict[Address, Mem-
      ory_Address_4G, boolean]
      ((pm' ro_addr ∪ pm' rw_addr))))
      (data(linear_resolve(a', Read)(s')))
{2} data(memory_read(pm_phy' mem)
      (data(linear_resolve(a', Read)(state(q'(s'))))
      (state(linear_resolve(a', Read)(state(q'(s'))))))
=
data(memory_read(pm_phy' mem)
      (data(linear_resolve(a', Read)(s'))
      (state(linear_resolve(a', Read)(s'))))

```

C Proof scripts

Hiding formulas: (-4 -16 2),

Expanding the definition of difference,

Expanding the definition of member,

Expanding the definition of extend,

Using lemma pm_linear_blessed,

Expanding the definition of linear_blessed?,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Hiding formulas: (-1 -2 -3 -4 -5 -7),

Expanding the definition of disjoint?,

Expanding the definition of empty?,

Expanding the definition of intersection,

Expanding the definition of member,

Instantiating the top quantifier in -1 with the terms: (data(linear_resolve(a!1, Read)(s!1))),

Case splitting on `restrict[Address, Memory_Address_4G, boolean] (union(pm!1'ro_addr, pm!1'rw_addr)) = union(restrict[Address, Memory_Address_4G, boolean](pm!1'ro_addr), restrict[Address, Memory_Address_4G, boolean](pm!1'rw_addr))`,

we get 2 subgoals:

linear_unchanged_invariant.1.1.1.1.1.2.1:

{-1}	restrict[Address, Memory_Address_4G, boolean]((pm'ro_addr ∪ pm'rw_addr)) = (restrict[Address, Memory_Address_4G, boolean](pm'ro_addr) ∪ restrict [Address, Memory_Address_4G,
{-2}	OK?(memory_read(pm_phy'mem) (data(linear_resolve(a', Read)(s'))(state(q'(s')))))
{-3}	OK?(memory_read(pm_phy'mem)(data(linear_resolve(a', Read)(s'))(s')))
{-4}	union(pm_phy'ro_addr, pm_phy'rw_addr)(data(linear_resolve(a', Read)(s')))
{-5}	pm'states(state(q'(s')))
{-6}	Mem?(data(linear_resolve(a', Read)(s'))'type_of)
{-7}	0 ≤ data(linear_resolve(a', Read)(s'))'offset
{-8}	data(linear_resolve(a', Read)(s'))'offset < max_linear_offset
{-9}	address_in_pt_range?(s', restrict[Address, Memory_Address_4G, boolean] ((pm'ro_addr ∪ pm'rw_addr)) (data(linear_resolve(a', Read)(s')))
{-10}	pm'states(s')
{-11}	union(pm'ro_addr, pm'rw_addr)(a')
{-12}	is_linear_plain_memory?(pm')
{-13}	OK?(linear_resolve(a', Read)(state(q'(s'))))
{-14}	OK?(linear_resolve(a', Read)(s'))
{-15}	data(linear_resolve(a', Read)(state(q'(s')))) = data(linear_resolve(a', Read)(s'))
{-16}	transformer_invariant?(pm'states, transformers')
{-17}	transformers'(q')
{-18}	pm'states = pm_phy'states
{-19}	(addresses' ⊆ restrict[Address, Memory_Address_4G, boolean]((pm'ro_addr ∪ pm'rw_addr)))
{-20}	0 ≤ a'offset
{-21}	a'offset < max_linear_offset
{-22}	addresses'(a')
{-23}	OK?(q'(s'))
{-24}	in_memory(min_linear, max_linear)(a')
{-25}	Mem?(type_of(a'))
{-26}	OK?(memory_read(pm_phy'mem) (data(linear_resolve(a', Read)(s')) (state(linear_resolve(a', Read)(s')))))
{-27}	OK?(memory_read(pm_phy'mem) (data(linear_resolve(a', Read)(state(q'(s')))) (state(linear_resolve(a', Read)(state(q'(s'))))))
{1}	virt_to_phys_range(s', (restrict[Address, Memory_Address_4G, boolean](pm'ro_addr) ∪ restrict [Address, (data(linear_resolve(a', Read)(s')) ∧ address_in_pt_range?(s', (restrict [Address, Memory_Address_4G, boolean](pm'ro_addr) ∪ restrict [Add (data(linear_resolve(a', Read)(s')))

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Expanding the definition of virt_to_phys_range,

Expanding the definition of virt_to_phys_range,

Instantiating quantified variables,

we get 2 subgoals:

linear_unchanged_invariant.1.1.1.1.1.2.1.1:

{-1}	restrict[Address, Memory_Address_4G, boolean]((pm'ro_addr ∪ pm'rw_addr) = (restrict[Address, Memory_Address_4G, boolean](pm'ro_addr) ∪ restrict [Address, Memory_A
{-2}	OK?(memory_read(pm_phy' mem) (data(linear_resolve(a', Read)(s'))(state(q'(s'))))
{-3}	OK?(memory_read(pm_phy' mem)(data(linear_resolve(a', Read)(s'))(s'))
{-4}	union(pm_phy'ro_addr, pm_phy'rw_addr)(data(linear_resolve(a', Read)(s'))
{-5}	pm' states(state(q'(s'))
{-6}	Mem?(data(linear_resolve(a', Read)(s'))'type_of)
{-7}	0 ≤ data(linear_resolve(a', Read)(s'))'offset
{-8}	data(linear_resolve(a', Read)(s'))'offset < max_linear_offset
{-9}	address_in_pt_range?(s', restrict[Address, Memory_Address_4G, boolean] (pm'ro_addr ∪ pm'rw_addr)) (data(linear_resolve(a', Read)(s'))
{-10}	pm' states(s')
{-11}	union(pm'ro_addr, pm'rw_addr)(a')
{-12}	is_linear_plain_memory?(pm')
{-13}	OK?(linear_resolve(a', Read)(state(q'(s'))))
{-14}	OK?(linear_resolve(a', Read)(s'))
{-15}	data(linear_resolve(a', Read)(state(q'(s')))) = data(linear_resolve(a', Read)(s'))
{-16}	transformer_invariant?(pm' states, transformers')
{-17}	transformers'(q')
{-18}	pm' states = pm_phy' states
{-19}	(addresses' ⊆ restrict[Address, Memory_Address_4G, boolean]((pm'ro_addr ∪ pm'rw_addr)))
{-20}	0 ≤ a' offset
{-21}	a' offset < max_linear_offset
{-22}	addresses'(a')
{-23}	OK?(q'(s'))
{-24}	in_memory(min_linear, max_linear)(a')
{-25}	Mem?(type_of(a'))
{-26}	OK?(memory_read(pm_phy' mem) (data(linear_resolve(a', Read)(s')) (state(linear_resolve(a', Read)(s'))))
{-27}	OK?(memory_read(pm_phy' mem) (data(linear_resolve(a', Read)(state(q'(s')))) (state(linear_resolve(a', Read)(state(q'(s'))))))
{1}	fullset[Memory_access](Read)

Expanding the definition of fullset,

which is trivially true.

This completes the proof of linear_unchanged_invariant.1.1.1.1.1.2.1.1.

linear_unchanged_invariant.1.1.1.1.1.2.1.2:

{-1}	restrict[Address, Memory_Address_4G, boolean]((pm'ro_addr ∪ pm'rw_addr) = (restrict[Address, Memory_Address_4G, boolean](pm'ro_addr) ∪ restrict [Address, Memory_Address_4G,
{-2}	OK?(memory_read(pm_phy'mem) (data(linear_resolve(a', Read)(s')))(state(q'(s'))))
{-3}	OK?(memory_read(pm_phy'mem)(data(linear_resolve(a', Read)(s')))(s'))
{-4}	union(pm_phy'ro_addr, pm_phy'rw_addr)(data(linear_resolve(a', Read)(s')))
{-5}	pm'states(state(q'(s')))
{-6}	Mem?(data(linear_resolve(a', Read)(s'))'type_of)
{-7}	0 ≤ data(linear_resolve(a', Read)(s'))'offset
{-8}	data(linear_resolve(a', Read)(s'))'offset < max_linear_offset
{-9}	address_in_pt_range?(s', restrict[Address, Memory_Address_4G, boolean] (pm'ro_addr ∪ pm'rw_addr)) (data(linear_resolve(a', Read)(s')))
{-10}	pm'states(s')
{-11}	union(pm'ro_addr, pm'rw_addr)(a')
{-12}	is_linear_plain_memory?(pm')
{-13}	OK?(linear_resolve(a', Read)(state(q'(s'))))
{-14}	OK?(linear_resolve(a', Read)(s'))
{-15}	data(linear_resolve(a', Read)(state(q'(s')))) = data(linear_resolve(a', Read)(s'))
{-16}	transformer_invariant?(pm'states, transformers')
{-17}	transformers'(q')
{-18}	pm'states = pm_phy'states
{-19}	(addresses' ⊆ restrict[Address, Memory_Address_4G, boolean]((pm'ro_addr ∪ pm'rw_addr)))
{-20}	0 ≤ a'offset
{-21}	a'offset < max_linear_offset
{-22}	addresses'(a')
{-23}	OK?(q'(s'))
{-24}	in_memory(min_linear, max_linear)(a')
{-25}	Mem?(type_of(a'))
{-26}	OK?(memory_read(pm_phy'mem) (data(linear_resolve(a', Read)(s')) (state(linear_resolve(a', Read)(s')))))
{-27}	OK?(memory_read(pm_phy'mem) (data(linear_resolve(a', Read)(state(q'(s')))) (state(linear_resolve(a', Read)(state(q'(s'))))))
{1}	union[Memory_Address_4G] (restrict[Address, Memory_Address_4G, boolean](pm'ro_addr), restrict[Address, Memory_Address_4G, boolean](pm'rw_addr)) (a')

Keeping (-11 1) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of linear_unchanged_invariant.1.1.1.1.1.2.1.2.

linear_unchanged_invariant.1.1.1.1.1.2.2:

{-1}	OK?(memory_read(pm_phy ' mem) (data(linear_resolve(a', Read)(s'))(state(q'(s')))))
{-2}	OK?(memory_read(pm_phy ' mem)(data(linear_resolve(a', Read)(s'))(s'))
{-3}	union(pm_phy ' ro_addr, pm_phy ' rw_addr)(data(linear_resolve(a', Read)(s')))
{-4}	pm' ' states(state(q'(s')))
{-5}	Mem?(data(linear_resolve(a', Read)(s')) ' type_of)
{-6}	0 ≤ data(linear_resolve(a', Read)(s')) ' offset
{-7}	data(linear_resolve(a', Read)(s')) ' offset < max_linear_offset
{-8}	address_in_pt_range?(s', restrict [Address, Memory_Address_4G, boolean] ((pm' ' ro_addr ∪ pm' ' rw_addr))) (data(linear_resolve(a', Read)(s')))
{-9}	pm' ' states(s')
{-10}	union(pm' ' ro_addr, pm' ' rw_addr)(a')
{-11}	is_linear_plain_memory?(pm')
{-12}	OK?(linear_resolve(a', Read)(state(q'(s'))))
{-13}	OK?(linear_resolve(a', Read)(s'))
{-14}	data(linear_resolve(a', Read)(state(q'(s')))) = data(linear_resolve(a', Read)(s'))
{-15}	transformer_invariant?(pm' ' states, transformers')
{-16}	transformers'(q')
{-17}	pm' ' states = pm_phy ' states
{-18}	(addresses' ⊆ restrict [Address, Memory_Address_4G, boolean]((pm' ' ro_addr ∪ pm' ' rw_addr)))
{-19}	0 ≤ a' ' offset
{-20}	a' ' offset < max_linear_offset
{-21}	addresses'(a')
{-22}	OK?(q'(s'))
{-23}	in_memory(min_linear, max_linear)(a')
{-24}	Mem?(type_of(a'))
{-25}	OK?(memory_read(pm_phy ' mem) (data(linear_resolve(a', Read)(s')) (state(linear_resolve(a', Read)(s'))))
{-26}	OK?(memory_read(pm_phy ' mem) (data(linear_resolve(a', Read)(state(q'(s')))) (state(linear_resolve(a', Read)(state(q'(s'))))))
{1}	restrict [Address, Memory_Address_4G, boolean]((pm' ' ro_addr ∪ pm' ' rw_addr)) = (restrict [Address, Memory_Address_4G, boolean](pm' ' ro_addr) ∪ restrict [Address, Memory_Address_4G, boolean](pm' ' rw_addr))
{2}	virt_to_phys_range(s', (restrict [Address, Memory_Address_4G, boolean](pm' ' ro_addr) ∪ restrict [Address, Memory_Address_4G, boolean](pm' ' rw_addr)) (data(linear_resolve(a', Read)(s')))) ∧ address_in_pt_range?(s', (restrict [Address, Memory_Address_4G, boolean](pm' ' ro_addr) ∪ restrict [Address, Memory_Address_4G, boolean](pm' ' rw_addr)) (data(linear_resolve(a', Read)(s'))))

Keeping (1) and hiding *,

Applying decompose-equality,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of linear_unchanged_invariant.1.1.1.1.1.2.2.

linear_unchanged_invariant.1.1.1.1.1.3:

```

{-1} OK?(memory_read(pm_phy' mem)
      (data(linear_resolve(a', Read)(s'))(state(q'(s')))))
{-2} OK?(memory_read(pm_phy' mem)(data(linear_resolve(a', Read)(s'))(s')))
{-3} union(pm_phy' ro_addr, pm_phy' rw_addr)(data(linear_resolve(a', Read)(s')))
{-4} unchanged_memory_invariant?(pm_phy' mem, pm' states,
      singleton(expr_2_super[Physical_memory, Address]
      (linear_resolve(a', Read))),
      ((pm_phy' ro_addr ∪ pm_phy' rw_addr) \ extend [Address, Memory_Address

{-5} pm' states(state(q'(s')))
{-6} pm' states(s')
{-7} union(pm' ro_addr, pm' rw_addr)(a')
{-8} is_linear_plain_memory?(pm')
{-9} OK?(linear_resolve(a', Read)(state(q'(s'))))
{-10} OK?(linear_resolve(a', Read)(s'))
{-11} data(linear_resolve(a', Read)(state(q'(s')))) =
      data(linear_resolve(a', Read)(s'))
{-12} transformer_invariant?(pm' states, transformers')
{-13} transformers'(q')
{-14} pm' states = pm_phy' states
{-15} (addresses' ⊆ restrict[Address, Memory_Address_4G, boolean]((pm' ro_addr ∪ pm' rw_addr)))
{-16} ∀ (s: (pm' states)):
      ∀ (s_1: Physical_memory, q: [Physical_memory → SuperResult[Physical_memory]],
      a: Address):
      pm_phy' states(s_1) ∧
      transformers'(q) ∧
      extend[Address, Memory_Address_4G, bool, FALSE]
      (virt_to_phys_range(s, addresses', (singleton(Read) ∪ singleton(Execute))))
      (a)
      ∧
      OK?(q(s_1)) ∧
      OK?(memory_read(pm_phy' mem)(a)(s_1)) ∧
      OK?(memory_read(pm_phy' mem)(a)(state(q(s_1))))
      ⊃
      data(memory_read(pm_phy' mem)(a)(state(q(s_1)))) =
      data(memory_read(pm_phy' mem)(a)(s_1))
{-17} extend[Address, Memory_Address_4G, bool, FALSE](addresses')(a')
{-18} OK?(q'(s'))
{-19} in_memory(min_linear, max_linear)(a')
{-20} Mem?(type_of(a'))
{-21} OK?(memory_read(pm_phy' mem)
      (data(linear_resolve(a', Read)(s'))
      (state(linear_resolve(a', Read)(s')))))
{-22} OK?(memory_read(pm_phy' mem)
      (data(linear_resolve(a', Read)(state(q'(s'))))
      (state(linear_resolve(a', Read)(state(q'(s'))))))



---


{1} expr_2_super(linear_resolve(a', Read)) =
      expr_2_super[Physical_memory, Address](linear_resolve(a', Read))
{2} data(memory_read(pm_phy' mem)
      (data(linear_resolve(a', Read)(state(q'(s'))))
      (state(linear_resolve(a', Read)(state(q'(s'))))))
      =
      data(memory_read(pm_phy' mem)
      (data(linear_resolve(a', Read)(s'))
      (state(linear_resolve(a', Read)(s')))))

```

C Proof scripts

Keeping (1) and hiding *,

Applying decompose-equality,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `linear_unchanged_invariant.1.1.1.1.1.3`.

linear_unchanged_invariant.1.1.1.1.2:

```

{-1} OK?(memory_read(pm_phy' mem)(data(linear_resolve(a', Read)(s')))(s'))
{-2} union(pm_phy' ro_addr, pm_phy' rw_addr)(data(linear_resolve(a', Read)(s')))
{-3} unchanged_memory_invariant?(pm_phy' mem, pm' states,
                                singleton(expr_2_super[Physical_memory, Address]
                                           (linear_resolve(a', Read))),
                                ((pm_phy' ro_addr ∪ pm_phy' rw_addr) \ extend [Address, Memory_Address
                                ^
                                pm' states(state(q'(s'))) ∧
                                singleton(expr_2_super[Physical_memory, Address](linear_resolve(a', Read)))
                                (expr_2_super(linear_resolve(a', Read)))
                                ^
                                difference((pm_phy' ro_addr ∪ pm_phy' rw_addr),
                                           extend[Address, Memory_Address_4G, bool, FALSE]
                                           (address_in_pt_range?(s',
                                           restrict[Address, Mem-
memory_Address_4G, boolean]
                                           ((pm' ro_addr ∪ pm' rw_addr))))
                                (data(linear_resolve(a', Read)(s')))
                                ^
                                OK?(expr_2_super(linear_resolve(a', Read))(state(q'(s')))) ∧
                                OK?(memory_read(pm_phy' mem)
                                (data(linear_resolve(a', Read)(s')))(state(q'(s'))))
                                ^
                                OK?(memory_read(pm_phy' mem)
                                (data(linear_resolve(a', Read)(s'))
                                (state(expr_2_super(linear_resolve(a', Read))(state(q'(s'))))))
                                ⊃
                                data(memory_read(pm_phy' mem)
                                (data(linear_resolve(a', Read)(s'))
                                (state(expr_2_super(linear_resolve(a', Read))(state(q'(s'))))))
                                =
                                data(memory_read(pm_phy' mem)
                                (data(linear_resolve(a', Read)(s')))(state(q'(s'))))
{-4} unchanged_memory_invariant?(pm_phy' mem, pm' states,
                                singleton(expr_2_super[Physical_memory, Address]
                                           (linear_resolve(a', Read))),
                                ((pm_phy' ro_addr ∪ pm_phy' rw_addr) \ extend [Address, Memory_Address
                                ^
                                pm' states(s') ∧
                                singleton(expr_2_super[Physical_memory, Address](linear_resolve(a', Read)))
                                (expr_2_super(linear_resolve(a', Read)))
                                ^
                                difference((pm_phy' ro_addr ∪ pm_phy' rw_addr),
                                           extend[Address, Memory_Address_4G, bool, FALSE]
                                           (address_in_pt_range?(s',
                                           restrict[Address, Mem-
memory_Address_4G, boolean]
                                           ((pm' ro_addr ∪ pm' rw_addr))))
                                (data(linear_resolve(a', Read)(s')))
                                ^
                                OK?(expr_2_super(linear_resolve(a', Read))(s')) ∧
                                OK?(memory_read(pm_phy' mem)(data(linear_resolve(a', Read)(s')))(s')) ∧
                                OK?(memory_read(pm_phy' mem)
                                (data(linear_resolve(a', Read)(s'))
                                (state(expr_2_super(linear_resolve(a', Read))(s'))))
                                ⊃
                                data(memory_read(pm_phy' mem)
                                (data(linear_resolve(a', Read)(s'))
                                (state(expr_2_super(linear_resolve(a', Read))(s'))))
                                = data(memory_read(pm_phy' mem)(data(linear_resolve(a', Read)(s')))(s'))
{-5} union(pm' ro_addr, pm' rw_addr)(a') ∧ is_linear_plain_memory?(pm') ∧ pm' states(s')
                                ⊃
                                unchanged_memory_invariant?[Physical_memory]

```

C Proof scripts

Hiding formulas: (-3 -4 -5 -13 2),

Using lemma plain_memory_transformers_ok_read_ro_rw,

Using lemma expr_transformers_ok_ok,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 2 subgoals:

`linear_unchanged_invariant.1.1.1.1.2.1:`

{-1}	transformers_ok?(pm_phy' states, memory_read_transformers(pm_phy' mem, (pm_phy' ro_addr ∪ pm_phy' rw_addr)))
{-2}	OK?(memory_read(pm_phy' mem)(data(linear_resolve(a', Read)(s')))(s'))
{-3}	union(pm_phy' ro_addr, pm_phy' rw_addr)(data(linear_resolve(a', Read)(s')))
{-4}	union(pm' ro_addr, pm' rw_addr)(a')
{-5}	is_linear_plain_memory?(pm')
{-6}	pm' states(state(q'(s')))
{-7}	pm' states(s')
{-8}	OK?(linear_resolve(a', Read)(state(q'(s'))))
{-9}	OK?(linear_resolve(a', Read)(s'))
{-10}	data(linear_resolve(a', Read)(state(q'(s')))) = data(linear_resolve(a', Read)(s'))
{-11}	transformer_invariant?(pm' states, transformers')
{-12}	transformers'(q')
{-13}	pm' states = pm_phy' states
{-14}	(addresses' ⊆ restrict[Address, Memory_Address_4G, boolean]((pm' ro_addr ∪ pm' rw_addr)))
{-15}	extend[Address, Memory_Address_4G, bool, FALSE](addresses')(a')
{-16}	OK?(q'(s'))
{-17}	in_memory(min_linear, max_linear)(a')
{-18}	Mem?(type_of(a'))
{-19}	OK?(memory_read(pm_phy' mem) (data(linear_resolve(a', Read)(s')) (state(linear_resolve(a', Read)(s'))))
{-20}	OK?(memory_read(pm_phy' mem) (data(linear_resolve(a', Read)(state(q'(s')))) (state(linear_resolve(a', Read)(state(q'(s')))))))
{1}	memory_read_transformers(pm_phy' mem, (pm_phy' ro_addr ∪ pm_phy' rw_addr)) (expr_2_super(memory_read(pm_phy' mem) (data(linear_resolve(a', Read)(s')))))
{2}	OK?(memory_read(pm_phy' mem) (data(linear_resolve(a', Read)(s')))(state(q'(s'))))

Using lemma memory_read_transformers_memory_read,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `linear_unchanged_invariant.1.1.1.1.2.1`.

linear_unchanged_invariant.1.1.1.1.2.2:

{-1}	OK?(memory_read(pm_phy'mem)(data(linear_resolve(a', Read)(s')))(s'))
{-2}	union(pm_phy'ro_addr, pm_phy'rw_addr)(data(linear_resolve(a', Read)(s')))
{-3}	union(pm'ro_addr, pm'rw_addr)(a')
{-4}	is_linear_plain_memory?(pm')
{-5}	pm'states(state(q'(s')))
{-6}	pm'states(s')
{-7}	OK?(linear_resolve(a', Read)(state(q'(s'))))
{-8}	OK?(linear_resolve(a', Read)(s'))
{-9}	data(linear_resolve(a', Read)(state(q'(s')))) = data(linear_resolve(a', Read)(s'))
{-10}	transformer_invariant?(pm'states, transformers')
{-11}	transformers'(q')
{-12}	pm'states = pm_phy'states
{-13}	(addresses' \subseteq restrict[Address, Memory_Address_4G, boolean]((pm'ro_addr \cup pm'rw_addr)))
{-14}	extend[Address, Memory_Address_4G, bool, FALSE](addresses')(a')
{-15}	OK?(q'(s'))
{-16}	in_memory(min_linear, max_linear)(a')
{-17}	Mem?(type_of(a'))
{-18}	OK?(memory_read(pm_phy'mem) (data(linear_resolve(a', Read)(s')) (state(linear_resolve(a', Read)(s'))))
{-19}	OK?(memory_read(pm_phy'mem) (data(linear_resolve(a', Read)(state(q'(s')))) (state(linear_resolve(a', Read)(state(q'(s'))))))
{1}	transformers_ok?(pm_phy'states, memory_read_transformers(pm_phy'mem, (pm_phy'ro_addr \cup pm_phy'rw_addr)))
{2}	OK?(memory_read(pm_phy'mem) (data(linear_resolve(a', Read)(s')))(state(q'(s'))))
{3}	plain_memory?(pm_phy)

Using lemma pm_plain_phy,

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of linear_unchanged_invariant.1.1.1.1.2.2.

linear_unchanged_invariant.1.1.1.1.3:

```

{-1} union(pm_phy'ro_addr, pm_phy'rw_addr)(data(linear_resolve(a', Read)(s')))
{-2} unchanged_memory_invariant?(pm_phy'mem, pm'states,
                                singleton(expr_2_super[Physical_memory, Address]
                                           (linear_resolve(a', Read))),
                                ((pm_phy'ro_addr ∪ pm_phy'rw_addr) \ extend [Address, Mem-
                                ^
                                pm'states(state(q'(s'))) ∧
                                singleton(expr_2_super[Physical_memory, Address](linear_resolve(a', Read)))
                                (expr_2_super(linear_resolve(a', Read)))
                                ^
                                difference((pm_phy'ro_addr ∪ pm_phy'rw_addr),
                                           extend[Address, Memory_Address_4G, bool, FALSE]
                                           (address_in_pt_range?(s',
                                restrict[Address, Mem-
                                ory_Address_4G, boolean]
                                ((pm'ro_addr ∪ pm'rw_addr))))
                                (data(linear_resolve(a', Read)(s')))
                                ^
                                OK?(expr_2_super(linear_resolve(a', Read))(state(q'(s')))) ∧
                                OK?(memory_read(pm_phy'mem)
                                (data(linear_resolve(a', Read)(s')))(state(q'(s'))))
                                ^
                                OK?(memory_read(pm_phy'mem)
                                (data(linear_resolve(a', Read)(s')))
                                (state(expr_2_super(linear_resolve(a', Read))(state(q'(s'))))))
                                ⊃
                                data(memory_read(pm_phy'mem)
                                (data(linear_resolve(a', Read)(s')))
                                (state(expr_2_super(linear_resolve(a', Read))(state(q'(s'))))))
                                =
                                data(memory_read(pm_phy'mem)
                                (data(linear_resolve(a', Read)(s')))(state(q'(s'))))
{-3} unchanged_memory_invariant?(pm_phy'mem, pm'states,
                                singleton(expr_2_super[Physical_memory, Address]
                                           (linear_resolve(a', Read))),
                                ((pm_phy'ro_addr ∪ pm_phy'rw_addr) \ extend [Address, Mem-
                                ^
                                pm'states(s') ∧
                                singleton(expr_2_super[Physical_memory, Address](linear_resolve(a', Read)))
                                (expr_2_super(linear_resolve(a', Read)))
                                ^
                                difference((pm_phy'ro_addr ∪ pm_phy'rw_addr),
                                           extend[Address, Memory_Address_4G, bool, FALSE]
                                           (address_in_pt_range?(s',
                                restrict[Address, Mem-
                                ory_Address_4G, boolean]
                                ((pm'ro_addr ∪ pm'rw_addr))))
                                (data(linear_resolve(a', Read)(s')))
                                ^
                                OK?(expr_2_super(linear_resolve(a', Read))(s')) ∧
                                OK?(memory_read(pm_phy'mem)(data(linear_resolve(a', Read)(s')))(s')) ∧
                                OK?(memory_read(pm_phy'mem)
                                (data(linear_resolve(a', Read)(s')))
                                (state(expr_2_super(linear_resolve(a', Read))(s'))))
                                ⊃
                                data(memory_read(pm_phy'mem)
                                (data(linear_resolve(a', Read)(s')))
                                (state(expr_2_super(linear_resolve(a', Read))(s'))))
                                = data(memory_read(pm_phy'mem)(data(linear_resolve(a', Read)(s')))(s'))
{-4} union(pm'ro_addr, pm'rw_addr)(a') ∧ is_linear_plain_memory?(pm') ∧ pm'states(s')
                                ⊃
                                unchanged_memory_invariant?[Physical_memory]
                                (pm_phy'mem, pm'states,

```

Hiding formulas: (-2 -3 -4 -12 2),

Using lemma plain_memory_transformers_ok_read_ro_rw,

Using lemma expr_transformers_ok_ok,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 2 subgoals:

linear_unchanged_invariant.1.1.1.1.3.1:

{-1}	transformers_ok?(pm_phy' states,
	memory_read_transformers(pm_phy' mem, (pm_phy' ro_addr ∪ pm_phy' rw_addr)))
{-2}	union(pm_phy' ro_addr, pm_phy' rw_addr)(data(linear_resolve(a', Read)(s')))
{-3}	union(pm' ro_addr, pm' rw_addr)(a')
{-4}	is_linear_plain_memory?(pm')
{-5}	pm' states(state(q'(s')))
{-6}	pm' states(s')
{-7}	OK?(linear_resolve(a', Read)(state(q'(s'))))
{-8}	OK?(linear_resolve(a', Read)(s'))
{-9}	data(linear_resolve(a', Read)(state(q'(s')))) = data(linear_resolve(a', Read)(s'))
{-10}	transformer_invariant?(pm' states, transformers')
{-11}	transformers'(q')
{-12}	pm' states = pm_phy' states
{-13}	(addresses' ⊆ restrict[Address, Memory_Address_4G, boolean]((pm' ro_addr ∪ pm' rw_addr)))
{-14}	extend[Address, Memory_Address_4G, bool, FALSE](addresses')(a')
{-15}	OK?(q'(s'))
{-16}	in_memory(min_linear, max_linear)(a')
{-17}	Mem?(type_of(a'))
{-18}	OK?(memory_read(pm_phy' mem) (data(linear_resolve(a', Read)(s')) (state(linear_resolve(a', Read)(s')))))
{-19}	OK?(memory_read(pm_phy' mem) (data(linear_resolve(a', Read)(state(q'(s')))) (state(linear_resolve(a', Read)(state(q'(s'))))))
{1}	memory_read_transformers(pm_phy' mem, (pm_phy' ro_addr ∪ pm_phy' rw_addr)) (expr_2_super(memory_read(pm_phy' mem) (data(linear_resolve(a', Read)(s')))))
{2}	OK?(memory_read(pm_phy' mem)(data(linear_resolve(a', Read)(s')))(s'))

Using lemma memory_read_transformers_memory_read,

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of linear_unchanged_invariant.1.1.1.1.3.1.

linear_unchanged_invariant.1.1.1.1.3.2:

{-1}	union(pm_phy'ro_addr, pm_phy'rw_addr)(data(linear_resolve(a', Read)(s')))
{-2}	union(pm'ro_addr, pm'rw_addr)(a')
{-3}	is_linear_plain_memory?(pm')
{-4}	pm' states(state(q'(s')))
{-5}	pm' states(s')
{-6}	OK?(linear_resolve(a', Read)(state(q'(s'))))
{-7}	OK?(linear_resolve(a', Read)(s'))
{-8}	data(linear_resolve(a', Read)(state(q'(s')))) = data(linear_resolve(a', Read)(s'))
{-9}	transformer_invariant?(pm' states, transformers')
{-10}	transformers'(q')
{-11}	pm' states = pm_phy' states
{-12}	(addresses' \subseteq restrict[Address, Memory_Address_4G, boolean]((pm'ro_addr \cup pm'rw_addr)))
{-13}	extend[Address, Memory_Address_4G, bool, FALSE](addresses')(a')
{-14}	OK?(q'(s'))
{-15}	in_memory(min_linear, max_linear)(a')
{-16}	Mem?(type_of(a'))
{-17}	OK?(memory_read(pm_phy' mem) (data(linear_resolve(a', Read)(s')) (state(linear_resolve(a', Read)(s'))))
{-18}	OK?(memory_read(pm_phy' mem) (data(linear_resolve(a', Read)(state(q'(s')))) (state(linear_resolve(a', Read)(state(q'(s')))))))
{1}	transformers_ok?(pm_phy' states, memory_read_transformers(pm_phy' mem, (pm_phy'ro_addr \cup pm_phy'rw_a
{2}	OK?(memory_read(pm_phy' mem)(data(linear_resolve(a', Read)(s')))(s'))
{3}	plain_memory?(pm_phy)

Using lemma pm_plain_phy,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of linear_unchanged_invariant.1.1.1.1.3.2.

linear_unchanged_invariant.1.1.1.2:

```

{-1}  unchanged_memory_invariant?(pm_phy' mem, pm' states,
                                         singleton(expr_2_super[Physical_memory, Address]
                                                         (linear_resolve(a', Read))),
                                         ((pm_phy'ro_addr ∪ pm_phy'rw_addr) \ extend [Address, Memory_Address_4G, boolean]
                                                         (address_in_pt_range?(s',
                                                         restrict[Address, Mem-
ory_Address_4G, boolean]
                                                         ((pm'ro_addr ∪ pm'rw_addr))))))
    ^
    pm' states(state(q'(s'))) ^
    singleton(expr_2_super[Physical_memory, Address](linear_resolve(a', Read))
              (expr_2_super(linear_resolve(a', Read)))
    ^
    difference((pm_phy'ro_addr ∪ pm_phy'rw_addr),
              extend[Address, Memory_Address_4G, bool, FALSE]
              (address_in_pt_range?(s',
              restrict[Address, Mem-
ory_Address_4G, boolean]
              ((pm'ro_addr ∪ pm'rw_addr))))))
    ^
    OK?(expr_2_super(linear_resolve(a', Read))(state(q'(s')))) ^
    OK?(memory_read(pm_phy' mem)
          (data(linear_resolve(a', Read)(s')))(state(q'(s'))))
    ^
    OK?(memory_read(pm_phy' mem)
          (data(linear_resolve(a', Read)(s'))
           (state(expr_2_super(linear_resolve(a', Read))(state(q'(s'))))))))
    ⊃
    data(memory_read(pm_phy' mem)
          (data(linear_resolve(a', Read)(s'))
           (state(expr_2_super(linear_resolve(a', Read))(state(q'(s'))))))))
    =
    data(memory_read(pm_phy' mem)
          (data(linear_resolve(a', Read)(s')))(state(q'(s'))))
{-2}  unchanged_memory_invariant?(pm_phy' mem, pm' states,
                                         singleton(expr_2_super[Physical_memory, Address]
                                                         (linear_resolve(a', Read))),
                                         ((pm_phy'ro_addr ∪ pm_phy'rw_addr) \ extend [Address, Memory_Address_4G, boolean]
                                                         (address_in_pt_range?(s',
                                                         restrict[Address, Mem-
ory_Address_4G, boolean]
                                                         ((pm'ro_addr ∪ pm'rw_addr))))))
    ^
    pm' states(s') ^
    singleton(expr_2_super[Physical_memory, Address](linear_resolve(a', Read))
              (expr_2_super(linear_resolve(a', Read)))
    ^
    difference((pm_phy'ro_addr ∪ pm_phy'rw_addr),
              extend[Address, Memory_Address_4G, bool, FALSE]
              (address_in_pt_range?(s',
              restrict[Address, Mem-
ory_Address_4G, boolean]
              ((pm'ro_addr ∪ pm'rw_addr))))))
    ^
    OK?(expr_2_super(linear_resolve(a', Read))(s')) ^
    OK?(memory_read(pm_phy' mem)(data(linear_resolve(a', Read)(s')))(s')) ^
    OK?(memory_read(pm_phy' mem)
          (data(linear_resolve(a', Read)(s'))
           (state(expr_2_super(linear_resolve(a', Read))(s'))))
    ⊃
    data(memory_read(pm_phy' mem)
          (data(linear_resolve(a', Read)(s'))
           (state(expr_2_super(linear_resolve(a', Read))(s'))))
    = data(memory_read(pm_phy' mem)(data(linear_resolve(a', Read)(s')))(s'))
{-3}  union(pm'ro_addr, pm'rw_addr)(a') ^ is_linear_plain_memory?(pm') ^ pm' states(s')
    ⊃
    unchanged_memory_invariant?[Physical_memory]
      (pm_phy' mem, pm' states,
       singleton(expr_2_super[Physical_memory, Address](linear_resolve(a', Read))),

```

C Proof scripts

Hiding formulas: (-1 -2 -3 -11 2),

Using lemma pm_linear_blessed,

Expanding the definition of linear_blessed?,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Hiding formulas: (-1 -2 -4 -5 -6 -7),

Expanding the definition of subset?,

Instantiating the top quantifier in -1 with the terms: (data(linear_resolve(a!1, Read)(s!1))),

we get 2 subgoals:

linear_unchanged_invariant.1.1.1.2.1:

{-1}	(data(linear_resolve(a', Read)(s')) ∈ virt_to_phys_range(s', (restrict [Address, Memory_Address_4G, boolean]((pm_phy'ro_addr ∪ pm_phy'rw_addr)(data(linear_resolve(a', Read)(s')))))) ⇒ (data(linear_resolve(a', Read)(s')) ∈ restrict [Address, Memory_Address_4G, boolean]((pm_phy'ro_addr ∪ pm_phy'rw_addr)(data(linear_resolve(a', Read)(s'))))))
{-2}	union(pm_phy'ro_addr, pm_phy'rw_addr)(a')
{-3}	is_linear_plain_memory?(pm')
{-4}	pm' states(state(q'(s')))
{-5}	pm' states(s')
{-6}	OK?(linear_resolve(a', Read)(state(q'(s'))))
{-7}	OK?(linear_resolve(a', Read)(s'))
{-8}	data(linear_resolve(a', Read)(state(q'(s')))) = data(linear_resolve(a', Read)(s'))
{-9}	transformer_invariant?(pm' states, transformers')
{-10}	transformers'(q')
{-11}	pm' states = pm_phy' states
{-12}	∀ (x: Memory_Address_4G): (x ∈ addresses') ⇒ (x ∈ restrict[Address, Memory_Address_4G, boolean]((pm_phy'ro_addr ∪ pm_phy'rw_addr)(data(linear_resolve(a', Read)(s'))))))
{-13}	extend[Address, Memory_Address_4G, bool, FALSE](addresses')(a')
{-14}	OK?(q'(s'))
{-15}	in_memory(min_linear, max_linear)(a')
{-16}	Mem?(type_of(a'))
{-17}	OK?(memory_read(pm_phy' mem) (data(linear_resolve(a', Read)(s')) (state(linear_resolve(a', Read)(s'))))
{-18}	OK?(memory_read(pm_phy' mem) (data(linear_resolve(a', Read)(state(q'(s')))) (state(linear_resolve(a', Read)(state(q'(s'))))))
{1}	union(pm_phy'ro_addr, pm_phy'rw_addr)(data(linear_resolve(a', Read)(s')))

Expanding the definition of member,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 2 subgoals:

linear_unchanged_invariant.1.1.1.2.1.1:

{-1}	restrict[Address, Memory_Address_4G, boolean] ((pm_phy'ro_addr ∪ pm_phy'rw_addr)(data(linear_resolve(a', Read)(s'))))
{-2}	union(pm_phy'ro_addr, pm_phy'rw_addr)(a')
{-3}	is_linear_plain_memory?(pm')
{-4}	pm' 'states(state(q'(s')))
{-5}	pm' 'states(s')
{-6}	OK?(linear_resolve(a', Read)(state(q'(s'))))
{-7}	OK?(linear_resolve(a', Read)(s'))
{-8}	data(linear_resolve(a', Read)(state(q'(s')))) = data(linear_resolve(a', Read)(s'))
{-9}	transformer_invariant?(pm' 'states, transformers')
{-10}	transformers'(q')
{-11}	pm' 'states = pm_phy 'states
{-12}	∀ (x: Memory_Address_4G): addresses'(x) ⇒ restrict[Address, Memory_Address_4G, boolean]((pm' 'ro_addr ∪ pm' 'rw_addr)(x))
{-13}	extend[Address, Memory_Address_4G, bool, FALSE](addresses')(a')
{-14}	OK?(q'(s'))
{-15}	in_memory(min_linear, max_linear)(a')
{-16}	Mem?(type_of(a'))
{-17}	OK?(memory_read(pm_phy 'mem) (data(linear_resolve(a', Read)(s')) (state(linear_resolve(a', Read)(s'))))
{-18}	OK?(memory_read(pm_phy 'mem) (data(linear_resolve(a', Read)(state(q'(s')))) (state(linear_resolve(a', Read)(state(q'(s')))))))
{1}	union(pm_phy'ro_addr, pm_phy'rw_addr)(data(linear_resolve(a', Read)(s')))

Expanding the definition of restrict,

which is trivially true.

This completes the proof of linear_unchanged_invariant.1.1.1.2.1.1.

linear_unchanged_invariant.1.1.1.2.1.2:

{-1}	union(pm'ro_addr, pm'rw_addr)(a')
{-2}	is_linear_plain_memory?(pm')
{-3}	pm'states(state(q'(s')))
{-4}	pm'states(s')
{-5}	OK?(linear_resolve(a', Read)(state(q'(s'))))
{-6}	OK?(linear_resolve(a', Read)(s'))
{-7}	data(linear_resolve(a', Read)(state(q'(s')))) = data(linear_resolve(a', Read)(s'))
{-8}	transformer_invariant?(pm'states, transformers')
{-9}	transformers'(q')
{-10}	pm'states = pm_phy'states
{-11}	∃ (x: Memory_Address_4G): addresses'(x) ⇒ restrict[Address, Memory_Address_4G, boolean]((pm'ro_addr ∪ pm'rw_addr))(x)
{-12}	extend[Address, Memory_Address_4G, bool, FALSE](addresses')(a')
{-13}	OK?(q'(s'))
{-14}	in_memory(min_linear, max_linear)(a')
{-15}	Mem?(type_of(a'))
{-16}	OK?(memory_read(pm_phy'mem) (data(linear_resolve(a', Read)(s')) (state(linear_resolve(a', Read)(s'))))
{-17}	OK?(memory_read(pm_phy'mem) (data(linear_resolve(a', Read)(state(q'(s')))) (state(linear_resolve(a', Read)(state(q'(s'))))))
{1}	virt_to_phys_range(s', (restrict[Address, Memory_Address_4G, boolean](pm'ro_addr) ∪ restrict (singleton(Read) ∪ singleton(Execute))) (data(linear_resolve(a', Read)(s'))
{2}	union(pm_phy'ro_addr, pm_phy'rw_addr)(data(linear_resolve(a', Read)(s')))

Hiding formulas: 2,

Expanding the definition of virt_to_phys_range,

Instantiating quantified variables,

we get 2 subgoals:

linear_unchanged_invariant.1.1.1.2.1.2.1:

{-1}	union(pm'ro_addr, pm'rw_addr)(a')
{-2}	is_linear_plain_memory?(pm')
{-3}	pm'states(state(q'(s')))
{-4}	pm'states(s')
{-5}	OK?(linear_resolve(a', Read)(state(q'(s'))))
{-6}	OK?(linear_resolve(a', Read)(s'))
{-7}	data(linear_resolve(a', Read)(state(q'(s')))) = data(linear_resolve(a', Read)(s'))
{-8}	transformer_invariant?(pm'states, transformers')
{-9}	transformers'(q')
{-10}	pm'states = pm_phy'states
{-11}	∀ (x: Memory_Address_4G): addresses'(x) ⇒ restrict[Address, Memory_Address_4G, boolean]((pm'ro_addr ∪ pm'rw_addr)(x)
{-12}	extend[Address, Memory_Address_4G, bool, FALSE](addresses')(a')
{-13}	OK?(q'(s'))
{-14}	in_memory(min_linear, max_linear)(a')
{-15}	Mem?(type_of(a'))
{-16}	OK?(memory_read(pm_phy'mem) (data(linear_resolve(a', Read)(s')) (state(linear_resolve(a', Read)(s'))))
{-17}	OK?(memory_read(pm_phy'mem) (data(linear_resolve(a', Read)(state(q'(s')))) (state(linear_resolve(a', Read)(state(q'(s')))))))
<hr/>	
{1}	union[Memory_access] (singleton[Memory_access](Read), singleton[Memory_access](Execute))(Read)

Keeping (1) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of linear_unchanged_invariant.1.1.1.2.1.2.1.

C Proof scripts

linear_unchanged_invariant.1.1.1.2.1.2.2:

{-1}	union(pm'ro_addr, pm'rw_addr)(a')
{-2}	is_linear_plain_memory?(pm')
{-3}	pm'states(state(q'(s')))
{-4}	pm'states(s')
{-5}	OK?(linear_resolve(a', Read)(state(q'(s'))))
{-6}	OK?(linear_resolve(a', Read)(s'))
{-7}	data(linear_resolve(a', Read)(state(q'(s')))) = data(linear_resolve(a', Read)(s'))
{-8}	transformer_invariant?(pm'states, transformers')
{-9}	transformers'(q')
{-10}	pm'states = pm_phy'states
{-11}	$\forall (x: \text{Memory_Address_4G}):$ addresses'(x) \Rightarrow restrict[Address, Memory_Address_4G, boolean]((pm'ro_addr \cup pm'rw_addr)(x))
{-12}	extend[Address, Memory_Address_4G, bool, FALSE](addresses')(a')
{-13}	OK?(q'(s'))
{-14}	in_memory(min_linear, max_linear)(a')
{-15}	Mem?(type_of(a'))
{-16}	OK?(memory_read(pm_phy'mem) (data(linear_resolve(a', Read)(s')) (state(linear_resolve(a', Read)(s'))))
{-17}	OK?(memory_read(pm_phy'mem) (data(linear_resolve(a', Read)(state(q'(s')))) (state(linear_resolve(a', Read)(state(q'(s')))))))
{1}	union[Memory_Address_4G] (restrict[Address, Memory_Address_4G, boolean](pm'ro_addr), restrict[Address, Memory_Address_4G, boolean](pm'rw_addr)) (a')

Keeping (-1 1) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of linear_unchanged_invariant.1.1.1.2.1.2.2.

linear_unchanged_invariant.1.1.1.2.2:

{-1}	union(pm'ro_addr, pm'rw_addr)(a')
{-2}	is_linear_plain_memory?(pm')
{-3}	pm'states(state(q'(s')))
{-4}	pm'states(s')
{-5}	OK?(linear_resolve(a', Read)(state(q'(s'))))
{-6}	OK?(linear_resolve(a', Read)(s'))
{-7}	data(linear_resolve(a', Read)(state(q'(s')))) = data(linear_resolve(a', Read)(s'))
{-8}	transformer_invariant?(pm'states, transformers')
{-9}	transformers'(q')
{-10}	pm'states = pm_phy'states
{-11}	$\forall (x: \text{Memory_Address_4G}):$ $(x \in \text{addresses}') \Rightarrow$ $(x \in \text{restrict}[\text{Address}, \text{Memory_Address_4G}, \text{boolean}]((\text{pm}'\text{ro_addr} \cup \text{pm}'\text{rw_addr})))$
{-12}	extend[Address, Memory_Address_4G, bool, FALSE](addresses')(a')
{-13}	OK?(q'(s'))
{-14}	in_memory(min_linear, max_linear)(a')
{-15}	Mem?(type_of(a'))
{-16}	OK?(memory_read(pm_phy'mem) (data(linear_resolve(a', Read)(s')) (state(linear_resolve(a', Read)(s')))))
{-17}	OK?(memory_read(pm_phy'mem) (data(linear_resolve(a', Read)(state(q'(s')))) (state(linear_resolve(a', Read)(state(q'(s')))))))
{1} Mem?(data[Physical_memory, Address] (linear_resolve[Physical_memory, pm_phy](a', Read)(s'))'type_of)	
	^
	0 ≤
	data[Physical_memory, Address] (linear_resolve[Physical_memory, pm_phy](a', Read)(s'))'offset
	^
	data[Physical_memory, Address] (linear_resolve[Physical_memory, pm_phy](a', Read)(s'))'offset
	< max_linear_offset
{2}	union(pm_phy'ro_addr, pm_phy'rw_addr)(data(linear_resolve(a', Read)(s')))

Using lemma linear_resolve_memory_address,

Expanding the definition of every,

which is trivially true.

This completes the proof of linear_unchanged_invariant.1.1.1.2.2.

linear_unchanged_invariant.1.1.2:

<pre> {-1} union(pm'ro_addr, pm'rw_addr)(a') {-2} transformer_invariant?(pm'states, transformers') {-3} is_linear_plain_memory?(pm') {-4} pm'states = pm_phy'states {-5} (addresses' ⊆ restrict[Address, Memory_Address_4G, boolean]((pm'ro_addr ∪ pm'rw_addr))) {-6} ∀ (s: (pm'states)): ∀ (s_1: Physical_memory, q: [Physical_memory → SuperResult[Physical_memory]], a: Address): pm_phy'states(s_1) ∧ transformers'(q) ∧ extend[Address, Memory_Address_4G, bool, FALSE] (virt_to_phys_range(s, addresses', (singleton(Read) ∪ singleton(Execute)))) (a) ∧ OK?(q(s_1)) ∧ OK?(memory_read(pm_phy'mem)(a)(s_1)) ∧ OK?(memory_read(pm_phy'mem)(a)(state(q(s_1)))) ⊃ data(memory_read(pm_phy'mem)(a)(state(q(s_1)))) = data(memory_read(pm_phy'mem)(a)(s_1)) {-7} pm'states(s') {-8} transformers'(q') {-9} extend[Address, Memory_Address_4G, bool, FALSE](addresses')(a') {-10} OK?(q'(s')) {-11} in_memory(min_linear, max_linear)(a') {-12} OK?(memory_read(pm_phy'mem)(a')(s')) {-13} OK?(memory_read(pm_phy'mem)(a')(state(q'(s')))) </pre>	<pre> {1} Mem?(type_of(a')) {2} data(memory_read(pm_phy'mem)(a')(state(q'(s')))) = data(memory_read(pm_phy'mem)(a')(s')) </pre>
---	---

Using lemma pm_memory_addr,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of linear_unchanged_invariant.1.1.2.

linear_unchanged_invariant.1.2:

```

{-1} transformer_invariant?(pm' states, transformers')
{-2} is_linear_plain_memory?(pm')
{-3} pm' states = pm_phy states
{-4} (addresses' ⊆ restrict[Address, Memory_Address_4G, boolean]((pm' ro_addr ∪ pm' rw_addr)))
{-5} ∀ (s: (pm' states)):
    ∀ (s_1: Physical_memory, q: [Physical_memory → SuperResult[Physical_memory]],
        a: Address):
        pm_phy states(s_1) ∧
        transformers'(q) ∧
        extend[Address, Memory_Address_4G, bool, FALSE]
            (virt_to_phys_range(s, addresses', (singleton(Read) ∪ singleton(Execute)))
            (a)
            ∧
            OK?(q(s_1)) ∧
            OK?(memory_read(pm_phy mem)(a)(s_1)) ∧
            OK?(memory_read(pm_phy mem)(a)(state(q(s_1))))
        ⊃
        data(memory_read(pm_phy mem)(a)(state(q(s_1)))) =
        data(memory_read(pm_phy mem)(a)(s_1))
{-6} pm' states(s')
{-7} transformers'(q')
{-8} extend[Address, Memory_Address_4G, bool, FALSE](addresses')(a')
{-9} OK?(q'(s'))
{-10} OK?(IF in_memory(min_linear, max_linear)(a')
    THEN IF Mem?(type_of(a'))
        THEN (linear_resolve(a', Read) ##
            (λ (pa: Address): memory_read(pm_phy mem)(pa)))
            (s')
        ELSE memory_read(pm_phy mem)(a')(s')
    ENDIF
    ELSE fatal_result(s')
    ENDIF)
{-11} OK?(IF in_memory(min_linear, max_linear)(a')
    THEN IF Mem?(type_of(a'))
        THEN (linear_resolve(a', Read) ##
            (λ (pa: Address): memory_read(pm_phy mem)(pa)))
            (state(q'(s')))
        ELSE memory_read(pm_phy mem)(a')(state(q'(s')))
    ENDIF
    ELSE fatal_result(state(q'(s')))
    ENDIF)

```

```

{1} union(pm' ro_addr, pm' rw_addr)(a')
{2} IF in_memory(min_linear, max_linear)(a')
    THEN IF Mem?(type_of(a'))
        THEN data((linear_resolve(a', Read) ##
            (λ (pa: Address): memory_read(pm_phy mem)(pa)))
            (state(q'(s'))))
        ELSE data(memory_read(pm_phy mem)(a')(state(q'(s'))))
    ENDIF
    ELSE data(fatal_result(state(q'(s'))))
    ENDIF
=
IF in_memory(min_linear, max_linear)(a')
    THEN IF Mem?(type_of(a'))
        THEN data((linear_resolve(a', Read) ##
            (λ (pa: Address): memory_read(pm_phy mem)(pa)))
            (s'))
        ELSE data(memory_read(pm_phy mem)(a')(s'))
    ENDIF
    ELSE data(fatal_result(s'))
    ENDIF

```

C Proof scripts

Keeping (-4 -8 1) and hiding *,
 Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of `linear_unchanged_invariant.1.2`.
`linear_unchanged_invariant.2`:

```

{-1} is_linear_plain_memory?(pm') ⊃ pm' states = pm_phy states
{-2} is_linear_plain_memory?(pm')
{-3} (addresses' ⊆ restrict[Address, Memory_Address_4G, boolean]((pm' ro_addr ∪ pm' rw_addr)))
{-4} ∀ (s: (pm' states)):
      transformer_invariant?(pm_phy states, transformers') ∧
      (∀ (s_1: Physical_memory, q: [Physical_memory → SuperResult[Physical_memory]],
        a: Address):
        pm_phy states(s_1) ∧
        transformers'(q) ∧
        extend[Address, Memory_Address_4G, bool, FALSE]
          (virt_to_phys_range(s, addresses',
            (singleton(Read) ∪ singleton(Execute))))
          (a)
          ∧
          OK?(q(s_1)) ∧
          OK?(memory_read(pm_phy mem)(a)(s_1)) ∧
          OK?(memory_read(pm_phy mem)(a)(state(q(s_1))))
        ⊃
        data(memory_read(pm_phy mem)(a)(state(q(s_1)))) =
        data(memory_read(pm_phy mem)(a)(s_1)))
      )


---


{1} transformer_invariant?(pm' states, transformers')
{2} transformer_invariant?(pm' states, transformers') ∧
      (∀ (s: Linear_memory[Physical_memory, pm_phy],
        q: [Linear_memory[Physical_memory, pm_phy] → SuperResult[Physical_memory]],
        a: Address):
        pm' states(s) ∧
        transformers'(q) ∧
        extend[Address, Memory_Address_4G, bool, FALSE](addresses')(a) ∧
        OK?(q(s)) ∧
        OK?(memory_read(pm' mem)(a)(s)) ∧
        OK?(memory_read(pm' mem)(a)(state(q(s))))
        ⊃
        data(memory_read(pm' mem)(a)(state(q(s)))) =
        data(memory_read(pm' mem)(a)(s)))
      )

```

Hiding formulas: 2,
 Expanding the definition of `transformer_invariant?`,
 Repeatedly Skolemizing and flattening,
 Instantiating quantified variables,
 Applying disjunctive simplification to flatten sequent,
 Instantiating quantified variables,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 This completes the proof of `linear_unchanged_invariant.2`.
 Q.E.D.

C.117.85**Linear_Memory_Properties.pm_phy_read_after_resolve_ok_TCC1**

Terse proof for pm_phy_read_after_resolve_ok_TCC1.

pm_phy_read_after_resolve_ok_TCC1:

$$\{1\} \quad \forall (s: \text{Linear_memory}[\text{Physical_memory}, \text{pm_phy}], a_2: \text{Memory_Address_4G}, ac: \text{Memory_access}):$$

$$\text{plain_memory?}(\text{pm_phy}) \wedge \text{OK?}(\text{linear_resolve}(a_2, ac)(s)) \supset$$

$$\text{OK?}[\text{Physical_memory}, \text{Address}](\text{linear_resolve}[\text{Physical_memory}, \text{pm_phy}](a_2, ac)(s)) \vee$$

$$\text{Exception?}[\text{Physical_memory}, \text{Address}]$$

$$(\text{linear_resolve}[\text{Physical_memory}, \text{pm_phy}](a_2, ac)(s))$$

Repeatedly Skolemizing and flattening,

This completes the proof of pm_phy_read_after_resolve_ok_TCC1.

Q.E.D.

C.117.86 Linear_Memory_Properties.pm_phy_read_after_resolve_ok

Terse proof for pm_phy_read_after_resolve_ok.

pm_phy_read_after_resolve_ok:

$$\{1\} \quad \forall (s: \text{Linear_memory}[\text{Physical_memory}, \text{pm_phy}], a_1: \text{Address}, a_2: \text{Memory_Address_4G},$$

$$ac: \text{Memory_access}):$$

$$\text{OK?}(\text{linear_resolve}(a_2, ac)(s)) \wedge$$

$$\text{plain_memory?}(\text{pm_phy}) \wedge$$

$$\text{pm_phy}'\text{states}(\text{state}(\text{linear_resolve}(a_2, ac)(s))) \wedge$$

$$\text{union}(\text{pm_phy}'\text{ro_addr}, \text{pm_phy}'\text{rw_addr})(a_1)$$

$$\supset \text{OK?}(\text{memory_read}(\text{pm_phy}'\text{mem})(a_1)(\text{state}(\text{linear_resolve}(a_2, ac)(s))))$$

Repeatedly Skolemizing and flattening,

Using lemma plain_memory_transformers_ok_read_ro_rw,

Using lemma expr_transformers_ok_ok,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Expanding the definition of memory_read_transformers,

Instantiating quantified variables,

This completes the proof of pm_phy_read_after_resolve_ok.

Q.E.D.

C.117.87 Linear_Memory_Properties.pm_resolve_address

Terse proof for pm_resolve_address.

pm_resolve_address:

<pre> {1} ∃ (pm: Plain_Memory[Linear_memory], s: Linear_memory[Physical_memory, pm_phy], a: Memory_Address_4G): is_linear_plain_memory?(pm) ∧ pm'states(s) ∧ virt_to_phys_range(s, restrict[Address, Memory_Address_4G, boolean] ((pm'ro_addr ∪ pm'rw_addr)), (singleton(Read) ∪ singleton(Execute))) (a) ⊃ difference((pm_phy'ro_addr ∪ pm_phy'rw_addr), extend[Address, Memory_Address_4G, bool, FALSE] (address_in_pt_range?(s, restrict[Address, Mem- ory_Address_4G, boolean] ((pm'ro_addr ∪ pm'rw_addr)))))) (a) </pre>
--

Repeatedly Skolemizing and flattening,

Using lemma pm_linear_blessed,

Expanding the definition of linear_blessed?,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Hiding formulas: (-1 -2 -3 -4 -6 -7 -9),

Case splitting on union(restrict[Address, Memory_Address_4G, boolean](pm!1'ro_addr), restrict[Address, Memory_Address_4G, boolean](pm!1'rw_addr)) = restrict[Address, Memory_Address_4G, boolean] (union(pm!1'ro_addr, pm!1'rw_addr)),

we get 2 subgoals:

pm_resolve_address.1:

<pre> {-1} (restrict[Address, Memory_Address_4G, boolean](pm'ro_addr) ∪ restrict [Address, Memory_Ad = restrict [Address, Memory_Address_4G, boolean]((pm'ro_addr ∪ pm'rw_addr)) {-2} (virt_to_phys_range(s', (restrict [Address, Memory_Address_4G, boolean](pm'ro_addr) ∪ restrict {-3} disjoint?(virt_to_phys_range(s', (restrict [Address, Memory_Address_4G, boolean](pm'ro_ad address_in_pt_range?(s', (restrict [Address, Memory_Address_4G, boolean](pm'ro {-4} Mem?(a' 'type_of) {-5} 0 ≤ a' 'offset {-6} a' 'offset < max_linear_offset {-7} virt_to_phys_range(s', restrict [Address, Memory_Address_4G, boolean] ((pm'ro_addr ∪ pm'rw_addr)), (singleton(Read) ∪ singleton(Execute))) (a') </pre> <hr/> <pre> {1} difference((pm_phy'ro_addr ∪ pm_phy'rw_addr), extend[Address, Memory_Address_4G, bool, FALSE] (address_in_pt_range?(s', restrict[Address, Mem- ory_Address_4G, boolean] ((pm'ro_addr ∪ pm'rw_addr)))))) (a') </pre>
--

Replacing using formula -1,
Hiding formulas: -1,
Trying repeated skolemization, instantiation, and if-lifting,
we get 2 subgoals:

pm_resolve_address.1.1:

<pre> {-1} pm_phy'ro_addr(a') {-2} Mem?(a' type_of) {-3} 0 ≤ a'offset {-4} a'offset < expt(2, bus_width) {-5} virt_to_phys_range(s', </pre>	<pre> restrict[Address, Memory_Address_4G, boolean] ((pm'ro_addr ∪ pm'rw_addr), (singleton(Read) ∪ singleton(Execute))) </pre>	
<pre> {-6} address_in_pt_range?(s', </pre>	<pre> restrict[Address, Memory_Address_4G, boolean] ((pm'ro_addr ∪ pm'rw_addr))) </pre>	
<pre> {1} virt_to_phys_range(s', </pre>		<pre> restrict[Address, Memory_Address_4G, boolean] ((pm'ro_addr ∪ pm'rw_addr))) </pre>

Keeping (-5 1) and hiding *,
Expanding the definition of virt_to_phys_range,
Expanding the definition of virt_to_phys_range,
Repeatedly Skolemizing and flattening,
Instantiating quantified variables,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of pm_resolve_address.1.1.

pm_resolve_address.1.2:

<pre> {-1} pm_phy'rw_addr(a') {-2} Mem?(a' type_of) {-3} 0 ≤ a'offset {-4} a'offset < expt(2, bus_width) {-5} virt_to_phys_range(s', </pre>	<pre> restrict[Address, Memory_Address_4G, boolean] ((pm'ro_addr ∪ pm'rw_addr), (singleton(Read) ∪ singleton(Execute))) </pre>	
<pre> {-6} address_in_pt_range?(s', </pre>	<pre> restrict[Address, Memory_Address_4G, boolean] ((pm'ro_addr ∪ pm'rw_addr))) </pre>	
<pre> {1} virt_to_phys_range(s', </pre>		<pre> restrict[Address, Memory_Address_4G, boolean] ((pm'ro_addr ∪ pm'rw_addr))) </pre>

Keeping (-5 1) and hiding *,
Expanding the definition of virt_to_phys_range,
Expanding the definition of virt_to_phys_range,
Repeatedly Skolemizing and flattening,

C Proof scripts

Instantiating quantified variables,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `pm_resolve_address.1.2`.

`pm_resolve_address.2`:

$$\frac{\begin{array}{l} \{-1\} \text{ (virt_to_phys_range}(s', \text{ (restrict [Address, Memory_Address_4G, boolean]}(pm'ro_addr) \cup \text{restrict} \\ \{-2\} \text{ disjoint?}(\text{virt_to_phys_range}(s', \\ \hspace{10em} \text{address_in_pt_range?}(s', \\ \hspace{10em} \text{(restrict [Address, Memory_Address_4G, boolean]}(pm'ro_ad} \\ \hspace{10em} \text{(restrict [Address, Memory_Address_4G, boolean]}(pm'ro_ad} \\ \{-3\} \text{ Mem?}(a' \text{'type_of}) \\ \{-4\} \text{ } 0 \leq a' \text{'offset} \\ \{-5\} \text{ } a' \text{'offset} < \text{max_linear_offset} \\ \{-6\} \text{ virt_to_phys_range}(s', \\ \hspace{10em} \text{restrict [Address, Memory_Address_4G, boolean]} \\ \hspace{10em} \text{((pm'ro_addr} \cup \text{pm'rw_addr)),} \\ \hspace{10em} \text{(singleton(Read) } \cup \text{singleton(Execute)))} \\ \hspace{10em} (a') \end{array}}{\begin{array}{l} \{1\} \text{ (restrict [Address, Memory_Address_4G, boolean]}(pm'ro_addr) \cup \text{restrict [Address, Memory_Ad} \\ \hspace{10em} \text{= restrict [Address, Memory_Address_4G, boolean]}(\text{(pm'ro_addr} \cup \text{pm'rw_addr)}) \\ \{2\} \text{ difference}(\text{(pm_phy'ro_addr} \cup \text{pm_phy'rw_addr)}, \\ \hspace{10em} \text{extend [Address, Memory_Address_4G, bool, FALSE]} \\ \hspace{10em} \text{(address_in_pt_range?}(s', \\ \hspace{10em} \text{restrict [Address, Mem-} \\ \hspace{10em} \text{ory_Address_4G, boolean]} \\ \hspace{10em} \text{((pm'ro_addr} \cup \text{pm'rw_addr)))))) \\ \hspace{10em} (a') \end{array}}$$

Keeping (1) and hiding *,

Applying `decompose-equality`,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `pm_resolve_address.2`.

Q.E.D.

C.117.88 Linear_Memory_Properties.pm_resolve_address_write

Terse proof for `pm_resolve_address_write`.

pm_resolve_address_write:

{1}	$\forall (pm: \text{Plain_Memory}[\text{Linear_memory}], s: \text{Linear_memory}[\text{Physical_memory}, pm_phy], a: \text{Memory_Address_4G}):$ $\text{is_linear_plain_memory?}(pm) \wedge$ $pm' \text{states}(s) \wedge$ $\text{virt_to_phys_range}(s, \text{restrict}[\text{Address}, \text{Memory_Address_4G}, \text{boolean}](pm' \text{rw_addr}), \text{singleton}(\text{Write}))$ (a) \supset $\text{difference}((pm_phy' \text{ro_addr} \cup pm_phy' \text{rw_addr}), \text{extend}[\text{Address}, \text{Memory_Address_4G}, \text{bool}, \text{FALSE}](\text{address_in_pt_range?}(s, \text{restrict}[\text{Address}, \text{Memory_Address_4G}, \text{boolean}](pm' \text{ro_addr} \cup pm' \text{rw_addr}))))$ (a)
-----	--

Repeatedly Skolemizing and flattening,

Using lemma pm_linear_blessed,

Expanding the definition of linear_blessed?,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Hiding formulas: (-1 -2 -3 -4 -5 -7 -9),

Case splitting on $\text{union}(\text{restrict}[\text{Address}, \text{Memory_Address_4G}, \text{boolean}](pm!1' \text{ro_addr}), \text{restrict}[\text{Address}, \text{Memory_Address_4G}, \text{boolean}](pm!1' \text{rw_addr})) = \text{restrict}[\text{Address}, \text{Memory_Address_4G}, \text{boolean}](\text{union}(pm!1' \text{ro_addr}, pm!1' \text{rw_addr}))$,

we get 2 subgoals:

pm_resolve_address_write.1:

{-1}	$(\text{restrict}[\text{Address}, \text{Memory_Address_4G}, \text{boolean}](pm' \text{ro_addr}) \cup \text{restrict}[\text{Address}, \text{Memory_Address_4G}, \text{boolean}](pm' \text{rw_addr})) = \text{restrict}[\text{Address}, \text{Memory_Address_4G}, \text{boolean}](pm' \text{ro_addr} \cup pm' \text{rw_addr})$
{-2}	$(\text{virt_to_phys_range}(s', \text{restrict}[\text{Address}, \text{Memory_Address_4G}, \text{boolean}](pm' \text{rw_addr}), \text{singleton}(\text{Write}))) \subseteq$
{-3}	$\text{disjoint?}(\text{virt_to_phys_range}(s', \text{restrict}[\text{Address}, \text{Memory_Address_4G}, \text{boolean}](pm' \text{ro_addr}) \cup \text{restrict}[\text{Address}, \text{Memory_Address_4G}, \text{boolean}](pm' \text{rw_addr})), \text{address_in_pt_range?}(s', \text{restrict}[\text{Address}, \text{Memory_Address_4G}, \text{boolean}](pm' \text{ro_addr}) \cup \text{restrict}[\text{Address}, \text{Memory_Address_4G}, \text{boolean}](pm' \text{rw_addr})))$
{-4}	$\text{Mem?}(a' \text{'type_of})$
{-5}	$0 \leq a' \text{'offset}$
{-6}	$a' \text{'offset} < \text{max_linear_offset}$
{-7}	$\text{virt_to_phys_range}(s', \text{restrict}[\text{Address}, \text{Memory_Address_4G}, \text{boolean}](pm' \text{rw_addr}), \text{singleton}(\text{Write}))$ (a')
{1}	$\text{difference}((pm_phy' \text{ro_addr} \cup pm_phy' \text{rw_addr}), \text{extend}[\text{Address}, \text{Memory_Address_4G}, \text{bool}, \text{FALSE}](\text{address_in_pt_range?}(s', \text{restrict}[\text{Address}, \text{Memory_Address_4G}, \text{boolean}](pm' \text{ro_addr} \cup pm' \text{rw_addr}))))$ (a')

Replacing using formula -1,

Hiding formulas: -1,

Trying repeated skolemization, instantiation, and if-lifting,

C Proof scripts

Keeping (-5 1) and hiding *,

Expanding the definition of `virt_to_phys_range`,

Expanding the definition of `virt_to_phys_range`,

Repeatedly Skolemizing and flattening,

Instantiating quantified variables,

we get 2 subgoals:

`pm_resolve_address_write.1.1:`

{-1}	<code>Mem?(lin_a' type_of)</code>
{-2}	<code>0 ≤ lin_a' offset</code>
{-3}	<code>lin_a' offset < max_linear_offset</code>
{-4}	<code>restrict[Address, Memory_Address_4G, boolean](pm' rw_addr)(lin_a')</code>
{-5}	<code>singleton(Write)(ac')</code>
{-6}	<code>OK?(linear_resolve(lin_a', ac')(s'))</code>
{-7}	<code>data(linear_resolve(lin_a', ac')(s')) = a'</code>
<hr/>	
{1}	<code>OK?(linear_resolve(lin_a', ac')(s')) ∧ data(linear_resolve(lin_a', ac')(s')) = a'</code>

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `pm_resolve_address_write.1.1`.

`pm_resolve_address_write.1.2:`

{-1}	<code>Mem?(lin_a' type_of)</code>
{-2}	<code>0 ≤ lin_a' offset</code>
{-3}	<code>lin_a' offset < max_linear_offset</code>
{-4}	<code>restrict[Address, Memory_Address_4G, boolean](pm' rw_addr)(lin_a')</code>
{-5}	<code>singleton(Write)(ac')</code>
{-6}	<code>OK?(linear_resolve(lin_a', ac')(s'))</code>
{-7}	<code>data(linear_resolve(lin_a', ac')(s')) = a'</code>
<hr/>	
{1}	<code>pm' ro_addr(lin_a') ∨ pm' rw_addr(lin_a')</code>

Keeping (-4 1) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `pm_resolve_address_write.1.2`.

pm_resolve_address_write.2:

{-1}	(virt_to_phys_range(s' , restrict [Address, Memory_Address_4G, boolean](pm' rw_addr), singleton(Write))) \supseteq				
{-2}	disjoint?(virt_to_phys_range(s' , <div style="margin-left: 100px;">(restrict [Address, Memory_Address_4G, boolean](pm'ro_addr) \cup restrict [Address, Memory_Address_4G, boolean](pm'rw_addr)))</div> <div style="margin-left: 100px;">address_in_pt_range?(s', <div style="margin-left: 100px;">(restrict [Address, Memory_Address_4G, boolean](pm'ro_addr) \cup restrict [Address, Memory_Address_4G, boolean](pm'rw_addr)))</div> </div>				
{-3}	Mem?(a' 'type_of)				
{-4}	$0 \leq a'$ 'offset				
{-5}	a' 'offset < max_linear_offset				
{-6}	virt_to_phys_range(s' , restrict [Address, Memory_Address_4G, boolean](pm' rw_addr), <div style="margin-left: 100px;">singleton(Write)) <div style="margin-left: 100px;">(a'))</div> </div>				
<table style="width: 100%; border-collapse: collapse;"> <tr> <td style="width: 5%; vertical-align: top;">{1}</td> <td style="padding: 2px;">(restrict [Address, Memory_Address_4G, boolean](pm'ro_addr) \cup restrict [Address, Memory_Address_4G, boolean](pm'rw_addr)) = restrict [Address, Memory_Address_4G, boolean](((pm'ro_addr \cup pm'rw_addr)))</td> </tr> <tr> <td style="vertical-align: top;">{2}</td> <td style="padding: 2px;">difference((pm_phy'ro_addr \cup pm_phy'rw_addr), <div style="margin-left: 100px;">extend [Address, Memory_Address_4G, bool, FALSE] <div style="margin-left: 100px;">(address_in_pt_range?(s', <div style="margin-left: 100px;">restrict [Address, Memory_Address_4G, boolean] <div style="margin-left: 100px;">((pm'ro_addr \cup pm'rw_addr))))</div> </div> </div> <div style="margin-left: 100px;">(a'))</div> </div></td> </tr> </table>		{1}	(restrict [Address, Memory_Address_4G, boolean](pm' ro_addr) \cup restrict [Address, Memory_Address_4G, boolean](pm' rw_addr)) = restrict [Address, Memory_Address_4G, boolean](((pm' ro_addr \cup pm' rw_addr)))	{2}	difference((pm_phy' ro_addr \cup pm_phy' rw_addr), <div style="margin-left: 100px;">extend [Address, Memory_Address_4G, bool, FALSE] <div style="margin-left: 100px;">(address_in_pt_range?(s', <div style="margin-left: 100px;">restrict [Address, Memory_Address_4G, boolean] <div style="margin-left: 100px;">((pm'ro_addr \cup pm'rw_addr))))</div> </div> </div> <div style="margin-left: 100px;">(a'))</div> </div>
{1}	(restrict [Address, Memory_Address_4G, boolean](pm' ro_addr) \cup restrict [Address, Memory_Address_4G, boolean](pm' rw_addr)) = restrict [Address, Memory_Address_4G, boolean](((pm' ro_addr \cup pm' rw_addr)))				
{2}	difference((pm_phy' ro_addr \cup pm_phy' rw_addr), <div style="margin-left: 100px;">extend [Address, Memory_Address_4G, bool, FALSE] <div style="margin-left: 100px;">(address_in_pt_range?(s', <div style="margin-left: 100px;">restrict [Address, Memory_Address_4G, boolean] <div style="margin-left: 100px;">((pm'ro_addr \cup pm'rw_addr))))</div> </div> </div> <div style="margin-left: 100px;">(a'))</div> </div>				

Keeping (1) and hiding *,
Applying decompose-equality,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of pm_resolve_address_write.2.
Q.E.D.

C.118 Proofs for List_Split (linear_memory.pvs)

C.118.1 List_Split.split_TCC1

Terse proof for split_TCC1.

split_TCC1:

{1}	\forall (size: nat, a : Address, bl: list[Byte], e2size: posnat): $\neg a + \text{length}(bl) \leq \text{floor}(e2size)(a) + e2size \wedge$ $\neg \text{null?}(bl) \wedge e2size = \text{expt}(2, \text{size})$ \supset $(\forall (\delta: \{r: \text{mod}(e2size) \mid \exists q: \text{offset}(a) = r + e2size \times q\}):$ $\delta = \text{rem}(e2size)(\text{offset}(a)) \supset$ $e2size - \delta \geq 0 \wedge e2size - \delta \leq \text{length}[\text{Byte}](bl))$
-----	---

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of split_TCC1.
Q.E.D.

C.118.2 List_Split.split_TCC2

Terse proof for split_TCC2.

split_TCC2:

$\begin{aligned} \{1\} \quad & \forall (\text{size}: \text{nat}, a: \text{Address}, \text{bl}: \text{list}[\text{Byte}], \text{e2size}: \text{posnat}): \\ & \neg a + \text{length}(\text{bl}) \leq \text{floor}(\text{e2size})(a) + \text{e2size} \wedge \\ & \neg \text{null}?(\text{bl}) \wedge \text{e2size} = \text{expt}(2, \text{size}) \\ & \supset \\ & (\forall (\delta: \{r: \text{mod}(\text{e2size}) \mid \exists q: \text{offset}(a) = r + \text{e2size} \times q\}): \\ & \quad \delta = \text{rem}(\text{e2size})(\text{offset}(a)) \supset \\ & \quad \text{length}[\text{Byte}](\text{tail}[\text{Byte}](\text{bl}, \text{e2size} - \delta)) < \text{length}[\text{Byte}](\text{bl})) \end{aligned}$
--

Repeatedly Skolemizing and flattening,

Replacing using formula -8,

Rewriting using length_tail, matching in *,

we get 2 subgoals:

split_TCC2.1:

$\begin{aligned} \{-1\} \quad & \text{integer_pred}(q') \\ \{-2\} \quad & \text{rem}(\text{e2size}')(\text{offset}(a')) < \text{e2size}' \\ \{-3\} \quad & \text{offset}(a') = \text{rem}(\text{e2size}')(\text{offset}(a')) + \text{e2size}' \times q' \\ \{-4\} \quad & \text{size}' \geq 0 \\ \{-5\} \quad & \text{every}(\lambda (x: \text{number}): \\ & \quad \text{number_field_pred}(x) \wedge \\ & \quad \text{real_pred}(x) \wedge \\ & \quad \text{rational_pred}(x) \wedge \text{integer_pred}(x) \wedge x \geq 0 \wedge x < \text{max_byte}) \\ & \quad (\text{bl}') \\ \{-6\} \quad & \text{e2size}' > 0 \\ \{-7\} \quad & \text{e2size}' = \text{expt}(2, \text{size}') \\ \{-8\} \quad & \text{delta}' = \text{rem}(\text{e2size}')(\text{offset}(a')) \end{aligned}$
$\begin{aligned} \{1\} \quad & a' + \text{length}(\text{bl}') \leq \text{floor}(\text{e2size}')(\text{offset}(a')) + \text{e2size}' \\ \{2\} \quad & \text{null}?(\text{bl}') \\ \{3\} \quad & \text{rem}(\text{e2size}')(\text{offset}(a')) + \text{length}(\text{bl}') - \text{e2size}' < \text{length}[\text{Byte}](\text{bl}') \end{aligned}$

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of split_TCC2.1.

split_TCC2.2:

$\begin{aligned} \{-1\} \quad & \text{integer_pred}(q') \\ \{-2\} \quad & \text{rem}(\text{e2size}')(\text{offset}(a')) < \text{e2size}' \\ \{-3\} \quad & \text{offset}(a') = \text{rem}(\text{e2size}')(\text{offset}(a')) + \text{e2size}' \times q' \\ \{-4\} \quad & \text{size}' \geq 0 \\ \{-5\} \quad & \text{every}(\lambda (x: \text{number}): \\ & \quad \text{number_field_pred}(x) \wedge \\ & \quad \text{real_pred}(x) \wedge \\ & \quad \text{rational_pred}(x) \wedge \text{integer_pred}(x) \wedge x \geq 0 \wedge x < \text{max_byte}) \\ & \quad (\text{bl}') \\ \{-6\} \quad & \text{e2size}' > 0 \\ \{-7\} \quad & \text{e2size}' = \text{expt}(2, \text{size}') \\ \{-8\} \quad & \text{delta}' = \text{rem}(\text{e2size}')(\text{offset}(a')) \end{aligned}$
$\begin{aligned} \{1\} \quad & \text{e2size}' - \text{rem}(\text{e2size}')(\text{offset}(a')) \leq \text{length}(\text{bl}') \\ \{2\} \quad & a' + \text{length}(\text{bl}') \leq \text{floor}(\text{e2size}')(\text{offset}(a')) + \text{e2size}' \\ \{3\} \quad & \text{null}?(\text{bl}') \\ \{4\} \quad & \text{length}[\text{Byte}](\text{tail}[\text{Byte}](\text{bl}', \text{e2size}' - \text{rem}(\text{e2size}')(\text{offset}(a')))) < \\ & \quad \text{length}[\text{Byte}](\text{bl}') \end{aligned}$

Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of `split_TCC2.2`.
 Q.E.D.

C.118.3 List_Split.split_pair_cross_size

Terse proof for `split_pair_cross_size`.

`split_pair_cross_size`:

```
{1}  ∀ (size: nat, a1: Address, bl1: list[Byte], n: nat):
      LET e2size = expt(2, size) IN
      n = length(bl1) ⊃
      every(λ (a2: Address, bl2: list[Byte]):
            rem(e2size)(offset(a2)) + length(bl2) ≤ e2size)
      (split(size, a1, bl1))
```

Inducting on n on formula 1 using induction scheme `NAT_induction`,
 we get 2 subgoals:

`split_pair_cross_size.1`:

```
{-1}  ∀ (size: nat, a1: Address, bl1: list[Byte]):
      n' = length(bl1) ⊃
      every(λ (a2: Address, bl2: list[Byte]):
            rem(expt(2, size))(offset(a2)) + length(bl2) ≤ expt(2, size))
      (split(size, a1, bl1))
```

```
{1}  ∀ (size: nat, a1: Address, bl1: list[Byte]):
      LET e2size = expt(2, size) IN
      n' = length(bl1) ⊃
      every(λ (a2: Address, bl2: list[Byte]):
            rem(e2size)(offset(a2)) + length(bl2) ≤ e2size)
      (split(size, a1, bl1))
```

Repeatedly Skolemizing and flattening,
 Instantiating the top quantifier in -3 with the terms: (size!1 a1!1 bl1!1),
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 This completes the proof of `split_pair_cross_size.1`.

`split_pair_cross_size.2`:

```
{1}  ∀ j:
      (∀ k:
         k < j ⊃
         (∀ (size: nat, a1: Address, bl1: list[Byte]):
            k = length(bl1) ⊃
            every(λ (a2: Address, bl2: list[Byte]):
                  rem(expt(2, size))(offset(a2)) + length(bl2) ≤
                  expt(2, size))
            (split(size, a1, bl1))))
      ⊃
      (∀ (size: nat, a1: Address, bl1: list[Byte]):
         j = length(bl1) ⊃
         every(λ (a2: Address, bl2: list[Byte]):
               rem(expt(2, size))(offset(a2)) + length(bl2) ≤ expt(2, size))
         (split(size, a1, bl1)))
```

C Proof scripts

Repeatedly Skolemizing and flattening,
 Expanding the definition of split,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 we get 3 subgoals:

split_pair_cross_size.2.1:

{-1}	$size' \geq 0$
{-2}	$every(\lambda (x: number):$ $number_field_pred(x) \wedge$ $real_pred(x) \wedge$ $rational_pred(x) \wedge integer_pred(x) \wedge x \geq 0 \wedge x < max_byte)$ $(bl1')$
{-3}	$j' \geq 0$
{-4}	$\forall k:$ $k < j' \supset$ $(\forall (size: nat, a_1: Address, bl1: list[Byte]):$ $k = length(bl1) \supset$ $every(\lambda (a_2: Address, bl2: list[Byte]):$ $rem(expt(2, size))(offset(a_2)) + length(bl2) \leq expt(2, size))$ $(split(size, a_1, bl1)))$
{-5}	$j' = length(bl1')$
{-6}	$null?(bl1')$
{1}	$every(\lambda (a_2: Address, bl2: list[Byte]):$ $rem(expt(2, size'))(offset(a_2)) + length(bl2) \leq expt(2, size'))$ $(null)$

Expanding the definition of every,
 which is trivially true.

This completes the proof of **split_pair_cross_size.2.1**.

split_pair_cross_size.2.2:

{-1}	$size' \geq 0$
{-2}	$every(\lambda (x: number):$ $number_field_pred(x) \wedge$ $real_pred(x) \wedge$ $rational_pred(x) \wedge integer_pred(x) \wedge x \geq 0 \wedge x < max_byte)$ $(bl1')$
{-3}	$j' \geq 0$
{-4}	$\forall k:$ $k < j' \supset$ $(\forall (size: nat, a_1: Address, bl1: list[Byte]):$ $k = length(bl1) \supset$ $every(\lambda (a_2: Address, bl2: list[Byte]):$ $rem(expt(2, size))(offset(a_2)) + length(bl2) \leq expt(2, size))$ $(split(size, a_1, bl1)))$
{-5}	$j' = length(bl1')$
{-6}	$a'_1 + length(bl1') \leq floor(expt(2, size'))(a'_1) + expt(2, size')$
{1}	$null?(bl1')$
{2}	$every(\lambda (a_2: Address, bl2: list[Byte]):$ $rem(expt(2, size'))(offset(a_2)) + length(bl2) \leq expt(2, size'))$ $(cons((a'_1, bl1'), null))$

Expanding the definition of every,
 Expanding the definition of every,
 Keeping (-6 2) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of split_pair_cross_size.2.2.

split_pair_cross_size.2.3:

<pre> {-1} size' ≥ 0 {-2} every(λ (x: number): number_field_pred(x) ∧ real_pred(x) ∧ rational_pred(x) ∧ integer_pred(x) ∧ x ≥ 0 ∧ x < max_byte) (bl1') {-3} j' ≥ 0 {-4} ∀ k: k < j' ⊃ (∀ (size: nat, a1: Address, bl1: list[Byte]): k = length(bl1) ⊃ every(λ (a2: Address, bl2: list[Byte]): rem(expt(2, size))(offset(a2)) + length(bl2) ≤ expt(2, size)) (split(size, a1, bl1)))) {-5} j' = length(bl1')</pre>	<hr style="border: 0.5px solid black;"/> <pre> {1} null?(bl1') {2} a1' + length(bl1') ≤ floor(expt(2, size'))(a1') + expt(2, size') {3} every(λ (a2: Address, bl2: list[Byte]): rem(expt(2, size'))(offset(a2)) + length(bl2) ≤ expt(2, size')) (cons((a1', head(bl1', expt(2, size') - rem(expt(2, size'))(offset(a1')))), split(size', floor(expt(2, size'))(a1') + expt(2, size'), tail(bl1', expt(2, size') - rem(expt(2, size'))(offset(a1'))))))</pre>
--	--

Expanding the definition of every,
Applying propositional simplification,
we get 2 subgoals:

split_pair_cross_size.2.3.1:

<pre> {-1} size' ≥ 0 {-2} every(λ (x: number): number_field_pred(x) ∧ real_pred(x) ∧ rational_pred(x) ∧ integer_pred(x) ∧ x ≥ 0 ∧ x < max_byte) (bl1') {-3} j' ≥ 0 {-4} ∀ k: k < j' ⊃ (∀ (size: nat, a1: Address, bl1: list[Byte]): k = length(bl1) ⊃ every(λ (a2: Address, bl2: list[Byte]): rem(expt(2, size))(offset(a2)) + length(bl2) ≤ expt(2, size)) (split(size, a1, bl1)))) {-5} j' = length(bl1')</pre>	<hr style="border: 0.5px solid black;"/> <pre> {1} rem(expt(2, size'))(offset(a1')) + length(head(bl1', expt(2, size') - rem(expt(2, size'))(offset(a1')))) ≤ expt(2, size') {2} null?(bl1') {3} a1' + length(bl1') ≤ floor(expt(2, size'))(a1') + expt(2, size')</pre>
--	---

C Proof scripts

Rewriting using `length_head`, matching in `*`,
we get 2 subgoals:

`split_pair_cross_size.2.3.1.1:`

{-1}	$size' \geq 0$
{-2}	every($\lambda (x: \text{number})$): $\text{number_field_pred}(x) \wedge$ $\text{real_pred}(x) \wedge$ $\text{rational_pred}(x) \wedge \text{integer_pred}(x) \wedge x \geq 0 \wedge x < \text{max_byte}$
	(bl1')
{-3}	$j' \geq 0$
{-4}	$\forall k$: $k < j' \supset$ ($\forall (\text{size}: \text{nat}, a_1: \text{Address}, \text{bl1}: \text{list}[\text{Byte}]$): $k = \text{length}(\text{bl1}) \supset$ every($\lambda (a_2: \text{Address}, \text{bl2}: \text{list}[\text{Byte}]$): $\text{rem}(\text{expt}(2, \text{size}))(\text{offset}(a_2)) + \text{length}(\text{bl2}) \leq \text{expt}(2, \text{size})$) (split(size, a_1 , bl1)))
{-5}	$j' = \text{length}(\text{bl1}')$
{1}	$\text{rem}(\text{expt}(2, \text{size}'))(\text{offset}(a_1')) + \text{expt}(2, \text{size}') - \text{rem}(\text{expt}(2, \text{size}'))(\text{offset}(a_1'))$ $\leq \text{expt}(2, \text{size}')$
{2}	$\text{null}?(bl1')$
{3}	$a_1' + \text{length}(bl1') \leq \text{floor}(\text{expt}(2, \text{size}'))(a_1') + \text{expt}(2, \text{size}')$

Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `split_pair_cross_size.2.3.1.1`.

`split_pair_cross_size.2.3.1.2:`

{-1}	$size' \geq 0$
{-2}	every($\lambda (x: \text{number})$): $\text{number_field_pred}(x) \wedge$ $\text{real_pred}(x) \wedge$ $\text{rational_pred}(x) \wedge \text{integer_pred}(x) \wedge x \geq 0 \wedge x < \text{max_byte}$
	(bl1')
{-3}	$j' \geq 0$
{-4}	$\forall k$: $k < j' \supset$ ($\forall (\text{size}: \text{nat}, a_1: \text{Address}, \text{bl1}: \text{list}[\text{Byte}]$): $k = \text{length}(\text{bl1}) \supset$ every($\lambda (a_2: \text{Address}, \text{bl2}: \text{list}[\text{Byte}]$): $\text{rem}(\text{expt}(2, \text{size}))(\text{offset}(a_2)) + \text{length}(\text{bl2}) \leq \text{expt}(2, \text{size})$) (split(size, a_1 , bl1)))
{-5}	$j' = \text{length}(\text{bl1}')$
{1}	$\text{expt}(2, \text{size}') - \text{rem}(\text{expt}(2, \text{size}'))(\text{offset}(a_1')) \leq \text{length}(\text{bl1}')$
{2}	$\text{rem}(\text{expt}(2, \text{size}'))(\text{offset}(a_1')) + \text{length}(\text{head}(\text{bl1}', \text{expt}(2, \text{size}') - \text{rem}(\text{expt}(2, \text{size}'))(\text{offset}(a_1'))))$ $\leq \text{expt}(2, \text{size}')$
{3}	$\text{null}?(bl1')$
{4}	$a_1' + \text{length}(bl1') \leq \text{floor}(\text{expt}(2, \text{size}'))(a_1') + \text{expt}(2, \text{size}')$

Expanding the definition of `floor`,

Expanding the definition of `floor`,

Expanding the definition of `<=`,

Expanding the definition of `+`,

which is trivially true.

This completes the proof of `split_pair_cross_size.2.3.1.2`.

split_pair_cross_size.2.3.2:

{-1}	$size' \geq 0$
{-2}	every($\lambda (x: \text{number})$): $\text{number_field_pred}(x) \wedge$ $\text{real_pred}(x) \wedge$ $\text{rational_pred}(x) \wedge \text{integer_pred}(x) \wedge x \geq 0 \wedge x < \text{max_byte}$) $(bl1')$
{-3}	$j' \geq 0$
{-4}	$\forall k$: $k < j' \supset$ $(\forall (\text{size}: \text{nat}, a_1: \text{Address}, bl1: \text{list}[\text{Byte}]$): $k = \text{length}(bl1) \supset$ $\text{every}(\lambda (a_2: \text{Address}, bl2: \text{list}[\text{Byte}]$): $\text{rem}(\text{expt}(2, \text{size}))(\text{offset}(a_2)) + \text{length}(bl2) \leq \text{expt}(2, \text{size}))$ $(\text{split}(\text{size}, a_1, bl1)))$)
{-5}	$j' = \text{length}(bl1')$
{1}	every($\lambda (a_2: \text{Address}, bl2: \text{list}[\text{Byte}]$): $\text{rem}(\text{expt}(2, \text{size}'))(\text{offset}(a_2)) + \text{length}(bl2) \leq \text{expt}(2, \text{size}')$ $(\text{split}(\text{size}', \text{floor}(\text{expt}(2, \text{size}'))(a_1') + \text{expt}(2, \text{size}')$, $\text{tail}(bl1'$, $\text{expt}(2, \text{size}') - \text{rem}(\text{expt}(2, \text{size}'))(\text{offset}(a_1'))))$)
{2}	$\text{null?}(bl1')$
{3}	$a_1' + \text{length}(bl1') \leq \text{floor}(\text{expt}(2, \text{size}'))(a_1') + \text{expt}(2, \text{size}')$

Instantiating the top quantifier in -4 with the terms: $(\text{length}(\text{tail}(bl1!1, \text{expt}(2, \text{size}!1) - \text{rem}(\text{expt}(2, \text{size}!1))(\text{offset}(a1!1))))$,

Splitting conjunctions,

we get 2 subgoals:

split_pair_cross_size.2.3.2.1:

{-1}	$\forall (\text{size}: \text{nat}, a_1: \text{Address}, bl1: \text{list}[\text{Byte}]$): $\text{length}(\text{tail}(bl1', \text{expt}(2, \text{size}') - \text{rem}(\text{expt}(2, \text{size}'))(\text{offset}(a_1')))) =$ $\text{length}(bl1)$ \supset $\text{every}(\lambda (a_2: \text{Address}, bl2: \text{list}[\text{Byte}]$): $\text{rem}(\text{expt}(2, \text{size}))(\text{offset}(a_2)) + \text{length}(bl2) \leq \text{expt}(2, \text{size}))$ $(\text{split}(\text{size}, a_1, bl1))$)
{-2}	$size' \geq 0$
{-3}	every($\lambda (x: \text{number})$): $\text{number_field_pred}(x) \wedge$ $\text{real_pred}(x) \wedge$ $\text{rational_pred}(x) \wedge \text{integer_pred}(x) \wedge x \geq 0 \wedge x < \text{max_byte}$) $(bl1')$
{-4}	$j' \geq 0$
{-5}	$j' = \text{length}(bl1')$
{1}	every($\lambda (a_2: \text{Address}, bl2: \text{list}[\text{Byte}]$): $\text{rem}(\text{expt}(2, \text{size}'))(\text{offset}(a_2)) + \text{length}(bl2) \leq \text{expt}(2, \text{size}')$ $(\text{split}(\text{size}', \text{floor}(\text{expt}(2, \text{size}'))(a_1') + \text{expt}(2, \text{size}')$, $\text{tail}(bl1'$, $\text{expt}(2, \text{size}') - \text{rem}(\text{expt}(2, \text{size}'))(\text{offset}(a_1'))))$)
{2}	$\text{null?}(bl1')$
{3}	$a_1' + \text{length}(bl1') \leq \text{floor}(\text{expt}(2, \text{size}'))(a_1') + \text{expt}(2, \text{size}')$

Instantiating the top quantifier in -1 with the terms: $(\text{size}!1 \text{ floor}(\text{expt}(2, \text{size}!1))(a1!1) + \text{expt}(2,$

C Proof scripts

size!1) tail(bl1!1, expt(2, size!1) - rem(expt(2, size!1))(offset(a1!1)))),

which is trivially true.

This completes the proof of `split_pair_cross_size.2.3.2.1`.

`split_pair_cross_size.2.3.2.2`:

<pre> {-1} size' ≥ 0 {-2} every(λ (x: number): number_field_pred(x) ∧ real_pred(x) ∧ rational_pred(x) ∧ integer_pred(x) ∧ x ≥ 0 ∧ x < max_byte) (bl1') {-3} j' ≥ 0 {-4} j' = length(bl1')</pre>	<pre> {1} length(tail(bl1', expt(2, size') - rem(expt(2, size'))(offset(a1')))) < j' {2} every(λ (a2: Address, bl2: list[Byte]): rem(expt(2, size'))(offset(a2)) + length(bl2) ≤ expt(2, size')) (split(size', floor(expt(2, size'))(a1) + expt(2, size'), tail(bl1', expt(2, size') - rem(expt(2, size'))(offset(a1'))))) {3} null?(bl1') {4} a1 + length(bl1') ≤ floor(expt(2, size'))(a1) + expt(2, size')</pre>
--	---

Rewriting using `length_tail`, matching in `*`,

we get 2 subgoals:

`split_pair_cross_size.2.3.2.2.1`:

<pre> {-1} size' ≥ 0 {-2} every(λ (x: number): number_field_pred(x) ∧ real_pred(x) ∧ rational_pred(x) ∧ integer_pred(x) ∧ x ≥ 0 ∧ x < max_byte) (bl1') {-3} j' ≥ 0 {-4} j' = length(bl1')</pre>	<pre> {1} rem(expt(2, size'))(offset(a1)) + length(bl1') - expt(2, size') < j' {2} every(λ (a2: Address, bl2: list[Byte]): rem(expt(2, size'))(offset(a2)) + length(bl2) ≤ expt(2, size')) (split(size', floor(expt(2, size'))(a1) + expt(2, size'), tail(bl1', expt(2, size') - rem(expt(2, size'))(offset(a1'))))) {3} null?(bl1') {4} a1 + length(bl1') ≤ floor(expt(2, size'))(a1) + expt(2, size')</pre>
--	--

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `split_pair_cross_size.2.3.2.2.1`.

split_pair_cross_size.2.3.2.2.2:

<pre> {-1} size' ≥ 0 {-2} every(λ (x: number): number_field_pred(x) ∧ real_pred(x) ∧ rational_pred(x) ∧ integer_pred(x) ∧ x ≥ 0 ∧ x < max_byte) (bl1') {-3} j' ≥ 0 {-4} j' = length(bl1')</pre>	<hr/> <pre> {1} expt(2, size') - rem(expt(2, size'))(offset(a1')) ≤ length(bl1') {2} length(tail(bl1', expt(2, size') - rem(expt(2, size'))(offset(a1')))) < j' {3} every(λ (a2: Address, bl2: list[Byte]): rem(expt(2, size'))(offset(a2)) + length(bl2) ≤ expt(2, size') (split(size', floor(expt(2, size'))(a1') + expt(2, size'), tail(bl1', expt(2, size') - rem(expt(2, size'))(offset(a1'))))) {4} null?(bl1') {5} a1' + length(bl1') ≤ floor(expt(2, size'))(a1') + expt(2, size')</pre>
--	---

Hiding formulas: -2, 3,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of split_pair_cross_size.2.3.2.2.2.

Q.E.D.

C.118.4 List_Split.split_pair_concat

Terse proof for split_pair_concat.

split_pair_concat:

<pre> {1} ∀ (size: nat, a: Address, bl: list[Byte], n: nat): LET e2size = expt(2, size) IN n = length(bl) ⊃ reduce(null, λ (e: [Address, list[Byte]], tail: list[Byte]): LET head = e'2 IN append(head, tail) (split(size, a, bl)) = bl</pre>	<hr/>
---	-------

Inducting on n on formula 1 using induction scheme NAT_induction,

we get 2 subgoals:

`split_pair_concat.1:`

<pre>{-1} ∀ (size: nat, a: Address, bl: list[Byte]): n' = length(bl) ⊃ reduce(null, λ (e: [Address, list[Byte]], tail: list[Byte]): append(e'2, tail)) (split(size, a, bl)) = bl</pre>	<hr style="border: 0.5px solid black;"/> <pre>{1} ∀ (size: nat, a: Address, bl: list[Byte]): LET e2size = expt(2, size) IN n' = length(bl) ⊃ reduce(null, λ (e: [Address, list[Byte]], tail: list[Byte]): LET head = e'2 IN append(head, tail)) (split(size, a, bl)) = bl</pre>
---	--

Repeatedly Skolemizing and flattening,

Simplifying, rewriting, and recording with decision procedures,

Hiding formulas: -2,

Instantiating quantified variables,

This completes the proof of `split_pair_concat.1`.

`split_pair_concat.2:`

<pre>{1} ∀ j: (∀ k: k < j ⊃ (∀ (size: nat, a: Address, bl: list[Byte]): k = length(bl) ⊃ reduce(null, λ (e: [Address, list[Byte]], tail: list[Byte]): append(e'2, tail)) (split(size, a, bl)) = bl)) ⊃ (∀ (size: nat, a: Address, bl: list[Byte]): j = length(bl) ⊃ reduce(null, λ (e: [Address, list[Byte]], tail: list[Byte]): append(e'2, tail)) (split(size, a, bl)) = bl)</pre>	<hr style="border: 0.5px solid black;"/>
--	--

Repeatedly Skolemizing and flattening,

Expanding the definition of split,

Expanding the definition of reduce,

Hiding formulas: -2,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 2 subgoals:

split_pair_concat.2.1:

<pre> {-1} size' ≥ 0 {-2} j' ≥ 0 {-3} ∀ k: k < j' ⊃ (∀ (size: nat, a: Address, bl: list[Byte]): k = length(bl) ⊃ reduce(null, λ (e: [Address, list[Byte]], tail: list[Byte]): append(e'2, tail)) (split(size, a, bl)) = bl) {-4} j' = length(bl') {-5} a' + length(bl') ≤ floor(expt(2, size'))(a') + expt(2, size') </pre>	<pre> {1} null?(bl') {2} append(bl', reduce(null, λ (e: [Address, list[Byte]], tail: list[Byte]): append(e'2, tail)) (null)) = bl' </pre>
--	---

Expanding the definition of reduce,

Rewriting using append_null, matching in *,

This completes the proof of split_pair_concat.2.1.

split_pair_concat.2.2:

<pre> {-1} size' ≥ 0 {-2} j' ≥ 0 {-3} ∀ k: k < j' ⊃ (∀ (size: nat, a: Address, bl: list[Byte]): k = length(bl) ⊃ reduce(null, λ (e: [Address, list[Byte]], tail: list[Byte]): append(e'2, tail)) (split(size, a, bl)) = bl) {-4} j' = length(bl') </pre>	<pre> {1} null?(bl') {2} a' + length(bl') ≤ floor(expt(2, size'))(a') + expt(2, size') {3} append(head(bl', expt(2, size') - rem(expt(2, size'))(offset(a'))), reduce(null, λ (e: [Address, list[Byte]], tail: list[Byte]): append(e'2, tail)) (split(size', floor(expt(2, size'))(a') + expt(2, size'), tail(bl', expt(2, size') - rem(expt(2, size'))(offset(a')))))) = bl' </pre>
---	--

Instantiating the top quantifier in -3 with the terms: (length(tail(bl!1, expt(2, size!1) - rem(expt(2, size!1))(offset(a!1))))),

Splitting conjunctions,

we get 2 subgoals:

`split_pair_concat.2.2.1:`

$$\begin{array}{l}
 \{-1\} \quad \forall (\text{size}: \text{nat}, a: \text{Address}, \text{bl}: \text{list}[\text{Byte}]): \\
 \quad \text{length}(\text{tail}(\text{bl}', \text{expt}(2, \text{size}') - \text{rem}(\text{expt}(2, \text{size}'))(\text{offset}(a')))) = \\
 \quad \quad \text{length}(\text{bl}) \\
 \quad \quad \supset \\
 \quad \quad \text{reduce}(\text{null}, \lambda (e: [\text{Address}, \text{list}[\text{Byte}]], \text{tail}: \text{list}[\text{Byte}]): \text{append}(e'2, \text{tail})) \\
 \quad \quad \quad (\text{split}(\text{size}, a, \text{bl})) \\
 \{-2\} \quad \text{size}' = \text{bl} \\
 \{-3\} \quad \text{size}' \geq 0 \\
 \{-4\} \quad j' = \text{length}(\text{bl}') \\
 \hline
 \{1\} \quad \text{null?}(\text{bl}') \\
 \{2\} \quad a' + \text{length}(\text{bl}') \leq \text{floor}(\text{expt}(2, \text{size}'))(a') + \text{expt}(2, \text{size}') \\
 \{3\} \quad \text{append}(\text{head}(\text{bl}', \text{expt}(2, \text{size}') - \text{rem}(\text{expt}(2, \text{size}'))(\text{offset}(a'))), \\
 \quad \quad \text{reduce}(\text{null}, \\
 \quad \quad \quad \lambda (e: [\text{Address}, \text{list}[\text{Byte}]], \text{tail}: \text{list}[\text{Byte}]): \text{append}(e'2, \text{tail})) \\
 \quad \quad \quad (\text{split}(\text{size}', \text{floor}(\text{expt}(2, \text{size}'))(a') + \text{expt}(2, \text{size}'), \\
 \quad \quad \quad \quad \text{tail}(\text{bl}', \\
 \quad \quad \quad \quad \quad \text{expt}(2, \text{size}') - \text{rem}(\text{expt}(2, \text{size}'))(\text{offset}(a'))))) \\
 \quad \quad = \text{bl}'
 \end{array}$$

Instantiating the top quantifier in -1 with the terms: $(\text{size!1 floor}(\text{expt}(2, \text{size!1}))(a!1) + \text{expt}(2, \text{size!1}) \text{tail}(\text{bl!1}, \text{expt}(2, \text{size!1}) - \text{rem}(\text{expt}(2, \text{size!1}))(\text{offset}(a!1))))$,

Replacing using formula -1,

Rewriting using `head_tail`, matching in *,

Expanding the definition of `floor`,

Expanding the definition of `floor`,

Expanding the definition of `+`,

Expanding the definition of `<=`,

which is trivially true.

This completes the proof of `split_pair_concat.2.2.1`.

`split_pair_concat.2.2.2:`

$$\begin{array}{l}
 \{-1\} \quad \text{size}' \geq 0 \\
 \{-2\} \quad j' \geq 0 \\
 \{-3\} \quad j' = \text{length}(\text{bl}') \\
 \hline
 \{1\} \quad \text{length}(\text{tail}(\text{bl}', \text{expt}(2, \text{size}') - \text{rem}(\text{expt}(2, \text{size}'))(\text{offset}(a')))) < j' \\
 \{2\} \quad \text{null?}(\text{bl}') \\
 \{3\} \quad a' + \text{length}(\text{bl}') \leq \text{floor}(\text{expt}(2, \text{size}'))(a') + \text{expt}(2, \text{size}') \\
 \{4\} \quad \text{append}(\text{head}(\text{bl}', \text{expt}(2, \text{size}') - \text{rem}(\text{expt}(2, \text{size}'))(\text{offset}(a'))), \\
 \quad \quad \text{reduce}(\text{null}, \\
 \quad \quad \quad \lambda (e: [\text{Address}, \text{list}[\text{Byte}]], \text{tail}: \text{list}[\text{Byte}]): \text{append}(e'2, \text{tail})) \\
 \quad \quad \quad (\text{split}(\text{size}', \text{floor}(\text{expt}(2, \text{size}'))(a') + \text{expt}(2, \text{size}'), \\
 \quad \quad \quad \quad \text{tail}(\text{bl}', \\
 \quad \quad \quad \quad \quad \text{expt}(2, \text{size}') - \text{rem}(\text{expt}(2, \text{size}'))(\text{offset}(a'))))) \\
 \quad \quad = \text{bl}'
 \end{array}$$

Rewriting using `length_tail`, matching in *,

we get 2 subgoals:

split_pair_concat.2.2.2.1:

<pre>{-1} size' ≥ 0 {-2} j' ≥ 0 {-3} j' = length(bl')</pre>	<pre>{1} rem(expt(2, size'))(offset(a')) + length[Byte](bl') - expt(2, size') < j' {2} null?(bl') {3} a' + length(bl') ≤ floor(expt(2, size'))(a') + expt(2, size') {4} append(head(bl', expt(2, size') - rem(expt(2, size'))(offset(a'))), reduce(null, λ (e: [Address, list[Byte]], tail: list[Byte]): append(e'2, tail) (split(size', floor(expt(2, size'))(a') + expt(2, size'), tail(bl', expt(2, size') - rem(expt(2, size'))(offset(a')))))))) = bl'</pre>
---	--

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of split_pair_concat.2.2.2.1.

split_pair_concat.2.2.2.2:

<pre>{-1} size' ≥ 0 {-2} j' ≥ 0 {-3} j' = length(bl')</pre>	<pre>{1} expt(2, size') - rem(expt(2, size'))(offset(a')) ≤ length[Byte](bl') {2} length(tail(bl', expt(2, size') - rem(expt(2, size'))(offset(a')))) < j' {3} null?(bl') {4} a' + length(bl') ≤ floor(expt(2, size'))(a') + expt(2, size') {5} append(head(bl', expt(2, size') - rem(expt(2, size'))(offset(a'))), reduce(null, λ (e: [Address, list[Byte]], tail: list[Byte]): append(e'2, tail) (split(size', floor(expt(2, size'))(a') + expt(2, size'), tail(bl', expt(2, size') - rem(expt(2, size'))(offset(a')))))))) = bl'</pre>
---	--

Hiding formulas: 2, 5,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of split_pair_concat.2.2.2.2.

Q.E.D.

C.118.5 List_Split.split_range

Terse proof for split_range.

split_range:

<pre>{1} ∃ (size: nat, a: Address, bl: list[Byte], n: nat): LET e2size = expt(2, size) IN n = length(bl) ⊃ every(λ (a2: Address, bl2: list[Byte]): a ≤ a2 ∧ a2 + length(bl2) ≤ a + n (split(size, a, bl))</pre>	
---	--

Inducting on n on formula 1 using induction scheme NAT_induction,

we get 2 subgoals:

C Proof scripts

`split_range.1:`

$\{-1\} \quad \forall (\text{size}: \text{nat}, a: \text{Address}, \text{bl}: \text{list}[\text{Byte}]):$ $n' = \text{length}(\text{bl}) \supset$ $\text{every}(\lambda (a_2: \text{Address}, \text{bl}_2: \text{list}[\text{Byte}]):$ $a \leq a_2 \wedge a_2 + \text{length}(\text{bl}_2) \leq a + n')$ $(\text{split}(\text{size}, a, \text{bl}))$	<hr style="border: 0.5px solid black;"/> $\{1\} \quad \forall (\text{size}: \text{nat}, a: \text{Address}, \text{bl}: \text{list}[\text{Byte}]):$ $\text{LET } e2\text{size} = \text{expt}(2, \text{size}) \text{ IN}$ $n' = \text{length}(\text{bl}) \supset$ $\text{every}(\lambda (a_2: \text{Address}, \text{bl}_2: \text{list}[\text{Byte}]):$ $a \leq a_2 \wedge a_2 + \text{length}(\text{bl}_2) \leq a + n')$ $(\text{split}(\text{size}, a, \text{bl}))$
--	---

Repeatedly Skolemizing and flattening,

Simplifying, rewriting, and recording with decision procedures,

Instantiating the top quantifier in -3 with the terms: (size!1 a!1 bl!1),

which is trivially true.

This completes the proof of `split_range.1`.

`split_range.2:`

$\{1\} \quad \forall j:$ $(\forall k:$ $k < j \supset$ $(\forall (\text{size}: \text{nat}, a: \text{Address}, \text{bl}: \text{list}[\text{Byte}]):$ $k = \text{length}(\text{bl}) \supset$ $\text{every}(\lambda (a_2: \text{Address}, \text{bl}_2: \text{list}[\text{Byte}]):$ $a \leq a_2 \wedge a_2 + \text{length}(\text{bl}_2) \leq a + k)$ $(\text{split}(\text{size}, a, \text{bl}))))$ \supset $(\forall (\text{size}: \text{nat}, a: \text{Address}, \text{bl}: \text{list}[\text{Byte}]):$ $j = \text{length}(\text{bl}) \supset$ $\text{every}(\lambda (a_2: \text{Address}, \text{bl}_2: \text{list}[\text{Byte}]):$ $a \leq a_2 \wedge a_2 + \text{length}(\text{bl}_2) \leq a + j)$ $(\text{split}(\text{size}, a, \text{bl}))))$	<hr style="border: 0.5px solid black;"/>
--	--

Repeatedly Skolemizing and flattening,

Hiding formulas: -2,

Expanding the definition of every,

Expanding the definition of split,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 6 subgoals:

split_range.2.1:

{-1}	$size' \geq 0$
{-2}	$j' \geq 0$
{-3}	$\forall k:$
	$k < j' \supset$
	$(\forall (size: nat, a: Address, bl: list[Byte]):$
	$k = length(bl) \supset$
	$every(\lambda (a_2: Address, bl2: list[Byte]):$
	$a \leq a_2 \wedge a_2 + length(bl2) \leq a + k$
	$(split(size, a, bl)))$
{-4}	$j' = length(bl')$
{-5}	$a' + length(bl') \leq floor(expt(2, size'))(a') + expt(2, size')$
{1}	$null?(bl')$
{2}	$every(\lambda (a_2: Address, bl2: list[Byte]):$
	$a' \leq a_2 \wedge a_2 + length(bl2) \leq a' + j')$
	$(null)$

Expanding the definition of every,
which is trivially true.

This completes the proof of **split_range.2.1**.

split_range.2.2:

{-1}	$size' \geq 0$
{-2}	$j' \geq 0$
{-3}	$\forall k:$
	$k < j' \supset$
	$(\forall (size: nat, a: Address, bl: list[Byte]):$
	$k = length(bl) \supset$
	$every(\lambda (a_2: Address, bl2: list[Byte]):$
	$a \leq a_2 \wedge a_2 + length(bl2) \leq a + k$
	$(split(size, a, bl)))$
{-4}	$j' = length(bl')$
{-5}	$a' + length(bl') \leq floor(expt(2, size'))(a') + expt(2, size')$
{1}	$null?(bl')$
{2}	$a' + length(bl') \leq a' + j'$

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of **split_range.2.2**.

split_range.2.3:

{-1}	$size' \geq 0$
{-2}	$j' \geq 0$
{-3}	$\forall k:$
	$k < j' \supset$
	$(\forall (size: nat, a: Address, bl: list[Byte]):$
	$k = length(bl) \supset$
	$every(\lambda (a_2: Address, bl2: list[Byte]):$
	$a \leq a_2 \wedge a_2 + length(bl2) \leq a + k$
	$(split(size, a, bl)))$
{-4}	$j' = length(bl')$
{-5}	$a' + length(bl') \leq floor(expt(2, size'))(a') + expt(2, size')$
{1}	$null?(bl')$
{2}	$a' \leq a'$

C Proof scripts

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `split_range.2.3`.

`split_range.2.4`:

<pre> {-1} size' ≥ 0 {-2} j' ≥ 0 {-3} ∀ k: k < j' ⊃ (∀ (size: nat, a: Address, bl: list[Byte]): k = length(bl) ⊃ every(λ (a₂: Address, bl₂: list[Byte]): a ≤ a₂ ∧ a₂ + length(bl₂) ≤ a + k) (split(size, a, bl))) {-4} j' = length(bl')</pre>	<hr style="border: 0.5px solid black;"/> <pre> {1} null?(bl') {2} a' + length(bl') ≤ floor(expt(2, size'))(a') + expt(2, size') {3} every(λ (a₂: Address, bl₂: list[Byte]): a' ≤ a₂ ∧ a₂ + length(bl₂) ≤ a' + j') (split(size', floor(expt(2, size'))(a') + expt(2, size'), tail(bl', expt(2, size') - rem(expt(2, size'))(offset(a')))))</pre>
--	---

Instantiating the top quantifier in -3 with the terms: $(\text{length}(\text{tail}(\text{bl}!1, \text{expt}(2, \text{size}!1) - \text{rem}(\text{expt}(2, \text{size}!1))(\text{offset}(a!1))))$,

Splitting conjunctions,

we get 2 subgoals:

`split_range.2.4.1`:

<pre> {-1} ∀ (size: nat, a: Address, bl: list[Byte]): length(tail(bl', expt(2, size') - rem(expt(2, size'))(offset(a')))) = length(bl) ⊃ every(λ (a₂: Address, bl₂: list[Byte]): a ≤ a₂ ∧ a₂ + length(bl₂) ≤ a + length(tail(bl', expt(2, size') - rem(expt(2, size'))(offset(a'))))) (split(size, a, bl)) {-2} size' ≥ 0 {-3} j' ≥ 0 {-4} j' = length(bl')</pre>	<hr style="border: 0.5px solid black;"/> <pre> {1} null?(bl') {2} a' + length(bl') ≤ floor(expt(2, size'))(a') + expt(2, size') {3} every(λ (a₂: Address, bl₂: list[Byte]): a' ≤ a₂ ∧ a₂ + length(bl₂) ≤ a' + j') (split(size', floor(expt(2, size'))(a') + expt(2, size'), tail(bl', expt(2, size') - rem(expt(2, size'))(offset(a')))))</pre>
---	---

Instantiating the top quantifier in -1 with the terms: $(\text{size}!1 \text{ floor}(\text{expt}(2, \text{size}!1))(\text{a}!1) + \text{expt}(2, \text{size}!1) \text{ tail}(\text{bl}!1, \text{expt}(2, \text{size}!1) - \text{rem}(\text{expt}(2, \text{size}!1))(\text{offset}(\text{a}!1))))$,

Using lemma `every_implied`,

Simplifying, rewriting, and recording with decision procedures,

Hiding formulas: (-1 4),

Repeatedly Skolemizing and flattening,

Rewriting using `length_tail`, matching in *,

we get 2 subgoals:

split_range.2.4.1.1:

{-1}	$\text{floor}(\text{expt}(2, \text{size}'))(a') + \text{expt}(2, \text{size}') \leq t'^1$	
{-2}	$t'^1 + \text{length}(t'^2) \leq$ $\text{floor}(\text{expt}(2, \text{size}'))(a') + \text{expt}(2, \text{size}') + \text{rem}(\text{expt}(2, \text{size}'))(\text{offset}(a')) + \text{length}$	[Byte](bl') -
{-3}	$\text{size}' \geq 0$	
{-4}	$j' \geq 0$	
{-5}	$j' = \text{length}(\text{bl}')$	
{1}	$a' \leq t'^1 \wedge t'^1 + \text{length}(t'^2) \leq a' + j'$	
{2}	$\text{null?}(\text{bl}')$	
{3}	$a' + \text{length}(\text{bl}') \leq \text{floor}(\text{expt}(2, \text{size}'))(a') + \text{expt}(2, \text{size}')$	

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of split_range.2.4.1.1.

split_range.2.4.1.2:

{-1}	$\text{floor}(\text{expt}(2, \text{size}'))(a') + \text{expt}(2, \text{size}') \leq t'^1$	
{-2}	$t'^1 + \text{length}(t'^2) \leq$ $\text{floor}(\text{expt}(2, \text{size}'))(a') + \text{expt}(2, \text{size}') + \text{length}(\text{tail}(\text{bl}', \text{expt}(2, \text{size}') - \text{rem}(\text{expt}(2, \text{size}'))(\text{offset}(a'))))$	
{-3}	$\text{size}' \geq 0$	
{-4}	$j' \geq 0$	
{-5}	$j' = \text{length}(\text{bl}')$	
{1}	$\text{expt}(2, \text{size}') - \text{rem}(\text{expt}(2, \text{size}'))(\text{offset}(a')) \leq \text{length}[\text{Byte}](\text{bl}')$	
{2}	$a' \leq t'^1 \wedge t'^1 + \text{length}(t'^2) \leq a' + j'$	
{3}	$\text{null?}(\text{bl}')$	
{4}	$a' + \text{length}(\text{bl}') \leq \text{floor}(\text{expt}(2, \text{size}'))(a') + \text{expt}(2, \text{size}')$	

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of split_range.2.4.1.2.

split_range.2.4.2:

{-1}	$\text{size}' \geq 0$	
{-2}	$j' \geq 0$	
{-3}	$j' = \text{length}(\text{bl}')$	
{1}	$\text{length}(\text{tail}(\text{bl}', \text{expt}(2, \text{size}') - \text{rem}(\text{expt}(2, \text{size}'))(\text{offset}(a')))) < j'$	
{2}	$\text{null?}(\text{bl}')$	
{3}	$a' + \text{length}(\text{bl}') \leq \text{floor}(\text{expt}(2, \text{size}'))(a') + \text{expt}(2, \text{size}')$	
{4}	$\text{every}(\lambda (a_2: \text{Address}, \text{bl2}: \text{list}[\text{Byte}]):$ $\quad a' \leq a_2 \wedge a_2 + \text{length}(\text{bl2}) \leq a' + j'$ $\quad (\text{split}(\text{size}', \text{floor}(\text{expt}(2, \text{size}'))(a') + \text{expt}(2, \text{size}'),$ $\quad \quad \text{tail}(\text{bl}', \text{expt}(2, \text{size}') - \text{rem}(\text{expt}(2, \text{size}'))(\text{offset}(a')))))$	

Rewriting using length_tail, matching in *,

we get 2 subgoals:

C Proof scripts

`split_range.2.4.2.1:`

{-1}	$size' \geq 0$
{-2}	$j' \geq 0$
{-3}	$j' = \text{length}(bl')$
{1}	$\text{rem}(\text{expt}(2, size'))(\text{offset}(a')) + \text{length}[Byte](bl') - \text{expt}(2, size') < j'$
{2}	$\text{null?}(bl')$
{3}	$a' + \text{length}(bl') \leq \text{floor}(\text{expt}(2, size'))(a') + \text{expt}(2, size')$
{4}	$\text{every}(\lambda (a_2: \text{Address}, bl2: \text{list}[Byte]):$ $\quad a' \leq a_2 \wedge a_2 + \text{length}(bl2) \leq a' + j'$ $\quad (\text{split}(size', \text{floor}(\text{expt}(2, size'))(a') + \text{expt}(2, size'),$ $\quad \quad \text{tail}(bl', \text{expt}(2, size') - \text{rem}(\text{expt}(2, size'))(\text{offset}(a')))))$

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `split_range.2.4.2.1`.

`split_range.2.4.2.2:`

{-1}	$size' \geq 0$
{-2}	$j' \geq 0$
{-3}	$j' = \text{length}(bl')$
{1}	$\text{expt}(2, size') - \text{rem}(\text{expt}(2, size'))(\text{offset}(a')) \leq \text{length}[Byte](bl')$
{2}	$\text{length}(\text{tail}(bl', \text{expt}(2, size') - \text{rem}(\text{expt}(2, size'))(\text{offset}(a')))) < j'$
{3}	$\text{null?}(bl')$
{4}	$a' + \text{length}(bl') \leq \text{floor}(\text{expt}(2, size'))(a') + \text{expt}(2, size')$
{5}	$\text{every}(\lambda (a_2: \text{Address}, bl2: \text{list}[Byte]):$ $\quad a' \leq a_2 \wedge a_2 + \text{length}(bl2) \leq a' + j'$ $\quad (\text{split}(size', \text{floor}(\text{expt}(2, size'))(a') + \text{expt}(2, size'),$ $\quad \quad \text{tail}(bl', \text{expt}(2, size') - \text{rem}(\text{expt}(2, size'))(\text{offset}(a')))))$

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `split_range.2.4.2.2`.

`split_range.2.5:`

{-1}	$size' \geq 0$
{-2}	$j' \geq 0$
{-3}	$\forall k:$ $\quad k < j' \supset$ $\quad (\forall (size: \text{nat}, a: \text{Address}, bl: \text{list}[Byte]):$ $\quad \quad k = \text{length}(bl) \supset$ $\quad \quad \text{every}(\lambda (a_2: \text{Address}, bl2: \text{list}[Byte]):$ $\quad \quad \quad a \leq a_2 \wedge a_2 + \text{length}(bl2) \leq a + k)$ $\quad \quad (\text{split}(size, a, bl)))$
{-4}	$j' = \text{length}(bl')$
{1}	$\text{null?}(bl')$
{2}	$a' + \text{length}(bl') \leq \text{floor}(\text{expt}(2, size'))(a') + \text{expt}(2, size')$
{3}	$a' + \text{length}(\text{head}(bl', \text{expt}(2, size') - \text{rem}(\text{expt}(2, size'))(\text{offset}(a'))))$ $\quad \leq a' + j'$

Rewriting using `length_head`, matching in `*`,

we get 2 subgoals:

`split_range.2.5.1:`

{-1}	$size' \geq 0$
{-2}	$j' \geq 0$
{-3}	$\forall k:$ $k < j' \supset$ $(\forall (size: nat, a: Address, bl: list[Byte]):$ $k = length(bl) \supset$ $every(\lambda (a_2: Address, bl2: list[Byte]):$ $a \leq a_2 \wedge a_2 + length(bl2) \leq a + k$ $(split(size, a, bl)))$
{-4}	$j' = length(bl')$
{1}	$null?(bl')$
{2}	$a' + length(bl') \leq floor(expt(2, size'))(a') + expt(2, size')$
{3}	$a' + expt(2, size') - rem(expt(2, size'))(offset(a')) \leq a' + j'$

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `split_range.2.5.1`.

`split_range.2.5.2:`

{-1}	$size' \geq 0$
{-2}	$j' \geq 0$
{-3}	$\forall k:$ $k < j' \supset$ $(\forall (size: nat, a: Address, bl: list[Byte]):$ $k = length(bl) \supset$ $every(\lambda (a_2: Address, bl2: list[Byte]):$ $a \leq a_2 \wedge a_2 + length(bl2) \leq a + k$ $(split(size, a, bl)))$
{-4}	$j' = length(bl')$
{1}	$expt(2, size') - rem(expt(2, size'))(offset(a')) \leq length[Byte](bl')$
{2}	$null?(bl')$
{3}	$a' + length(bl') \leq floor(expt(2, size'))(a') + expt(2, size')$
{4}	$a' + length(head(bl', expt(2, size') - rem(expt(2, size'))(offset(a'))))$ $\leq a' + j'$

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `split_range.2.5.2`.

`split_range.2.6:`

{-1}	$size' \geq 0$
{-2}	$j' \geq 0$
{-3}	$\forall k:$ $k < j' \supset$ $(\forall (size: nat, a: Address, bl: list[Byte]):$ $k = length(bl) \supset$ $every(\lambda (a_2: Address, bl2: list[Byte]):$ $a \leq a_2 \wedge a_2 + length(bl2) \leq a + k$ $(split(size, a, bl)))$
{-4}	$j' = length(bl')$
{1}	$null?(bl')$
{2}	$a' + length(bl') \leq floor(expt(2, size'))(a') + expt(2, size')$
{3}	$a' \leq a'$

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `split_range.2.6`.

Q.E.D.

C.118.6 List_Split.split_no_null

Terse proof for `split_no_null`.

`split_no_null`:

$\{1\} \quad \forall (\text{size}: \text{nat}, a: \text{Address}, \text{bl}: \text{list}[\text{Byte}], n: \text{nat}):$ $\text{LET } e2\text{size} = \text{expt}(2, \text{size}) \text{ IN}$ $n = \text{length}(\text{bl}) \wedge \text{cons}^?(\text{bl}) \supset$ $\text{every}(\lambda (a_2: \text{Address}, \text{bl}_2: \text{list}[\text{Byte}]): \text{cons}^?(\text{bl}_2))(\text{split}(\text{size}, a, \text{bl}))$

Inducting on n on formula 1 using induction scheme `NAT_induction`,

we get 2 subgoals:

`split_no_null.1`:

$\{-1\} \quad \forall (\text{size}: \text{nat}, a: \text{Address}, \text{bl}: \text{list}[\text{Byte}]):$ $n' = \text{length}(\text{bl}) \wedge \text{cons}^?(\text{bl}) \supset$ $\text{every}(\lambda (a_2: \text{Address}, \text{bl}_2: \text{list}[\text{Byte}]): \text{cons}^?(\text{bl}_2))(\text{split}(\text{size}, a, \text{bl}))$
$\{1\} \quad \forall (\text{size}: \text{nat}, a: \text{Address}, \text{bl}: \text{list}[\text{Byte}]):$ $\text{LET } e2\text{size} = \text{expt}(2, \text{size}) \text{ IN}$ $n' = \text{length}(\text{bl}) \wedge \text{cons}^?(\text{bl}) \supset$ $\text{every}(\lambda (a_2: \text{Address}, \text{bl}_2: \text{list}[\text{Byte}]): \text{cons}^?(\text{bl}_2))(\text{split}(\text{size}, a, \text{bl}))$

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `split_no_null.1`.

`split_no_null.2`:

$\{1\} \quad \forall j:$ $(\forall k:$ $k < j \supset$ $(\forall (\text{size}: \text{nat}, a: \text{Address}, \text{bl}: \text{list}[\text{Byte}]):$ $k = \text{length}(\text{bl}) \wedge \text{cons}^?(\text{bl}) \supset$ $\text{every}(\lambda (a_2: \text{Address}, \text{bl}_2: \text{list}[\text{Byte}]): \text{cons}^?(\text{bl}_2))$ $(\text{split}(\text{size}, a, \text{bl}))))$ \supset $(\forall (\text{size}: \text{nat}, a: \text{Address}, \text{bl}: \text{list}[\text{Byte}]):$ $j = \text{length}(\text{bl}) \wedge \text{cons}^?(\text{bl}) \supset$ $\text{every}(\lambda (a_2: \text{Address}, \text{bl}_2: \text{list}[\text{Byte}]): \text{cons}^?(\text{bl}_2))(\text{split}(\text{size}, a, \text{bl})))$
--

Repeatedly Skolemizing and flattening,

Hiding formulas: -2,

Expanding the definition of `every`,

Expanding the definition of `split`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 3 subgoals:

split_no_null.2.1:

{-1}	$size' \geq 0$
{-2}	$j' \geq 0$
{-3}	$\forall k:$ $k < j' \supset$ $(\forall (size: nat, a: Address, bl: list[Byte]):$ $k = length(bl) \wedge cons?(bl) \supset$ $every(\lambda (a_2: Address, bl2: list[Byte]): cons?(bl2))(split(size, a, bl)))$
{-4}	$j' = length(bl')$
{-5}	$cons?(bl')$
{-6}	$a' + length(bl') \leq floor(expt(2, size'))(a') + expt(2, size')$
{1}	$every(\lambda (a_2: Address, bl2: list[Byte]): cons?(bl2))(null)$

Expanding the definition of every,

which is trivially true.

This completes the proof of split_no_null.2.1.

split_no_null.2.2:

{-1}	$size' \geq 0$
{-2}	$j' \geq 0$
{-3}	$\forall k:$ $k < j' \supset$ $(\forall (size: nat, a: Address, bl: list[Byte]):$ $k = length(bl) \wedge cons?(bl) \supset$ $every(\lambda (a_2: Address, bl2: list[Byte]): cons?(bl2))(split(size, a, bl)))$
{-4}	$j' = length(bl')$
{-5}	$cons?(bl')$
{1}	$a' + length(bl') \leq floor(expt(2, size'))(a') + expt(2, size')$
{2}	$every(\lambda (a_2: Address, bl2: list[Byte]): cons?(bl2))$ $(split(size', floor(expt(2, size'))(a') + expt(2, size'),$ $tail(bl', expt(2, size') - rem(expt(2, size'))(offset(a'))))$

Instantiating the top quantifier in -3 with the terms: $(length(tail(bl!1, expt(2, size!1) - rem(expt(2, size!1))(offset(a!1))))),$

Splitting conjunctions,

we get 2 subgoals:

split_no_null.2.2.1:

{-1}	$\forall (size: nat, a: Address, bl: list[Byte]):$ $length(tail(bl', expt(2, size') - rem(expt(2, size'))(offset(a')))) =$ $length(bl)$ $\wedge cons?(bl)$ $\supset every(\lambda (a_2: Address, bl2: list[Byte]): cons?(bl2))(split(size, a, bl))$
{-2}	$size' \geq 0$
{-3}	$j' \geq 0$
{-4}	$j' = length(bl')$
{-5}	$cons?(bl')$
{1}	$a' + length(bl') \leq floor(expt(2, size'))(a') + expt(2, size')$
{2}	$every(\lambda (a_2: Address, bl2: list[Byte]): cons?(bl2))$ $(split(size', floor(expt(2, size'))(a') + expt(2, size'),$ $tail(bl', expt(2, size') - rem(expt(2, size'))(offset(a'))))$

Instantiating the top quantifier in -1 with the terms: $(size!1 floor(expt(2, size!1))(a!1) + expt(2, size!1) tail(bl!1, expt(2, size!1) - rem(expt(2, size!1))(offset(a!1))))),$

C Proof scripts

Simplifying, rewriting, and recording with decision procedures,
Using lemma `length_tail`,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `split_no_null.2.2.1`.

`split_no_null.2.2.2`:

{-1}	$size' \geq 0$
{-2}	$j' \geq 0$
{-3}	$j' = \text{length}(bl')$
{-4}	$\text{cons?}(bl')$
{1}	$\text{length}(\text{tail}(bl', \text{expt}(2, size') - \text{rem}(\text{expt}(2, size'))(\text{offset}(a')))) < j'$
{2}	$a' + \text{length}(bl') \leq \text{floor}(\text{expt}(2, size'))(a') + \text{expt}(2, size')$
{3}	$\text{every}(\lambda (a_2: \text{Address}, bl2: \text{list}[\text{Byte}]): \text{cons?}(bl2))$ $(\text{split}(size', \text{floor}(\text{expt}(2, size'))(a') + \text{expt}(2, size'),$ $\text{tail}(bl', \text{expt}(2, size') - \text{rem}(\text{expt}(2, size'))(\text{offset}(a'))))$

Hiding formulas: 3,

Using lemma `length_tail`,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `split_no_null.2.2.2`.

`split_no_null.2.3`:

{-1}	$size' \geq 0$
{-2}	$j' \geq 0$
{-3}	$\forall k:$ $k < j' \supset$ $(\forall (size: \text{nat}, a: \text{Address}, bl: \text{list}[\text{Byte}]):$ $k = \text{length}(bl) \wedge \text{cons?}(bl) \supset$ $\text{every}(\lambda (a_2: \text{Address}, bl2: \text{list}[\text{Byte}]): \text{cons?}(bl2))(\text{split}(size, a, bl)))$
{-4}	$j' = \text{length}(bl')$
{-5}	$\text{cons?}(bl')$
{1}	$a' + \text{length}(bl') \leq \text{floor}(\text{expt}(2, size'))(a') + \text{expt}(2, size')$
{2}	$\text{cons?}(\text{head}(bl', \text{expt}(2, size') - \text{rem}(\text{expt}(2, size'))(\text{offset}(a'))))$

Expanding the definition of `head`,

which is trivially true.

This completes the proof of `split_no_null.2.3`.

Q.E.D.

C.118.7 List_Split.split_null

Terse proof for `split_null`.

`split_null`:

{1}	$\forall (size: \text{nat}, a: \text{Address}): \text{split}(size, a, \text{null}) = \text{null}$
-----	---

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `split_null`.

Q.E.D.

C.118.8 List_Split.split_type_TCC1

Terse proof for `split_type_TCC1`.

split_type_TCC1:

{1} Mem?(min_linear 'type_of)

Expanding the definition of min_linear,

Expanding the definition of Mem,

which is trivially true.

This completes the proof of split_type_TCC1.

Q.E.D.

C.118.9 List_Split.split_type_TCC2

Terse proof for split_type_TCC2.

split_type_TCC2:

{1} $\forall (a: \text{Address}): \text{reg_base}(\text{min_linear})(\text{type_of}(a)) \leq a \supset \text{Mem}?(\text{max_linear} ' \text{type_of})$

Expanding the definition of max_linear,

Expanding the definition of Mem,

which is trivially true.

This completes the proof of split_type_TCC2.

Q.E.D.

C.118.10 List_Split.split_type

Terse proof for split_type.

split_type:

{1} $\forall (\text{size}: \text{nat}, a: \text{Address}, \text{bl}: \text{list}[\text{Byte}], n: \text{nat}):$
 $\text{reg_base}(\text{min_linear})(\text{type_of}(a)) \leq a \wedge$
 $a + \text{length}(\text{bl}) \leq \text{reg_size}(\text{max_linear})(\text{type_of}(a))$
 \supset
 $\text{every}(\lambda (t: [\text{Address}, \text{list}[\text{Byte}]]):$
 $\text{type_of}(t'1) = \text{type_of}(a) \wedge$
 $\text{reg_base}(\text{min_linear})(\text{type_of}(a)) \leq t'1 \wedge$
 $t'1 < \text{reg_size}(\text{max_linear})(\text{type_of}(a)))$
 $(\text{split}(\text{min_page}, a, \text{bl}))$

Repeatedly Skolemizing and flattening,

Hiding formulas: -2,

Case splitting on bl!1 = null,

we get 2 subgoals:

split_type.1:

{-1}	$bl' = \text{null}$
{-2}	$\text{size}' \geq 0$
{-3}	$n' \geq 0$
{-4}	$\text{reg_base}(\text{min_linear})(\text{type_of}(a')) \leq a'$
{-5}	$a' + \text{length}(bl') \leq \text{reg_size}(\text{max_linear})(\text{type_of}(a'))$
{1} $\text{every}(\lambda (t: [\text{Address}, \text{list}[\text{Byte}]]):$ $\text{type_of}(t'1) = \text{type_of}(a') \wedge$ $\text{reg_base}(\text{min_linear})(\text{type_of}(a')) \leq t'1 \wedge$ $t'1 < \text{reg_size}(\text{max_linear})(\text{type_of}(a'))$ $(\text{split}(\text{min_page}, a', bl'))$	

Using lemma `split_null`,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `split_type.1`.

split_type.2:

{-1}	$\text{size}' \geq 0$
{-2}	$n' \geq 0$
{-3}	$\text{reg_base}(\text{min_linear})(\text{type_of}(a')) \leq a'$
{-4}	$a' + \text{length}(bl') \leq \text{reg_size}(\text{max_linear})(\text{type_of}(a'))$
{1} $bl' = \text{null}$ {2} $\text{every}(\lambda (t: [\text{Address}, \text{list}[\text{Byte}]]):$ $\text{type_of}(t'1) = \text{type_of}(a') \wedge$ $\text{reg_base}(\text{min_linear})(\text{type_of}(a')) \leq t'1 \wedge$ $t'1 < \text{reg_size}(\text{max_linear})(\text{type_of}(a'))$ $(\text{split}(\text{min_page}, a', bl'))$	

Using lemma `split_range`,

Using lemma `split_no_null`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Using lemma `every_implied`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 2 subgoals:

split_type.2.1:

{-1}	$\text{cons?}(bl')$
{-2}	$\text{every}(\lambda (a_2: \text{Address}, bl2: \text{list}[\text{Byte}]): \text{cons?}(bl2))(\text{split}(\text{min_page}, a', bl'))$
{-3}	$\text{every}(\lambda (a_2: \text{Address}, bl2: \text{list}[\text{Byte}]):$ $a' \leq a_2 \wedge a_2 + \text{length}(bl2) \leq a' + \text{length}(bl')$ $(\text{split}(\text{min_page}, a', bl'))$
{-4}	$\text{size}' \geq 0$
{-5}	$n' \geq 0$
{-6}	$\text{reg_base}(\text{min_linear})(\text{type_of}(a')) \leq a'$
{-7}	$a' + \text{length}(bl') \leq \text{reg_size}(\text{max_linear})(\text{type_of}(a'))$
{1} $\text{every}(\lambda (a_2: \text{Address}, bl2: \text{list}[\text{Byte}]):$ $\text{cons?}(bl2) \wedge a' \leq a_2 \wedge a_2 + \text{length}(bl2) \leq a' + \text{length}(bl')$ $(\text{split}(\text{min_page}, a', bl'))$ {2} $\text{every}(\lambda (t: [\text{Address}, \text{list}[\text{Byte}]]):$ $\text{type_of}(t'1) = \text{type_of}(a') \wedge$ $\text{reg_base}(\text{min_linear})(\text{type_of}(a')) \leq t'1 \wedge$ $t'1 < \text{reg_size}(\text{max_linear})(\text{type_of}(a'))$ $(\text{split}(\text{min_page}, a', bl'))$	

Using lemma `every_conjunct_left`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `split_type.2.1`.

`split_type.2.2`:

{-1}	<code>cons?(bl')</code>
{-2}	<code>every(λ (a₂: Address, bl2: list[Byte]): cons?(bl2))(split(min_page, a', bl'))</code>
{-3}	<code>every(λ (a₂: Address, bl2: list[Byte]): a' ≤ a₂ ∧ a₂ + length(bl2) ≤ a' + length(bl')) (split(min_page, a', bl'))</code>
{-4}	<code>size' ≥ 0</code>
{-5}	<code>n' ≥ 0</code>
{-6}	<code>reg_base(min_linear)(type_of(a')) ≤ a'</code>
{-7}	<code>a' + length(bl') ≤ reg_size(max_linear)(type_of(a'))</code>
{1}	<code>∀ (t₁: [Address, list[Byte]]): cons?(t₁'2) ∧ a' ≤ t₁'1 ∧ t₁'1 + length(t₁'2) ≤ a' + length(bl') ⊃ type_of(t₁'1) = type_of(a') ∧ reg_base(min_linear)(type_of(a')) ≤ t₁'1 ∧ t₁'1 < reg_size(max_linear)(type_of(a'))</code>
{2}	<code>every(λ (t: [Address, list[Byte]]): type_of(t'1) = type_of(a') ∧ reg_base(min_linear)(type_of(a')) ≤ t'1 ∧ t'1 < reg_size(max_linear)(type_of(a')) (split(min_page, a', bl'))</code>

Hiding formulas: (-2 -3 2),

Repeatedly Skolemizing and flattening,

Expanding the definition of `<=`,

Expanding the definition of `<`,

Expanding the definition of `+`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Expanding the definition of `length`,

Keeping (-5 -12 1) and hiding `*`,

Adding type constraints for `length(cdr(t!1'2))`,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `split_type.2.2`.

Q.E.D.

C.119 Proofs for LogicalExpressions (expressions.pvs)

This theory contains no provable formal statements.

C.120 Proofs for LvalueToRvalueConversion (conversions.pvs)

This theory contains no provable formal statements.

C.121 Proofs for Memory (memory.pvs)

C.121.1 Memory.memory_write_list_nse_TCC1

Terse proof for memory_write_list_nse_TCC1.

memory_write_list_nse_TCC1:

$$\frac{}{\{1\} \quad \forall (bl: \text{list}[\text{Byte}], b: \text{Byte}, rl: \text{list}[\text{Byte}]): bl = \text{cons}(b, rl) \supset \text{cons}^?[\text{Byte}](bl)}$$

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of memory_write_list_nse_TCC1.

Q.E.D.

C.121.2 Memory.memory_write_list_nse_TCC2

Terse proof for memory_write_list_nse_TCC2.

memory_write_list_nse_TCC2:

$$\frac{}{\{1\} \quad \forall (bl: \text{list}[\text{Byte}], b: \text{Byte}, rl: \text{list}[\text{Byte}]): \\ bl = \text{cons}(b, rl) \supset \text{length}[\text{Byte}](\text{cdr}[\text{Byte}](bl)) < \text{length}[\text{Byte}](bl)}$$

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of memory_write_list_nse_TCC2.

Q.E.D.

C.121.3 Memory.memory_read_list_nse_TCC1

Terse proof for memory_read_list_nse_TCC1.

memory_read_list_nse_TCC1:

$$\frac{}{\{1\} \quad \forall (\text{size}: \text{nat}): \text{size} > 0 \supset \text{size} - 1 \geq 0}$$

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of memory_read_list_nse_TCC1.

Q.E.D.

C.121.4 Memory.memory_read_list_nse_TCC2

Terse proof for memory_read_list_nse_TCC2.

memory_read_list_nse_TCC2:

$$\frac{}{\{1\} \quad \forall (\text{size}: \text{nat}): \text{size} > 0 \supset \text{size} - 1 < \text{size}}$$

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of memory_read_list_nse_TCC2.

Q.E.D.

C.121.5 Memory.memory_read_list_ok_length

Terse proof for memory_read_list_ok_length.

memory_read_list_ok_length:

{1} \forall (addr: Address, pm: Memory_struct, s: State, size: nat):
 OK?(memory_read_list_nse(pm)(addr, size)(s)) \supset
 length(data(memory_read_list_nse(pm)(addr, size)(s))) = size

Inducting on size on formula 1,

we get 2 subgoals:

memory_read_list_ok_length.1:

{1} \forall (addr: Address, pm: Memory_struct, s: State):
 OK?(memory_read_list_nse(pm)(addr, 0)(s)) \supset
 length(data(memory_read_list_nse(pm)(addr, 0)(s))) = 0

Repeatedly Skolemizing and flattening,

Expanding the definition of memory_read_list_nse,

Expanding the definition of ok_result,

Expanding the definition of length,

which is trivially true.

This completes the proof of memory_read_list_ok_length.1.

memory_read_list_ok_length.2:

{1} $\forall j$:
 $(\forall$ (addr: Address, pm: Memory_struct, s: State):
 OK?(memory_read_list_nse(pm)(addr, j)(s)) \supset
 length(data(memory_read_list_nse(pm)(addr, j)(s))) = j)
 \supset
 $(\forall$ (addr: Address, pm: Memory_struct, s: State):
 OK?(memory_read_list_nse(pm)(addr, j + 1)(s)) \supset
 length(data(memory_read_list_nse(pm)(addr, j + 1)(s))) = j + 1)

Repeatedly Skolemizing and flattening,

Expanding the definition of memory_read_list_nse,

Expanding the definition of ##,

Expanding the definition of ok_result,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Expanding the definition of length,

Instantiating quantified variables,

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of memory_read_list_ok_length.2.

Q.E.D.

C.121.6 Memory.memory_read_transformers_memory_read

Terse proof for memory_read_transformers_memory_read.

memory_read_transformers_memory_read:

{1} \forall (addresses: PRED[Address], pm: Memory_struct, addr: Address):
 addresses(addr) \supset
 memory_read_transformers(pm, addresses)(expr_2_super(memory_read(pm)(addr)))

Repeatedly Skolemizing and flattening,

Expanding the definition of `memory_read_transformers`,
 Instantiating quantified variables,
 This completes the proof of `memory_read_transformers_memory_read`.
 Q.E.D.

C.121.7 Memory.memory_read_transformers_mono

Terse proof for `memory_read_transformers_mono`.

`memory_read_transformers_mono`:

{1}	$\forall (pm: \text{Memory_struct}, \text{addresses}_1, \text{addresses}_2: \text{PRED}[\text{Address}]):$ $(\text{addresses}_1 \subseteq \text{addresses}_2) \supset$ $(\text{memory_read_transformers}(pm, \text{addresses}_1) \subseteq \text{memory_read_transformers}(pm, \text{addresses}_2))$
-----	---

Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of `memory_read_transformers_mono`.
 Q.E.D.

C.121.8 Memory.memory_read_transformers_union

Terse proof for `memory_read_transformers_union`.

`memory_read_transformers_union`:

{1}	$\forall (pm: \text{Memory_struct}, \text{addresses}_1, \text{addresses}_2: \text{PRED}[\text{Address}]):$ $(\text{memory_read_transformers}(pm, \text{addresses}_1) \cup \text{memory_read_transformers}(pm, \text{addresses}_2))$ $= \text{memory_read_transformers}(pm, (\text{addresses}_1 \cup \text{addresses}_2))$
-----	--

Repeatedly Skolemizing and flattening,
 Expanding the definition of `memory_read_transformers`,
 Expanding the definition of union,
 Expanding the definition of member,
 Applying decompose-equality,
 Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of `memory_read_transformers_union`.
 Q.E.D.

C.121.9 Memory.memory_write_transformers_memory_write

Terse proof for `memory_write_transformers_memory_write`.

`memory_write_transformers_memory_write`:

{1}	$\forall (\text{addresses}: \text{PRED}[\text{Address}], pm: \text{Memory_struct}, \text{addr}: \text{Address}, b: \text{Byte}):$ $\text{addresses}(\text{addr}) \supset$ $\text{memory_write_transformers}(pm, \text{addresses})(\text{expr}_2\text{-super}(\text{memory_write}(pm)(\text{addr}, b)))$
-----	---

Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of `memory_write_transformers_memory_write`.
 Q.E.D.

C.121.10 Memory.memory_write_transformers_mono

Terse proof for memory_write_transformers_mono.

memory_write_transformers_mono:

$$\{1\} \quad \forall (\text{pm: Memory_struct, addresses}_1, \text{addresses}_2: \text{PRED}[\text{Address}]):$$

$$(\text{addresses}_1 \subseteq \text{addresses}_2) \supset$$

$$(\text{memory_write_transformers}(\text{pm}, \text{addresses}_1) \subseteq \text{memory_write_transformers}(\text{pm}, \text{addresses}_2))$$

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of memory_write_transformers_mono.
Q.E.D.

C.121.11 Mem- memory.memory_read_side_effect_super_transformers_memory_read_side_effect

Terse proof for memory_read_side_effect_super_transformers_memory_read_side_effect.

memory_read_side_effect_super_transformers_memory_read_side_effect:

$$\{1\} \quad \forall (\text{addresses: PRED}[\text{Address}], \text{pm: Memory_struct}, a: \text{Ad-}$$

$$\text{dress, bl: list}[\text{Byte}], \text{cp: bool}):$$

$$(\text{address_block}(a, \text{length}(\text{bl})) \subseteq \text{addresses}) \supset$$

$$\text{memory_read_side_effect_super_transformers}(\text{pm}, \text{addresses})$$

$$(\text{expr}_2\text{-super}(\text{memory_read_side_effect}(\text{pm}$$

$$(a, \text{bl}, \text{cp})))$$

Repeatedly Skolemizing and flattening,
Expanding the definition of memory_read_side_effect_super_transformers,
Instantiating quantified variables,
This completes the proof of memory_read_side_effect_super_transformers_memory_read_side_effect.
Q.E.D.

C.121.12 Memory.memory_read_side_effect_super_transformers_mono

Terse proof for memory_read_side_effect_super_transformers_mono.

memory_read_side_effect_super_transformers_mono:

$$\{1\} \quad \forall (\text{pm: Memory_struct, addresses}_1, \text{addresses}_2: \text{PRED}[\text{Address}]):$$

$$(\text{addresses}_1 \subseteq \text{addresses}_2) \supset$$

$$(\text{memory_read_side_effect_super_transformers}(\text{pm}, \text{addresses}_1) \subseteq \text{memory_read_side_effect_super_transformers}(\text{pm}, \text{addresses}_2))$$

Repeatedly Skolemizing and flattening,
Expanding the definition of subset?,
Expanding the definition of member,
Expanding the definition of memory_read_side_effect_super_transformers,
Repeatedly Skolemizing and flattening,
Instantiating quantified variables,
Simplifying, rewriting, and recording with decision procedures,
Forward chaining on subset_transitive,
This completes the proof of memory_read_side_effect_super_transformers_mono.

Q.E.D.

C.121.13 Memory.memory_write_side_effect_super_transformers_memory_write_side_effect

Terse proof for memory_write_side_effect_super_transformers_memory_write_side_effect.

memory_write_side_effect_super_transformers_memory_write_side_effect:

{1}	$\forall (\text{addresses: PRED}[\text{Address}], \text{pm: Memory_struct}, a: \text{Address}, \text{bl: list}[\text{Byte}], \text{cp: bool}):$ $(\text{address_block}(a, \text{length}(\text{bl})) \subseteq \text{addresses}) \supset$ $\text{memory_write_side_effect_super_transformers}(\text{pm}, \text{addresses})$ $= (\text{expr_2_super}(\text{memory_write_side_effect}(\text{pm}, a, \text{bl}, \text{cp})))$
-----	---

Repeatedly Skolemizing and flattening,

Expanding the definition of memory_write_side_effect_super_transformers,

Instantiating quantified variables,

This completes the proof of memory_write_side_effect_super_transformers_memory_write_side_effect.

Q.E.D.

C.121.14 Memory.memory_write_side_effect_super_transformers_mono

Terse proof for memory_write_side_effect_super_transformers_mono.

memory_write_side_effect_super_transformers_mono:

{1}	$\forall (\text{pm: Memory_struct}, \text{addresses}_1, \text{addresses}_2: \text{PRED}[\text{Address}]):$ $(\text{addresses}_1 \subseteq \text{addresses}_2) \supset$ $(\text{memory_write_side_effect_super_transformers}(\text{pm}, \text{addresses}_1) \subseteq \text{memory_write_side_effect_super_transformers}(\text{pm}, \text{addresses}_2))$
-----	---

Repeatedly Skolemizing and flattening,

Expanding the definition of subset?,

Expanding the definition of member,

Expanding the definition of memory_write_side_effect_super_transformers,

Repeatedly Skolemizing and flattening,

Instantiating quantified variables,

Simplifying, rewriting, and recording with decision procedures,

Forward chaining on subset_transitive,

This completes the proof of memory_write_side_effect_super_transformers_mono.

Q.E.D.

C.121.15 Memory.side_effect_content_unchanged_content

Terse proof for side_effect_content_unchanged_content.

side_effect_content_unchanged_content:

$$\{1\} \quad \forall (\text{se_transformer} : \\ \quad [[\text{Address}, \text{list}[\text{Byte}], \text{bool}] \rightarrow [\text{State} \rightarrow \text{ExprResult}[\text{State}, \text{list}[\text{Byte}]]]], \\ \quad \text{states} : \text{PRED}[\text{State}], \text{addresses} : \text{PRED}[\text{Address}], s : \text{State}, a : \text{Ad-} \\ \text{dress}, \text{bl} : \text{list}[\text{Byte}], \\ \quad \text{cp} : \text{bool}) : \\ \text{side_effect_content_unchanged}(\text{addresses}, \text{states}, \text{se_transformer}) \wedge \\ \text{states}(s) \wedge \\ (\text{address_block}(a, \text{length}(\text{bl})) \subseteq \text{addresses}) \wedge \\ \text{OK}?(\text{se_transformer}(a, \text{bl}, \text{cp})(s)) \\ \supset \text{data}(\text{se_transformer}(a, \text{bl}, \text{cp})(s)) = \text{bl}$$

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of side_effect_content_unchanged_content.

Q.E.D.

C.121.16 Memory.side_effect_content_unchanged_mono

Terse proof for side_effect_content_unchanged_mono.

side_effect_content_unchanged_mono:

$$\{1\} \quad \forall (\text{se_transformer} : \\ \quad [[\text{Address}, \text{list}[\text{Byte}], \text{bool}] \rightarrow [\text{State} \rightarrow \text{ExprResult}[\text{State}, \text{list}[\text{Byte}]]]], \\ \quad \text{states} : \text{PRED}[\text{State}], \text{addresses}_1, \text{addresses}_2 : \text{PRED}[\text{Address}]) : \\ (\text{addresses}_1 \subseteq \text{addresses}_2) \wedge \\ \text{side_effect_content_unchanged}(\text{addresses}_2, \text{states}, \text{se_transformer}) \\ \supset \text{side_effect_content_unchanged}(\text{addresses}_1, \text{states}, \text{se_transformer})$$

Expanding the definition of side_effect_content_unchanged,

Repeatedly Skolemizing and flattening,

Instantiating quantified variables,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Keeping (-2 -5 1) and hiding *,

Rewriting using subset_transitive, matching in * where a gets address_block(a!1, length(bl!1)), b gets addresses!1, c gets addresses!2,

This completes the proof of side_effect_content_unchanged_mono.

Q.E.D.

C.121.17 Memory.side_effect_content_unchanged_composition

Terse proof for side_effect_content_unchanged_composition.

side_effect_content_unchanged_composition:

{1}	$\forall (\text{addresses: PRED}[\text{Address}],$ $\text{se_transformer1},$ se_transformer2: $[[\text{Address}, \text{list}[\text{Byte}], \text{bool}] \rightarrow [\text{State} \rightarrow \text{ExprResult}[\text{State}, \text{list}[\text{Byte}]]]],$ $\text{states: PRED}[\text{State}]):$ $\text{transformer_invariant?}(\text{states}, \text{se_super_transformers}(\text{addresses}, \text{se_transformer1})) \wedge$ $\text{side_effect_content_unchanged}(\text{addresses}, \text{states}, \text{se_transformer1}) \wedge$ $\text{side_effect_content_unchanged}(\text{addresses}, \text{states}, \text{se_transformer2})$ \supset $\text{side_effect_content_unchanged}(\text{addresses}, \text{states},$ $\lambda (a: \text{Address}, \text{bl}: \text{list}[\text{Byte}], \text{cp}: \text{bool}):$ $\text{se_transformer1}(a, \text{bl}, \text{cp}) \#\#$ $(\lambda (\text{bl1}: \text{list}[\text{Byte}]): \text{se_transformer2}(a, \text{bl1}, \text{cp})))$
-----	---

Expanding the definition of side_effect_content_unchanged,

Repeatedly Skolemizing and flattening,

Instantiating the top quantifier in -3 with the terms: (s!1 a!1 bl!1 cp!1),

Expanding the definition of ##,

Installing automatic rewrites from: has_next_state

Using lemma expr_transformer_invariant_next_ok,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 2 subgoals:

side_effect_content_unchanged_composition.1:

{-1}	transformer_invariant?(states', se_super_transformers(addresses', se_transformer1'))
{-2}	states'(s')
{-3}	states'(state(se_transformer1'(a', bl', cp')(s')))
{-4}	every($\lambda (x: \text{number}):$ $\text{number_field_pred}(x) \wedge$ $\text{real_pred}(x) \wedge$ $\text{rational_pred}(x) \wedge \text{integer_pred}(x) \wedge x \geq 0 \wedge x < \text{max_byte}$ (bl'))
{-5}	(address_block(a', length(bl')) \subseteq addresses')
{-6}	data(se_transformer1'(a', bl', cp')(s')) = bl'
{-7}	$\forall (s: \text{State}, a: \text{Address}, \text{bl}: \text{list}[\text{Byte}], \text{cp}: \text{bool}):$ states'(s) \wedge (address_block(a, length(bl)) \subseteq addresses') \wedge OK?(se_transformer2'(a, bl, cp)(s)) \supset data(se_transformer2'(a, bl, cp)(s)) = bl
{-8}	OK?(se_transformer1'(a', bl', cp')(s'))
{-9}	OK?(se_transformer2'(a', data(se_transformer1'(a', bl', cp')(s')), cp') (state(se_transformer1'(a', bl', cp')(s'))))
{1}	$\text{data}(\text{se_transformer2}'(a', \text{data}(\text{se_transformer1}'(a', \text{bl}', \text{cp}')(s')), \text{cp}')$ (state(se_transformer1'(a', bl', cp')(s')))) = bl'

Instantiating the top quantifier in -7 with the terms: (state(se_transformer1!1(a!1, bl!1, cp!1)(s!1)) a!1 bl!1 cp!1),

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of side_effect_content_unchanged_composition.1.

side_effect_content_unchanged_composition.2:

<pre> {-1} transformer_invariant?(states', se_super_transformers(addresses', se_transformer1')) {-2} states'(s') {-3} every(λ (x: number): number_field_pred(x) ∧ real_pred(x) ∧ rational_pred(x) ∧ integer_pred(x) ∧ x ≥ 0 ∧ x < max_byte) (bl') {-4} (address_block(a', length(bl')) ⊆ addresses') {-5} data(se_transformer1'(a', bl', cp')(s')) = bl' {-6} ∀ (s: State, a: Address, bl: list[Byte], cp: bool): states'(s) ∧ (address_block(a, length(bl)) ⊆ addresses') ∧ OK?(se_transformer2'(a, bl, cp)(s)) ⊃ data(se_transformer2'(a, bl, cp)(s)) = bl {-7} OK?(se_transformer1'(a', bl', cp')(s')) {-8} OK?(se_transformer2'(a', data(se_transformer1'(a', bl', cp')(s')), cp') (state(se_transformer1'(a', bl', cp')(s')))) </pre>	<pre> {1} se_super_transformers(addresses', se_transformer1') (expr_2_super(se_transformer1'(a', bl', cp'))) {2} data(se_transformer2'(a', data(se_transformer1'(a', bl', cp')(s')), cp') (state(se_transformer1'(a', bl', cp')(s')))) = bl' </pre>
---	---

Expanding the definition of se_super_transformers,

Instantiating quantified variables,

This completes the proof of side_effect_content_unchanged_composition.2.

Q.E.D.

C.122 Proofs for Memory_Change (memory.pvs)

C.122.1 Memory_Change.unchanged_memory_invariant?_TCC1

Terse proof for unchanged_memory_invariant?_TCC1.

unchanged_memory_invariant?_TCC1:

<pre> {1} ∀ (pm: Memory_struct[State], states: PRED[State], transformers: PRED[[State → SuperResult[State]]], ad- dresses: PRED[Address]): transformer_invariant?(states, transformers) ⊃ (∀ (s: State, q: [State → SuperResult[State]], a: Address): OK?(memory_read(pm)(a)(s)) ∧ OK?(q(s)) ∧ addresses(a) ∧ transformers(q) ∧ states(s) ⊃ OK?[State](q(s)) ∨ abnormal?[State](q(s))) </pre>	
---	--

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of unchanged_memory_invariant?_TCC1.

Q.E.D.

C.122.2

Memory_Change.unchanged_memory_invariant_unchanged_TCC1

Terse proof for `unchanged_memory_invariant_unchanged_TCC1`.

`unchanged_memory_invariant_unchanged_TCC1`:

$$\{1\} \quad \forall (\text{addresses: PRED}[\text{Address}], \text{pm: Memory_struct}[\text{State}], \text{states: PRED}[\text{State}], \\ \text{transformers: PRED}[[\text{State} \rightarrow \text{SuperResult}[\text{State}]]], s: \text{State}, \\ q: [\text{State} \rightarrow \text{SuperResult}[\text{State}]], a: \text{Address}): \\ \text{OK?}(\text{memory_read}(\text{pm})(a)(s)) \wedge \\ \text{OK?}(q(s)) \wedge \\ \text{addresses}(a) \wedge \\ \text{transformers}(q) \wedge \\ \text{states}(s) \wedge \text{unchanged_memory_invariant?}(\text{pm}, \text{states}, \text{transformers}, \text{addresses}) \\ \supset \text{OK?}[\text{State}](q(s)) \vee \text{abnormal?}[\text{State}](q(s))$$

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `unchanged_memory_invariant_unchanged_TCC1`.

Q.E.D.

C.122.3 Memory_Change.unchanged_memory_invariant_unchanged

Terse proof for `unchanged_memory_invariant_unchanged`.

`unchanged_memory_invariant_unchanged`:

$$\{1\} \quad \forall (\text{addresses: PRED}[\text{Address}], \text{pm: Memory_struct}[\text{State}], \text{states: PRED}[\text{State}], \\ \text{transformers: PRED}[[\text{State} \rightarrow \text{SuperResult}[\text{State}]]], s: \text{State}, \\ q: [\text{State} \rightarrow \text{SuperResult}[\text{State}]], a: \text{Address}): \\ \text{unchanged_memory_invariant?}(\text{pm}, \text{states}, \text{transformers}, \text{addresses}) \wedge \\ \text{states}(s) \wedge \\ \text{transformers}(q) \wedge \\ \text{addresses}(a) \wedge \\ \text{OK?}(q(s)) \wedge \\ \text{OK?}(\text{memory_read}(\text{pm})(a)(s)) \wedge \text{OK?}(\text{memory_read}(\text{pm})(a)(\text{state}(q(s)))) \\ \supset \text{data}(\text{memory_read}(\text{pm})(a)(\text{state}(q(s)))) = \text{data}(\text{memory_read}(\text{pm})(a)(s))$$

Repeatedly Skolemizing and flattening,

Expanding the definition of `unchanged_memory_invariant?`,

Applying disjunctive simplification to flatten sequent,

Instantiating quantified variables,

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `unchanged_memory_invariant_unchanged`.

Q.E.D.

C.122.4 Memory_Change.unchanged_memory_invariant_invariant

Terse proof for `unchanged_memory_invariant_invariant`.

unchanged_memory_invariant_invariant:

$\{1\} \quad \forall (\text{addresses: PRED}[\text{Address}], \text{pm: Memory_struct}[\text{State}], \text{states: PRED}[\text{State}], \\ \text{transformers: PRED}[[\text{State} \rightarrow \text{SuperResult}[\text{State}]]]): \\ \text{unchanged_memory_invariant?}(\text{pm}, \text{states}, \text{transformers}, \text{addresses}) \supset \\ \text{transformer_invariant?}(\text{states}, \text{transformers})$

Repeatedly Skolemizing and flattening,
 Expanding the definition of unchanged_memory_invariant?,
 Simplifying, rewriting, and recording with decision procedures,
 This completes the proof of unchanged_memory_invariant_invariant.
 Q.E.D.

C.122.5 Memory_Change.unchanged_memory_invariant_mono

Terse proof for unchanged_memory_invariant_mono.

unchanged_memory_invariant_mono:

$\{1\} \quad \forall (\text{pm: Memory_struct}[\text{State}], \text{states: PRED}[\text{State}], \\ \text{transformers}_1, \text{transformers}_2: \text{PRED}[[\text{State} \rightarrow \text{SuperResult}[\text{State}]]], \\ \text{addresses}_1, \text{addresses}_2: \text{PRED}[\text{Address}]): \\ (\text{transformers}_1 \subseteq \text{transformers}_2) \wedge \\ (\text{addresses}_1 \subseteq \text{addresses}_2) \wedge \\ \text{unchanged_memory_invariant?}(\text{pm}, \text{states}, \text{transformers}_2, \text{addresses}_2) \\ \supset \text{unchanged_memory_invariant?}(\text{pm}, \text{states}, \text{transformers}_1, \text{addresses}_1)$
--

Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of unchanged_memory_invariant_mono.
 Q.E.D.

C.122.6

Memory_Change.unchanged_memory_invariant_next_ok_TCC1

Terse proof for unchanged_memory_invariant_next_ok_TCC1.

unchanged_memory_invariant_next_ok_TCC1:

$\{1\} \quad \forall (\text{addresses: PRED}[\text{Address}], \text{pm: Memory_struct}[\text{State}], \text{states: PRED}[\text{State}], \\ \text{transformers: PRED}[[\text{State} \rightarrow \text{SuperResult}[\text{State}]]], s: \text{State}, \\ q: [\text{State} \rightarrow \text{SuperResult}[\text{State}]]): \\ \text{unchanged_memory_invariant?}(\text{pm}, \text{states}, \text{transformers}, \text{addresses}) \wedge \\ \text{states}(s) \wedge \text{transformers}(q) \wedge \text{has_next_state}(q(s)) \\ \supset \text{OK?}[\text{State}](q(s)) \vee \text{abnormal?}[\text{State}](q(s))$

Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of unchanged_memory_invariant_next_ok_TCC1.
 Q.E.D.

C.122.7 Memory_Change.unchanged_memory_invariant_next_ok

Terse proof for unchanged_memory_invariant_next_ok.

C Proof scripts

unchanged_memory_invariant_next_ok:

$$\{1\} \quad \forall (\text{addresses: PRED}[\text{Address}], \text{pm: Memory_struct}[\text{State}], \text{states: PRED}[\text{State}], \\ \text{transformers: PRED}[[\text{State} \rightarrow \text{SuperResult}[\text{State}]]], s: \text{State}, \\ q: [\text{State} \rightarrow \text{SuperResult}[\text{State}]]): \\ \text{unchanged_memory_invariant?}(\text{pm}, \text{states}, \text{transformers}, \text{addresses}) \wedge \\ \text{states}(s) \wedge \text{transformers}(q) \wedge \text{has_next_state}(q(s)) \\ \supset \text{states}(\text{state}(q(s)))$$

Repeatedly Skolemizing and flattening,

Expanding the definition of `unchanged_memory_invariant?`,

Applying disjunctive simplification to flatten sequent,

Using lemma `super_transformer_invariant_next_ok[State]`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `unchanged_memory_invariant_next_ok`.

Q.E.D.

C.122.8

Memory_Change.unchanged_memory_invariant_union_transformers

Terse proof for `unchanged_memory_invariant_union_transformers`.

unchanged_memory_invariant_union_transformers:

$$\{1\} \quad \forall (\text{addresses: PRED}[\text{Address}], \text{pm: Memory_struct}[\text{State}], \text{states: PRED}[\text{State}], \\ \text{transformers}_1, \text{transformers}_2: \text{PRED}[[\text{State} \rightarrow \text{SuperResult}[\text{State}]]]): \\ \text{unchanged_memory_invariant?}(\text{pm}, \text{states}, \text{transformers}_1, \text{addresses}) \wedge \\ \text{unchanged_memory_invariant?}(\text{pm}, \text{states}, \text{transformers}_2, \text{addresses}) \\ \supset \text{unchanged_memory_invariant?}(\text{pm}, \text{states}, (\text{transformers}_1 \cup \text{transformers}_2), \text{ad-} \\ \text{resses})$$

Installing automatic rewrites from: `member`

Repeatedly Skolemizing and flattening,

Expanding the definition of `unchanged_memory_invariant?`,

Applying propositional simplification,

we get 2 subgoals:

unchanged_memory_invariant_union_transformers.1:

{-1}	transformer_invariant?(states', transformers')
{-2}	$\forall (s: \text{State}, q: [\text{State} \rightarrow \text{SuperResult}[\text{State}]], a: \text{Address}):$ states'(s) \wedge transformers'(q) \wedge addresses'(a) \wedge OK?(q(s)) \wedge OK?(memory_read(pm')(a)(s)) \wedge OK?(memory_read(pm')(a)(state(q(s)))) \supset data(memory_read(pm')(a)(state(q(s)))) = data(memory_read(pm')(a)(s))
{-3}	transformer_invariant?(states', transformers'')
{-4}	$\forall (s: \text{State}, q: [\text{State} \rightarrow \text{SuperResult}[\text{State}]], a: \text{Address}):$ states'(s) \wedge transformers''(q) \wedge addresses'(a) \wedge OK?(q(s)) \wedge OK?(memory_read(pm')(a)(s)) \wedge OK?(memory_read(pm')(a)(state(q(s)))) \supset data(memory_read(pm')(a)(state(q(s)))) = data(memory_read(pm')(a)(s))
{1}	transformer_invariant?(states', (transformers' \cup transformers''))

Rewriting using transformer_invariant_union_transformers, matching in *,

This completes the proof of unchanged_memory_invariant_union_transformers.1.

unchanged_memory_invariant_union_transformers.2:

$\begin{array}{l} \{-1\} \text{ transformer_invariant?}(\text{states}', \text{transformers}') \\ \{-2\} \forall (s: \text{State}, q: [\text{State} \rightarrow \text{SuperResult}[\text{State}]], a: \text{Address}): \\ \quad \text{states}'(s) \wedge \\ \quad \text{transformers}'(q) \wedge \\ \quad \text{addresses}'(a) \wedge \\ \quad \text{OK?}(q(s)) \wedge \\ \quad \text{OK?}(\text{memory_read}(\text{pm}') (a)(s)) \wedge \\ \quad \text{OK?}(\text{memory_read}(\text{pm}') (a)(\text{state}(q(s)))) \\ \quad \supset \\ \quad \text{data}(\text{memory_read}(\text{pm}') (a)(\text{state}(q(s)))) = \text{data}(\text{memory_read}(\text{pm}') (a)(s)) \\ \{-3\} \text{ transformer_invariant?}(\text{states}', \text{transformers}'') \\ \{-4\} \forall (s: \text{State}, q: [\text{State} \rightarrow \text{SuperResult}[\text{State}]], a: \text{Address}): \\ \quad \text{states}'(s) \wedge \\ \quad \text{transformers}''(q) \wedge \\ \quad \text{addresses}'(a) \wedge \\ \quad \text{OK?}(q(s)) \wedge \\ \quad \text{OK?}(\text{memory_read}(\text{pm}') (a)(s)) \wedge \\ \quad \text{OK?}(\text{memory_read}(\text{pm}') (a)(\text{state}(q(s)))) \\ \quad \supset \\ \quad \text{data}(\text{memory_read}(\text{pm}') (a)(\text{state}(q(s)))) = \text{data}(\text{memory_read}(\text{pm}') (a)(s)) \end{array}$	<hr style="border: 0.5px solid black;"/> $\begin{array}{l} \{1\} \forall (s: \text{State}, q: [\text{State} \rightarrow \text{SuperResult}[\text{State}]], a: \text{Address}): \\ \quad \text{states}'(s) \wedge \\ \quad \text{union}(\text{transformers}', \text{transformers}'')(q) \wedge \\ \quad \text{addresses}'(a) \wedge \\ \quad \text{OK?}(q(s)) \wedge \\ \quad \text{OK?}(\text{memory_read}(\text{pm}') (a)(s)) \wedge \\ \quad \text{OK?}(\text{memory_read}(\text{pm}') (a)(\text{state}(q(s)))) \\ \quad \supset \\ \quad \text{data}(\text{memory_read}(\text{pm}') (a)(\text{state}(q(s)))) = \text{data}(\text{memory_read}(\text{pm}') (a)(s)) \end{array}$
---	---

Hiding formulas: (-1 -3),

Repeatedly Skolemizing and flattening,

Expanding the definition of union,

Splitting conjunctions,

we get 2 subgoals:

unchanged_memory_invariant_union_transformers.2.1:

<div style="display: flex; flex-direction: column; gap: 5px;"> <div style="display: flex; align-items: flex-start;"> {-1} $(q' \in \text{transformers}')$ </div> <div style="display: flex; align-items: flex-start;"> {-2} $\text{states}'(s')$ </div> <div style="display: flex; align-items: flex-start;"> {-3} $\text{addresses}'(a')$ </div> <div style="display: flex; align-items: flex-start;"> {-4} $\text{OK?}(q'(s'))$ </div> <div style="display: flex; align-items: flex-start;"> {-5} $\text{OK?}(\text{memory_read}(\text{pm}'))(a')(s')$ </div> <div style="display: flex; align-items: flex-start;"> {-6} $\text{OK?}(\text{memory_read}(\text{pm}'))(a')(\text{state}(q'(s')))$ </div> <div style="display: flex; align-items: flex-start;"> {-7} $\forall (s: \text{State}, q: [\text{State} \rightarrow \text{SuperResult}[\text{State}]], a: \text{Address}):$ $\text{states}'(s) \wedge$ $\text{transformers}'(q) \wedge$ $\text{addresses}'(a) \wedge$ $\text{OK?}(q(s)) \wedge$ $\text{OK?}(\text{memory_read}(\text{pm}'))(a)(s) \wedge$ $\text{OK?}(\text{memory_read}(\text{pm}'))(a)(\text{state}(q(s)))$ \supset $\text{data}(\text{memory_read}(\text{pm}'))(a)(\text{state}(q(s))) = \text{data}(\text{memory_read}(\text{pm}'))(a)(s)$ </div> <div style="display: flex; align-items: flex-start;"> {-8} $\forall (s: \text{State}, q: [\text{State} \rightarrow \text{SuperResult}[\text{State}]], a: \text{Address}):$ $\text{states}'(s) \wedge$ $\text{transformers}''(q) \wedge$ $\text{addresses}'(a) \wedge$ $\text{OK?}(q(s)) \wedge$ $\text{OK?}(\text{memory_read}(\text{pm}'))(a)(s) \wedge$ $\text{OK?}(\text{memory_read}(\text{pm}'))(a)(\text{state}(q(s)))$ \supset $\text{data}(\text{memory_read}(\text{pm}'))(a)(\text{state}(q(s))) = \text{data}(\text{memory_read}(\text{pm}'))(a)(s)$ </div> </div>	<div style="display: flex; align-items: flex-start; border-top: 1px solid black; border-left: 1px solid black; border-right: 1px solid black; padding: 5px 10px 5px 10px;"> {1} $\text{data}(\text{memory_read}(\text{pm}'))(a')(\text{state}(q'(s'))) =$ $\text{data}(\text{memory_read}(\text{pm}'))(a')(s')$ </div>
---	--

Instantiating quantified variables,

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of unchanged_memory_invariant_union_transformers.2.1.

unchanged_memory_invariant_union_transformers.2.2:

<pre> {-1} (q' ∈ transformers'') {-2} states'(s') {-3} addresses'(a') {-4} OK?(q'(s')) {-5} OK?(memory_read(pm')(a')(s')) {-6} OK?(memory_read(pm')(a')(state(q'(s')))) {-7} ∀ (s: State, q: [State → SuperResult[State]], a: Address): states'(s) ∧ transformers'(q) ∧ addresses'(a) ∧ OK?(q(s)) ∧ OK?(memory_read(pm')(a)(s)) ∧ OK?(memory_read(pm')(a)(state(q(s)))) ⊃ data(memory_read(pm')(a)(state(q(s)))) = data(memory_read(pm')(a)(s)) {-8} ∀ (s: State, q: [State → SuperResult[State]], a: Address): states'(s) ∧ transformers''(q) ∧ addresses'(a) ∧ OK?(q(s)) ∧ OK?(memory_read(pm')(a)(s)) ∧ OK?(memory_read(pm')(a)(state(q(s)))) ⊃ data(memory_read(pm')(a)(state(q(s)))) = data(memory_read(pm')(a)(s)) </pre>	<hr style="border: 0.5px solid black; margin-bottom: 5px;"/> <pre> {1} data(memory_read(pm')(a')(state(q'(s')))) = data(memory_read(pm')(a')(s')) </pre>
---	---

Instantiating quantified variables,

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `unchanged_memory_invariant_union_transformers.2.2`.

Q.E.D.

C.122.9

Memory_Change.unchanged_memory_invariant_union_addresses

Terse proof for `unchanged_memory_invariant_union_addresses`.

`unchanged_memory_invariant_union_addresses`:

<pre> {1} ∀ (pm: Memory_struct[State], states: PRED[State], transformers: PRED[[State → SuperResult[State]]], addresses_1, addresses_2: PRED[Address]): unchanged_memory_invariant?(pm, states, transformers, addresses_1) ∧ unchanged_memory_invariant?(pm, states, transformers, addresses_2) ⊃ unchanged_memory_invariant?(pm, states, transform- ers, (addresses_1 ∪ addresses_2)) </pre>	<hr style="border: 0.5px solid black; margin-bottom: 5px;"/>
--	--

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `unchanged_memory_invariant_union_addresses`.

Q.E.D.

C.122.10 Memory_Change.unchanged_memory_invariant_empty

Terse proof for `unchanged_memory_invariant_empty`.

`unchanged_memory_invariant_empty`:

$$\{1\} \quad \forall (\text{addresses: PRED}[\text{Address}], \text{pm: Memory_struct}[\text{State}], \text{states: PRED}[\text{State}], \\ \text{transformers: PRED}[[\text{State} \rightarrow \text{SuperResult}[\text{State}]]]): \\ \text{empty?}(\text{states}) \vee \text{empty?}(\text{transformers}) \supset \\ \text{unchanged_memory_invariant?}(\text{pm}, \text{states}, \text{transformers}, \text{addresses})$$

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `unchanged_memory_invariant_empty`.

Q.E.D.

C.122.11

Memory_Change.unchanged_memory_invariant_all_transformers

Terse proof for `unchanged_memory_invariant_all_transformers`.

`unchanged_memory_invariant_all_transformers`:

$$\{1\} \quad \forall (\text{addresses: PRED}[\text{Address}], \text{pm: Memory_struct}[\text{State}], \text{states: PRED}[\text{State}], \\ \text{transformers: PRED}[[\text{State} \rightarrow \text{SuperResult}[\text{State}]]]): \\ (\forall (q: [\text{State} \rightarrow \text{SuperResult}[\text{State}]]): \\ \text{transformers}(q) \supset \text{unchanged_memory_invariant?}(\text{pm}, \text{states}, \text{singleton}(q), \text{ad-} \\ \text{resses})) \\ \supset \text{unchanged_memory_invariant?}(\text{pm}, \text{states}, \text{transformers}, \text{addresses})$$

Expanding the definition of `unchanged_memory_invariant?`,

Repeatedly Skolemizing and flattening,

Applying propositional simplification,

we get 2 subgoals:

`unchanged_memory_invariant_all_transformers.1`:

$$\{-1\} \quad \forall (q: [\text{State} \rightarrow \text{SuperResult}[\text{State}]]): \\ \text{transformers}'(q) \supset \\ \text{transformer_invariant?}(\text{states}', \text{singleton}(q)) \wedge \\ (\forall (s: \text{State}, q_1: [\text{State} \rightarrow \text{SuperResult}[\text{State}]], a: \text{Address}): \\ \text{states}'(s) \wedge \\ \text{singleton}(q)(q_1) \wedge \\ \text{addresses}'(a) \wedge \\ \text{OK?}(q_1(s)) \wedge \\ \text{OK?}(\text{memory_read}(\text{pm}')(a)(s)) \wedge \\ \text{OK?}(\text{memory_read}(\text{pm}')(a)(\text{state}(q_1(s)))) \\ \supset \\ \text{data}(\text{memory_read}(\text{pm}')(a)(\text{state}(q_1(s)))) = \\ \text{data}(\text{memory_read}(\text{pm}')(a)(s))) \\ \{1\} \quad \text{transformer_invariant?}(\text{states}', \text{transformers}')$$

Rewriting using `transformer_invariant_all_transformers`, matching in `*`,

Repeatedly Skolemizing and flattening,

Instantiating quantified variables,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `unchanged_memory_invariant_all_transformers.1`.

unchanged_memory_invariant_all_transformers.2:

$\{-1\} \quad \forall (q: [\text{State} \rightarrow \text{SuperResult}[\text{State}]]):$ $\text{transformers}'(q) \supset$ $\text{transformer_invariant}'(\text{states}', \text{singleton}(q)) \wedge$ $(\forall (s: \text{State}, q_1: [\text{State} \rightarrow \text{SuperResult}[\text{State}]], a: \text{Address}):$ $\text{states}'(s) \wedge$ $\text{singleton}(q)(q_1) \wedge$ $\text{addresses}'(a) \wedge$ $\text{OK}'(q_1(s)) \wedge$ $\text{OK}'(\text{memory_read}(\text{pm}') (a)(s)) \wedge$ $\text{OK}'(\text{memory_read}(\text{pm}') (a)(\text{state}(q_1(s))))$ \supset $\text{data}(\text{memory_read}(\text{pm}') (a)(\text{state}(q_1(s)))) =$ $\text{data}(\text{memory_read}(\text{pm}') (a)(s)))$	$\{1\} \quad \forall (s: \text{State}, q: [\text{State} \rightarrow \text{SuperResult}[\text{State}]], a: \text{Address}):$ $\text{states}'(s) \wedge$ $\text{transformers}'(q) \wedge$ $\text{addresses}'(a) \wedge$ $\text{OK}'(q(s)) \wedge$ $\text{OK}'(\text{memory_read}(\text{pm}') (a)(s)) \wedge$ $\text{OK}'(\text{memory_read}(\text{pm}') (a)(\text{state}(q(s))))$ \supset $\text{data}(\text{memory_read}(\text{pm}') (a)(\text{state}(q(s)))) = \text{data}(\text{memory_read}(\text{pm}') (a)(s))$
--	--

Repeatedly Skolemizing and flattening,

Instantiating quantified variables,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Instantiating quantified variables,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Expanding the definition of singleton,

which is trivially true.

This completes the proof of `unchanged_memory_invariant_all_transformers.2`.

Q.E.D.

C.123 Proofs for Memory_Change_2 (memory.pvs)

C.123.1 Mem-

ory_Change_2.unchanged_memory_write_invariant_mono_addresses

Terse proof for `unchanged_memory_write_invariant_mono_addresses`.

`unchanged_memory_write_invariant_mono_addresses`:

$\{1\} \quad \forall (\text{pm}: \text{Memory_struct}[\text{State}], \text{states}: \text{PRED}[\text{State}],$ $\text{addresses}_1, \text{addresses}_2: \text{PRED}[\text{Address}]):$ $(\text{addresses}_1 \subseteq \text{addresses}_2) \wedge \text{unchanged_memory_write_invariant}'(\text{pm}, \text{states}, \text{ad-}$ $\text{resses}_2)$ $\supset \text{unchanged_memory_write_invariant}'(\text{pm}, \text{states}, \text{addresses}_1)$	
---	--

Repeatedly Skolemizing and flattening,

Expanding the definition of `unchanged_memory_write_invariant'`,

Repeatedly Skolemizing and flattening,

Instantiating quantified variables,
 Applying propositional simplification,
 we get 2 subgoals:

unchanged_memory_write_invariant_mono_addresses.1:

$\{-1\}$	$\text{unchanged_memory_invariant?}(pm', \text{states}',$ $\quad \text{singleton}(\text{expr_2_super}(\text{memory_write}(pm')(\text{waddr}', b')),$ $\quad (\text{addresses}'' \setminus \{\text{waddr}'\}))$
$\{-2\}$	$b' < \text{max_byte}$
$\{-3\}$	$(\text{addresses}' \subseteq \text{addresses}'')$
$\{-4\}$	$\text{addresses}'(\text{waddr}')$
$\{1\}$	
$\{1\}$	$\text{unchanged_memory_invariant?}(pm', \text{states}',$ $\quad \text{singleton}(\text{expr_2_super}(\text{memory_write}(pm')(\text{waddr}', b')),$ $\quad (\text{addresses}' \setminus \{\text{waddr}'\}))$

Using lemma unchanged_memory_invariant_mono,
 Simplifying, rewriting, and recording with decision procedures,
 Rewriting using subset_equal, matching in *,
 Keeping (-3 1) and hiding *,
 Trying repeated skolemization, instantiation, and if-lifting,
 Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of unchanged_memory_write_invariant_mono_addresses.1.

unchanged_memory_write_invariant_mono_addresses.2:

$\{-1\}$	$b' < \text{max_byte}$
$\{-2\}$	$(\text{addresses}' \subseteq \text{addresses}'')$
$\{-3\}$	$\text{addresses}'(\text{waddr}')$
$\{1\}$	
$\{1\}$	$\text{addresses}''(\text{waddr}')$
$\{2\}$	$\text{unchanged_memory_invariant?}(pm', \text{states}',$ $\quad \text{singleton}(\text{expr_2_super}(\text{memory_write}(pm')(\text{waddr}', b')),$ $\quad (\text{addresses}' \setminus \{\text{waddr}'\}))$

Installing automatic rewrites from: subset? singleton member

Repeatedly simplifying with decision procedures, rewriting, propositional reasoning, quantifier instantiation, skolemization, if-lifting and equality replacement,

This completes the proof of unchanged_memory_write_invariant_mono_addresses.2.

Q.E.D.

C.123.2 Memory_Change_2.unchanged_memory_write_invariant_transformer_invariant

Terse proof for unchanged_memory_write_invariant_transformer_invariant.

unchanged_memory_write_invariant_transformer_invariant:

$\{1\}$	
$\{1\}$	$\forall (\text{addresses}: \text{PRED}[\text{Address}], \text{pm}: \text{Memory_struct}[\text{State}], \text{states}: \text{PRED}[\text{State}]):$ $\quad \text{unchanged_memory_write_invariant?}(\text{pm}, \text{states}, \text{addresses}) \supset$ $\quad \text{transformer_invariant?}(\text{states}, \text{memory_write_transformers}(\text{pm}, \text{addresses}))$

Repeatedly Skolemizing and flattening,
 Expanding the definition of unchanged_memory_write_invariant?,
 Expanding the definition of unchanged_memory_invariant?,
 Expanding the definition of transformer_invariant?,
 Repeatedly Skolemizing and flattening,
 Expanding the definition of memory_write_transformers,

C Proof scripts

Repeatedly Skolemizing and flattening,
Replacing using formula -5,
Instantiating quantified variables,
Simplifying, rewriting, and recording with decision procedures,
Applying disjunctive simplification to flatten sequent,
Expanding the definition of singleton,
Instantiating quantified variables,
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `unchanged_memory_write_invariant_transformer_invariant`.
Q.E.D.

C.123.3 Mem- ory_Change_2.unchanged_memory_write_invariant_unchanged_memory_invariant

Terse proof for `unchanged_memory_write_invariant_unchanged_memory_invariant`.

`unchanged_memory_write_invariant_unchanged_memory_invariant`:

$\{1\} \quad \forall (pm: \text{Memory_struct}[\text{State}], \text{states}: \text{PRED}[\text{State}], \\ \text{all_addresses}, \text{unchanged_addresses}, \text{changed_addresses}: \text{PRED}[\text{Address}]): \\ \text{unchanged_memory_write_invariant?}(pm, \text{states}, \text{all_addresses}) \wedge \\ (\text{unchanged_addresses} \subseteq \text{all_addresses}) \wedge \\ (\text{changed_addresses} \subseteq \text{all_addresses}) \wedge \text{disjoint?}(\text{unchanged_addresses}, \text{changed_addresses}) \\ \supset \\ \text{unchanged_memory_invariant?}(pm, \text{states}, \text{memory_write_transformers}(pm, \text{changed_addresses}, \\ \text{unchanged_addresses}))$

Repeatedly Skolemizing and flattening,
Expanding the definition of `unchanged_memory_invariant?`,
Applying propositional simplification,
we get 2 subgoals:

`unchanged_memory_write_invariant_unchanged_memory_invariant.1`:

$\begin{array}{l} \{-1\} \quad \text{unchanged_memory_write_invariant?}(pm', \text{states}', \text{all_addresses}') \\ \{-2\} \quad (\text{unchanged_addresses}' \subseteq \text{all_addresses}') \\ \{-3\} \quad (\text{changed_addresses}' \subseteq \text{all_addresses}') \\ \{-4\} \quad \text{disjoint?}(\text{unchanged_addresses}', \text{changed_addresses}') \\ \hline \{1\} \quad \text{transformer_invariant?}(\text{states}', \text{memory_write_transformers}(pm', \text{changed_addresses}')) \end{array}$

Rewriting using `unchanged_memory_write_invariant_transformer_invariant`, matching in *,
Hiding formulas: -2,
Forward chaining on `unchanged_memory_write_invariant_mono_addresses`,
This completes the proof of `unchanged_memory_write_invariant_unchanged_memory_invariant.1`.

unchanged_memory_write_invariant_unchanged_memory_invariant.2:

{-1}	unchanged_memory_write_invariant?(pm', states', all_addresses')
{-2}	(unchanged_addresses' \subseteq all_addresses')
{-3}	(changed_addresses' \subseteq all_addresses')
{-4}	disjoint?(unchanged_addresses', changed_addresses')
{1}	$\forall (s: \text{State}, q: [\text{State} \rightarrow \text{SuperResult}[\text{State}]], a: \text{Address}):$ $\text{states}'(s) \wedge$ $\text{memory_write_transformers}(pm', \text{changed_addresses}')(q) \wedge$ $\text{unchanged_addresses}'(a) \wedge$ $\text{OK?}(q(s)) \wedge$ $\text{OK?}(\text{memory_read}(pm')(a)(s)) \wedge$ $\text{OK?}(\text{memory_read}(pm')(a)(\text{state}(q(s))))$ \supset $\text{data}(\text{memory_read}(pm')(a)(\text{state}(q(s)))) = \text{data}(\text{memory_read}(pm')(a)(s))$

Expanding the definition of memory_write_transformers,

Repeatedly Skolemizing and flattening,

Replacing using formula -4,

Expanding the definition of unchanged_memory_write_invariant?,

Expanding the definition of unchanged_memory_invariant?,

Instantiating quantified variables,

Applying propositional simplification,

we get 2 subgoals:

unchanged_memory_write_invariant_unchanged_memory_invariant.2.1:

{-1}	transformer_invariant?(states', singleton(expr_2_super(memory_write(pm')(a'', b'))))
{-2}	$\forall (s: \text{State}, q: [\text{State} \rightarrow \text{SuperResult}[\text{State}]], a: \text{Address}):$ $\text{states}'(s) \wedge$ $\text{singleton}(\text{expr_2_super}(\text{memory_write}(pm')(a'', b')))(q) \wedge$ $\text{remove}(a'', \text{all_addresses}')(a) \wedge$ $\text{OK?}(q(s)) \wedge$ $\text{OK?}(\text{memory_read}(pm')(a)(s)) \wedge$ $\text{OK?}(\text{memory_read}(pm')(a)(\text{state}(q(s))))$ \supset $\text{data}(\text{memory_read}(pm')(a)(\text{state}(q(s)))) = \text{data}(\text{memory_read}(pm')(a)(s))$
{-3}	$b' < \text{max_byte}$
{-4}	$\text{states}'(s')$
{-5}	$\text{changed_addresses}'(a'')$
{-6}	$q' = \text{expr_2_super}(\text{memory_write}(pm')(a'', b'))$
{-7}	$\text{unchanged_addresses}'(a')$
{-8}	$\text{OK?}(\text{expr_2_super}(\text{memory_write}(pm')(a'', b'))(s'))$
{-9}	$\text{OK?}(\text{memory_read}(pm')(a')(s'))$
{-10}	$\text{OK?}(\text{memory_read}(pm')$ $(a')(\text{state}(\text{expr_2_super}(\text{memory_write}(pm')(a'', b'))(s'))))$
{-11}	$(\text{unchanged_addresses}' \subseteq \text{all_addresses}')$
{-12}	$(\text{changed_addresses}' \subseteq \text{all_addresses}')$
{-13}	$\text{disjoint?}(\text{unchanged_addresses}', \text{changed_addresses}')$
{1}	$\text{data}(\text{memory_read}(pm')$ $(a')(\text{state}(\text{expr_2_super}(\text{memory_write}(pm')(a'', b'))(s'))))$ $= \text{data}(\text{memory_read}(pm')(a')(s'))$

Rewriting using -2, matching in *,

we get 2 subgoals:

unchanged_memory_write_invariant_unchanged_memory_invariant.2.1.1.1:

{-1}	transformer_invariant?(states', singleton(expr_2_super(memory_write(pm')(a'', b'))))
{-2}	$\forall (s: \text{State}, q: [\text{State} \rightarrow \text{SuperResult}[\text{State}]], a: \text{Address}):$ states'(s) \wedge singleton(expr_2_super(memory_write(pm')(a'', b')))(q) \wedge remove(a'', all_addresses')(a) \wedge OK?(q(s)) \wedge OK?(memory_read(pm')(a)(s)) \wedge OK?(memory_read(pm')(a)(state(q(s)))) \supset data(memory_read(pm')(a)(state(q(s)))) = data(memory_read(pm')(a)(s))
{-3}	b' < max_byte
{-4}	states'(s')
{-5}	changed_addresses'(a'')
{-6}	q' = expr_2_super(memory_write(pm')(a'', b'))
{-7}	unchanged_addresses'(a')
{-8}	OK?(expr_2_super(memory_write(pm')(a'', b'))(s'))
{-9}	OK?(memory_read(pm')(a')(s'))
{-10}	OK?(memory_read(pm')(a')(state(expr_2_super(memory_write(pm')(a'', b'))(s'))))
{-11}	(unchanged_addresses' \subseteq all_addresses')
{-12}	(changed_addresses' \subseteq all_addresses')
{-13}	disjoint?(unchanged_addresses', changed_addresses')
{1}	singleton(expr_2_super(memory_write(pm')(a'', b')) (expr_2_super(memory_write(pm')(a'', b'))))
{2}	data(memory_read(pm')(a')(state(expr_2_super(memory_write(pm')(a'', b'))(s')))) = data(memory_read(pm')(a')(s'))

Expanding the definition of singleton,

which is trivially true.

This completes the proof of unchanged_memory_write_invariant_unchanged_memory_invariant.2.1.1.1.

unchanged_memory_write_invariant_unchanged_memory_invariant.2.1.2:

{-1}	transformer_invariant?(states', singleton(expr_2_super(memory_write(pm')(a'', b'))))
{-2}	$\forall (s: \text{State}, q: [\text{State} \rightarrow \text{SuperResult}[\text{State}]], a: \text{Address}):$ $\text{states}'(s) \wedge$ $\text{singleton}(\text{expr_2_super}(\text{memory_write}(\text{pm}')(\text{a}'', \text{b}'))(q) \wedge$ $\text{remove}(\text{a}'', \text{all_addresses}')(\text{a}) \wedge$ $\text{OK?}(q(s)) \wedge$ $\text{OK?}(\text{memory_read}(\text{pm}')(\text{a})(s)) \wedge$ $\text{OK?}(\text{memory_read}(\text{pm}')(\text{a})(\text{state}(q(s))))$
	\supset
	$\text{data}(\text{memory_read}(\text{pm}')(\text{a})(\text{state}(q(s)))) = \text{data}(\text{memory_read}(\text{pm}')(\text{a})(s))$
{-3}	$b' < \text{max_byte}$
{-4}	$\text{states}'(s')$
{-5}	$\text{changed_addresses}'(a'')$
{-6}	$q' = \text{expr_2_super}(\text{memory_write}(\text{pm}')(\text{a}'', \text{b}'))$
{-7}	$\text{unchanged_addresses}'(a')$
{-8}	$\text{OK?}(\text{expr_2_super}(\text{memory_write}(\text{pm}')(\text{a}'', \text{b}'))(s'))$
{-9}	$\text{OK?}(\text{memory_read}(\text{pm}')(\text{a}')(s'))$
{-10}	$\text{OK?}(\text{memory_read}(\text{pm}')$ $(\text{a}')(\text{state}(\text{expr_2_super}(\text{memory_write}(\text{pm}')(\text{a}'', \text{b}'))(s'))))$
{-11}	$(\text{unchanged_addresses}' \subseteq \text{all_addresses}')$
{-12}	$(\text{changed_addresses}' \subseteq \text{all_addresses}')$
{-13}	$\text{disjoint?}(\text{unchanged_addresses}', \text{changed_addresses}')$
{1}	$\text{remove}(\text{a}'', \text{all_addresses}')(\text{a}')$
{2}	$\text{data}(\text{memory_read}(\text{pm}')$ $(\text{a}')(\text{state}(\text{expr_2_super}(\text{memory_write}(\text{pm}')(\text{a}'', \text{b}'))(s'))))$ $= \text{data}(\text{memory_read}(\text{pm}')(\text{a}')(s'))$

Keeping (-5 -7 -11 -12 -13 1) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of unchanged_memory_write_invariant_unchanged_memory_invariant.2.1.2.

unchanged_memory_write_invariant_unchanged_memory_invariant.2.2:

{-1}	$b' < \text{max_byte}$
{-2}	$\text{states}'(s')$
{-3}	$\text{changed_addresses}'(a'')$
{-4}	$q' = \text{expr_2_super}(\text{memory_write}(\text{pm}')(\text{a}'', \text{b}'))$
{-5}	$\text{unchanged_addresses}'(a')$
{-6}	$\text{OK?}(\text{expr_2_super}(\text{memory_write}(\text{pm}')(\text{a}'', \text{b}'))(s'))$
{-7}	$\text{OK?}(\text{memory_read}(\text{pm}')(\text{a}')(s'))$
{-8}	$\text{OK?}(\text{memory_read}(\text{pm}')$ $(\text{a}')(\text{state}(\text{expr_2_super}(\text{memory_write}(\text{pm}')(\text{a}'', \text{b}'))(s'))))$
{-9}	$(\text{unchanged_addresses}' \subseteq \text{all_addresses}')$
{-10}	$(\text{changed_addresses}' \subseteq \text{all_addresses}')$
{-11}	$\text{disjoint?}(\text{unchanged_addresses}', \text{changed_addresses}')$
{1}	$\text{all_addresses}'(a'')$
{2}	$\text{data}(\text{memory_read}(\text{pm}')$ $(\text{a}')(\text{state}(\text{expr_2_super}(\text{memory_write}(\text{pm}')(\text{a}'', \text{b}'))(s'))))$ $= \text{data}(\text{memory_read}(\text{pm}')(\text{a}')(s'))$

Keeping (-3 -10 1) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of unchanged_memory_write_invariant_unchanged_memory_invariant.2.2.

Q.E.D.

C.123.4 Memory_Change_2.unchanged_memory_invariant_union

Terse proof for `unchanged_memory_invariant_union`.

```
unchanged_memory_invariant_union:
```

<pre>{1} ∃ (pm: Memory_struct[State], states: PRED[State], ro_addr, rw_addr, mod_addr: PRED[Address]): unchanged_memory_invariant?(pm, states, memory_write_transformers(pm, rw_addr), ro_addr) unchanged_memory_write_invariant?(pm, states, rw_addr) ∧ (mod_addr ⊆ rw_addr) ⊃ unchanged_memory_invariant?(pm, states, memory_write_transformers(pm, mod_addr), ((ro_addr ∪ rw_addr) \ mod_addr))</pre>

Repeatedly Skolemizing and flattening,

Case splitting on `difference(union(ro_addr!1, rw_addr!1), mod_addr!1) = union(difference(ro_addr!1, mod_addr!1), difference(rw_addr!1, mod_addr!1))`,

we get 2 subgoals:

```
unchanged_memory_invariant_union.1:
```

<pre>{-1} ((ro_addr' ∪ rw_addr') \ mod_addr') = ((ro_addr' \ mod_addr') ∪ (rw_addr' \ mod_addr'))</pre>
<pre>{-2} unchanged_memory_invariant?(pm', states', memory_write_transformers(pm', rw_addr'), ro_addr')</pre>
<pre>{-3} unchanged_memory_write_invariant?(pm', states', rw_addr')</pre>
<pre>{-4} (mod_addr' ⊆ rw_addr')</pre>
<pre>{1} unchanged_memory_invariant?(pm', states', memory_write_transformers(pm', mod_addr'), ((ro_addr' ∪ rw_addr') \ mod_addr'))</pre>

Replacing using formula -1,

Hiding formulas: -1,

Rewriting using `unchanged_memory_invariant_union_addresses`, matching in *,

we get 2 subgoals:

```
unchanged_memory_invariant_union.1.1:
```

<pre>{-1} unchanged_memory_invariant?(pm', states', memory_write_transformers(pm', rw_addr'), ro_addr')</pre>
<pre>{-2} unchanged_memory_write_invariant?(pm', states', rw_addr')</pre>
<pre>{-3} (mod_addr' ⊆ rw_addr')</pre>
<pre>{1} unchanged_memory_invariant?(pm', states', memory_write_transformers(pm', mod_addr'), (ro_addr' \ mod_addr'))</pre>
<pre>{2} unchanged_memory_invariant?(pm', states', memory_write_transformers(pm', mod_addr'), ((ro_addr' \ mod_addr') ∪ (rw_addr' \ mod_addr')))</pre>

Hiding formulas: -2, 2,

Using lemma `unchanged_memory_invariant_mono`,Installing automatic rewrites from: `memory_write_transformers_mono`

Simplifying, rewriting, and recording with decision procedures,

Keeping 1 and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `unchanged_memory_invariant_union.1.1`.

unchanged_memory_invariant_union.1.2:

{-1}	unchanged_memory_invariant?(pm', states', memory_write_transformers(pm', rw_addr'), ro_addr')
{-2}	unchanged_memory_write_invariant?(pm', states', rw_addr')
{-3}	(mod_addr' \subseteq rw_addr')
{1}	unchanged_memory_invariant?(pm', states', memory_write_transformers(pm', mod_addr'), (rw_addr' \ mod_addr'))
{2}	unchanged_memory_invariant?(pm', states', memory_write_transformers(pm', mod_addr'), ((ro_addr' \ mod_addr') \cup (rw_addr' \ mod_addr')))

Hiding formulas: -1, 2,

Using lemma unchanged_memory_write_invariant_unchanged_memory_invariant,

Simplifying, rewriting, and recording with decision procedures,

Installing automatic rewrites from: difference_subset difference_disjoint_3

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of unchanged_memory_invariant_union.1.2.

unchanged_memory_invariant_union.2:

{-1}	unchanged_memory_invariant?(pm', states', memory_write_transformers(pm', rw_addr'), ro_addr')
{-2}	unchanged_memory_write_invariant?(pm', states', rw_addr')
{-3}	(mod_addr' \subseteq rw_addr')
{1}	((ro_addr' \cup rw_addr') \ mod_addr') = ((ro_addr' \ mod_addr') \cup (rw_addr' \ mod_addr'))
{2}	unchanged_memory_invariant?(pm', states', memory_write_transformers(pm', mod_addr'), ((ro_addr' \cup rw_addr') \ mod_addr'))

Keeping (-3 1) and hiding *,

Trying repeated skolemization, instantiation, if-lifting, and extensionality,

This completes the proof of unchanged_memory_invariant_union.2.

Q.E.D.

C.123.5 Memory_Change_2.transformer_invariant_write_list_nse

Terse proof for transformer_invariant_write_list_nse.

transformer_invariant_write_list_nse:

{1}	\forall (pm: Memory_struct[State], states: PRED[State], waddr: Ad- dress, bl: list[Byte], s: State): transformer_invariant?(states, memory_write_transformers(pm, ad- dress_block(waddr, length(bl)))) \wedge states(s) \supset result_pred(states)(expr_2_super(memory_write_list_nse(pm)(waddr, bl))(s))
-----	---

For the top quantifier in 1, we introduce Skolem constants: (pm' _ _ _ _),

Inducting on bl on formula 1,

we get 2 subgoals:

C Proof scripts

transformer_invariant_write_list_nse.1:

$$\{1\} \quad \forall (states: \text{PRED}[\text{State}], waddr: \text{Address}, s: \text{State}):$$

$$\begin{aligned} & \text{transformer_invariant?}(states, \\ & \qquad \qquad \qquad \text{memory_write_transformers}(pm', \\ & \qquad \qquad \qquad \qquad \qquad \qquad \text{ad-} \\ & \text{dress_block}(waddr, \text{length}(\text{null}))) \\ & \quad \wedge \text{states}(s) \\ & \quad \supset \text{result_pred}(states)(\text{expr_2_super}(\text{memory_write_list_nse}(pm')(waddr, \text{null}))(s)) \end{aligned}$$

Expanding the definition of memory_write_list_nse,

Expanding the definition(s) of (ok_result expr_2_super expr_2_super_res result_pred has_next_state),

Repeatedly Skolemizing and flattening,

This completes the proof of transformer_invariant_write_list_nse.1.

transformer_invariant_write_list_nse.2:

$$\{1\} \quad \forall (cons1_var: \text{Byte}, cons2_var: \text{list}[\text{Byte}]):$$

$$\begin{aligned} & (\forall (states: \text{PRED}[\text{State}], waddr: \text{Address}, s: \text{State}): \\ & \quad \text{transformer_invariant?}(states, \\ & \qquad \qquad \qquad \text{memory_write_transformers}(pm', \\ & \qquad \qquad \qquad \qquad \qquad \qquad \text{address_block}(waddr, \\ & \qquad \qquad \qquad \qquad \qquad \qquad \qquad \qquad \qquad \qquad \text{length}(cons2_var))) \\ & \quad \wedge \text{states}(s) \\ & \quad \supset \\ & \quad \text{result_pred}(states) \\ & \qquad \qquad \qquad (\text{expr_2_super}(\text{memory_write_list_nse}(pm')(waddr, cons2_var))(s))) \\ & \quad \supset \\ & \quad (\forall (states: \text{PRED}[\text{State}], waddr: \text{Address}, s: \text{State}): \\ & \quad \quad \text{transformer_invariant?}(states, \\ & \quad \qquad \qquad \text{memory_write_transformers}(pm', \\ & \quad \quad \quad \text{ad-} \\ & \quad \text{dress_block}(waddr, \\ & \qquad \qquad \qquad \qquad \qquad \qquad \qquad \qquad \qquad \qquad \text{length} \\ & \quad \qquad \qquad \qquad \qquad \qquad \qquad \qquad \qquad \qquad \qquad \text{(cons} \\ & \quad \qquad \qquad \qquad \qquad \qquad \qquad \qquad \qquad \qquad \qquad \qquad \text{(cons1_var} \\ & \quad \qquad \qquad \qquad \qquad \qquad \qquad \qquad \qquad \qquad \qquad \qquad \text{cons2_var} \\ & \quad \quad \wedge \text{states}(s) \\ & \quad \quad \supset \\ & \quad \quad \text{result_pred}(states) \\ & \quad \qquad \qquad (\text{expr_2_super}(\text{memory_write_list_nse}(pm') \\ & \quad \qquad \qquad \qquad \qquad \qquad \qquad \qquad \qquad \qquad \qquad \text{(waddr, cons(cons1_var, cons2_var))} \\ & \quad \qquad \qquad \qquad \qquad \qquad \qquad \qquad \qquad \qquad \qquad \qquad \qquad \qquad \qquad \qquad \text{(s)))) \end{aligned}$$

Repeatedly Skolemizing and flattening,

Expanding the definition of memory_write_list_nse,

Installing automatic rewrites from: length super_comp_expr_forget_expr! address_block

Using lemma memory_write_transformers_memory_write[State],

Simplifying, rewriting, and recording with decision procedures,

Case splitting on OK?(expr_2_super(memory_write(pm!1)(waddr!1, cons1_var!1))(s!1)),

we get 2 subgoals:

transformer_invariant_write_list_nse.2.1:

{-1}	OK?(expr_2_super(memory_write(pm')(waddr', cons1_var'))(s'))		
{-2}	memory_write_transformers(pm', address_block(waddr', 1 + length(cons2_var')) (expr_2_super(memory_write(pm')(waddr', cons1_var'))))		
{-3}	\forall (states: PRED[State], waddr: Address, s: State): transformer_invariant?(states, memory_write_transformers(pm', address_block(waddr, length(cons2_var')))) \wedge states(s) \supset result_pred(states) (expr_2_super(memory_write_list_nse(pm')(waddr, cons2_var'))(s))		
{-4}	transformer_invariant?(states', memory_write_transformers(pm', address_block(waddr', 1 + length(cons2_var'))))		
{-5}	states'(s')		
<table style="width: 100%; border-collapse: collapse;"> <tr> <td style="border-right: 1px solid black; padding-right: 10px; vertical-align: top;">{1}</td> <td style="padding-left: 10px;">result_pred(states') (CASES expr_2_super(memory_write(pm')(waddr', cons1_var'))(s') OF OK(state): expr_2_super(memory_write_list_nse(pm')(waddr' + 1, cons2_var')) (state) ELSE expr_2_super(memory_write(pm')(waddr', cons1_var'))(s') ENDCASES)</td> </tr> </table>		{1}	result_pred(states') (CASES expr_2_super(memory_write(pm')(waddr', cons1_var'))(s') OF OK(state): expr_2_super(memory_write_list_nse(pm')(waddr' + 1, cons2_var')) (state) ELSE expr_2_super(memory_write(pm')(waddr', cons1_var'))(s') ENDCASES)
{1}	result_pred(states') (CASES expr_2_super(memory_write(pm')(waddr', cons1_var'))(s') OF OK(state): expr_2_super(memory_write_list_nse(pm')(waddr' + 1, cons2_var')) (state) ELSE expr_2_super(memory_write(pm')(waddr', cons1_var'))(s') ENDCASES)		

Simplifying, rewriting, and recording with decision procedures,

Rewriting using -3, matching in *,

we get 2 subgoals:

C Proof scripts

transformer_invariant_write_list_nse.2.1.1:

<div style="display: flex; flex-direction: column; gap: 5px;"> <div style="display: flex; align-items: flex-start;"> {-1} $OK?(expr_2_super(memory_write(pm')(waddr', cons1_var'))(s'))$ </div> <div style="display: flex; align-items: flex-start;"> {-2} $memory_write_transformers(pm', address_block(waddr', 1 + length(cons2_var'))$ $(expr_2_super(memory_write(pm')(waddr', cons1_var')))$ </div> <div style="display: flex; align-items: flex-start;"> {-3} $\forall (states: PRED[State], waddr: Address, s: State):$ $transformer_invariant?(states,$ $memory_write_transformers(pm',$ $address_block(waddr,$ $length(cons2_var'))$ $\wedge states(s)$ \supset $result_pred(states)$ $(expr_2_super(memory_write_list_nse(pm')(waddr, cons2_var'))(s))$ </div> <div style="display: flex; align-items: flex-start;"> {-4} $transformer_invariant?(states',$ $memory_write_transformers(pm',$ $address_block(waddr',$ $1 + length(cons2_va$ </div> <div style="display: flex; align-items: flex-start;"> {-5} $states'(s')$ </div> </div> <hr style="border: 0.5px solid black; margin: 5px 0;"/> <div style="display: flex; flex-direction: column; gap: 5px;"> <div style="display: flex; align-items: flex-start;"> {1} $transformer_invariant?(states',$ $memory_write_transformers(pm',$ $address_block(waddr' + 1,$ $length(cons2_var'))$ </div> <div style="display: flex; align-items: flex-start;"> {2} $result_pred(states')$ $(expr_2_super(memory_write_list_nse(pm')(waddr' + 1, cons2_var'))$ $(state(expr_2_super(memory_write(pm')(waddr', cons1_var'))$ $(s'))))$ </div> </div>	
--	--

Keeping (-4 1) and hiding *,

Installing automatic rewrites from: address_block_subset_1 memory_write_transformers_mono

Using lemma transformer_invariant_mono_transformers [State],

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of transformer_invariant_write_list_nse.2.1.1.

transformer_invariant_write_list_nse.2.1.2:

<pre> {-1} OK?(expr_2_super(memory_write(pm')(waddr', cons1_var'))(s')) {-2} memory_write_transformers(pm', address_block(waddr', 1 + length(cons2_var'))) (expr_2_super(memory_write(pm')(waddr', cons1_var'))) {-3} ∀ (states: PRED[State], waddr: Address, s: State): transformer_invariant?(states, memory_write_transformers(pm', address_block(waddr, length(cons2_var')))) ∧ states(s) ⊃ result_pred(states) (expr_2_super(memory_write_list_nse(pm')(waddr, cons2_var'))(s)) {-4} transformer_invariant?(states', memory_write_transformers(pm', address_block(waddr', 1 + length(cons2_var')))) {-5} states'(s') </pre>	<pre> {1} states'(state(expr_2_super(memory_write(pm')(waddr', cons1_var'))(s'))) {2} result_pred(states') (expr_2_super(memory_write_list_nse(pm')(waddr' + 1, cons2_var')) (state(expr_2_super(memory_write(pm')(waddr', cons1_var')) (s')))) </pre>
---	---

Using lemma super_transformer_invariant_next_ok[State],

Installing automatic rewrites from: has_next_state

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of transformer_invariant_write_list_nse.2.1.2.

transformer_invariant_write_list_nse.2.2:

<pre> {-1} memory_write_transformers(pm', address_block(waddr', 1 + length(cons2_var'))) (expr_2_super(memory_write(pm')(waddr', cons1_var'))) {-2} ∀ (states: PRED[State], waddr: Address, s: State): transformer_invariant?(states, memory_write_transformers(pm', address_block(waddr, length(cons2_var')))) ∧ states(s) ⊃ result_pred(states) (expr_2_super(memory_write_list_nse(pm')(waddr, cons2_var'))(s)) {-3} transformer_invariant?(states', memory_write_transformers(pm', address_block(waddr', 1 + length(cons2_var')))) {-4} states'(s') </pre>	<pre> {1} OK?(expr_2_super(memory_write(pm')(waddr', cons1_var'))(s')) {2} result_pred(states') (CASES expr_2_super(memory_write(pm')(waddr', cons1_var'))(s') OF OK(state): expr_2_super(memory_write_list_nse(pm')(waddr' + 1, cons2_var')) (state) ELSE expr_2_super(memory_write(pm')(waddr', cons1_var'))(s') ENDCASES) </pre>
---	---

Hiding formulas: -2,
Simplifying, rewriting, and recording with decision procedures,
Expanding the definition of transformer_invariant?,
Instantiating quantified variables,
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of transformer_invariant_write_list_nse.2.2.
Q.E.D.

C.123.6 Memory_Change_2.transformer_invariant_write_list

Terse proof for transformer_invariant_write_list.

transformer_invariant_write_list:

<pre>{1} ∃ (pm: Memory_struct[State], states: PRED[State], waddr: Ad- dress, bl: list[Byte], s: State): transformer_invariant?(states, (memory_write_transformers(pm, address_block(waddr, length(bl))) ^ side_effect_content_unchanged(address_block(waddr, length(bl)), states, memory_write_side_effect(pm)) ^ states(s) ⊃ result_pred(states)(expr_2_super(memory_write_list(pm)(waddr, bl))(s))</pre>
--

Repeatedly Skolemizing and flattening,
Expanding the definition of memory_write_list,
Rewriting using super_comp_expr_expr, matching in *,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
we get 2 subgoals:

transformer_invariant_write_list.1:

<pre>{-1} every(λ (x: number): number_field_pred(x) ∧ real_pred(x) ∧ rational_pred(x) ∧ integer_pred(x) ∧ x ≥ 0 ∧ x < max_byte) (bl') {-2} transformer_invariant?(states', (memory_write_transformers(pm', address_block(waddr', length(bl')) ^ side_effect_content_unchanged(address_block(waddr', length(bl')), states', memory_write_side_effect(pm')) ^ states'(s') ^ OK?(expr_2_super(memory_write_side_effect(pm')(waddr', bl', FALSE))(s'))</pre>
<pre>{1} result_pred(states') (expr_2_super(memory_write_list_nse(pm') (waddr', data(memory_write_side_effect(pm') (waddr', bl', FALSE)(s')) (state(expr_2_super(memory_write_side_effect(pm') (waddr', bl', FALSE) (s')))))</pre>

Using lemma side_effect_content_unchanged_content [State],
Installing automatic rewrites from: subset_equal ok_expr_2_super

Simplifying, rewriting, and recording with decision procedures,

Replacing using formula -1,

Hiding formulas: -1,

Rewriting using transformer_invariant_write_list_nse, matching in *,

we get 2 subgoals:

transformer_invariant_write_list.1.1:

{-1}	every(λ (x : number): number_field_pred(x) \wedge real_pred(x) \wedge rational_pred(x) \wedge integer_pred(x) \wedge $x \geq 0 \wedge x < \text{max_byte}$) (bl')
{-2}	transformer_invariant?(states', (memory_write_transformers(pm', address_block(waddr', length(bl')))) \cup {expr_2.
{-3}	side_effect_content_unchanged(address_block(waddr', length(bl')), states', memory_write_side_effect(pm'))
{-4}	states'(s')
{-5}	OK?(memory_write_side_effect(pm')(waddr', bl', FALSE)(s'))
<hr/>	
{1}	transformer_invariant?(states', memory_write_transformers(pm', ad- dress_block(waddr', length(bl'))))
{2}	result_pred(states') (expr_2_super(memory_write_list_nse(pm')(waddr', bl')) (state(expr_2_super(memory_write_side_effect(pm') (waddr', bl', FALSE)) (s'))))

Keeping (-2 1) and hiding *,

Rewriting using add_as_union, matching in *,

Using lemma transformer_invariant_mono_transformers,

Simplifying, rewriting, and recording with decision procedures,

Rewriting using union_subset1, matching in *,

This completes the proof of **transformer_invariant_write_list.1.1**.

transformer_invariant_write_list.1.2:

{-1}	every(λ (x : number): number_field_pred(x) \wedge real_pred(x) \wedge rational_pred(x) \wedge integer_pred(x) \wedge $x \geq 0 \wedge x < \text{max_byte}$) (bl')
{-2}	transformer_invariant?(states', (memory_write_transformers(pm', address_block(waddr', length(bl')))) \cup {expr_2.
{-3}	side_effect_content_unchanged(address_block(waddr', length(bl')), states', memory_write_side_effect(pm'))
{-4}	states'(s')
{-5}	OK?(memory_write_side_effect(pm')(waddr', bl', FALSE)(s'))
<hr/>	
{1}	states'(state(expr_2_super(memory_write_side_effect(pm')(waddr', bl', FALSE))(s')))
{2}	result_pred(states') (expr_2_super(memory_write_list_nse(pm')(waddr', bl')) (state(expr_2_super(memory_write_side_effect(pm') (waddr', bl', FALSE)) (s'))))

Hiding formulas: -1, -3, 2,

C Proof scripts

Using lemma `super_transformer_invariant_next_ok[State]`,
Installing automatic rewrites from: `has_next_state` add
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `transformer_invariant_write_list.1.2`.
`transformer_invariant_write_list.2`:

{-1}	every(λ (x : number): number_field_pred(x) \wedge real_pred(x) \wedge rational_pred(x) \wedge integer_pred(x) \wedge $x \geq 0 \wedge x < \text{max_byte}$) (bl')
{-2}	transformer_invariant?(states', (memory_write_transformers(pm', address_block(waddr', length(bl'))
{-3}	side_effect_content_unchanged(address_block(waddr', length(bl')), states', memory_write_side_effect(pm'))
{-4}	states'(s')
{1}	OK?(expr_2_super(memory_write_side_effect(pm')(waddr', bl', FALSE))(s'))
{2}	result_pred(states') (expr_2_super(memory_write_side_effect(pm')(waddr', bl', FALSE))(s'))

Hiding formulas: -1, -3,
Expanding the definition of `transformer_invariant?`,
Instantiating quantified variables,
Simplifying, rewriting, and recording with decision procedures,
Expanding the definition of `add`,
which is trivially true.
This completes the proof of `transformer_invariant_write_list.2`.
Q.E.D.

C.123.7 Memory_Change_2.changed_memory_invariant?_TCC1

Terse proof for `changed_memory_invariant?_TCC1`.
`changed_memory_invariant?_TCC1`:

{1}	\forall (pm: Memory_struct[State], states: PRED[State], addresses: PRED[Address]): transformer_invariant?(states, memory_write_transformers(pm, addresses)) \wedge transformer_invariant?(states, memory_read_transformers(pm, addresses)) \supset (\forall (s: State, a: Address, b: Byte): OK?(memory_write(pm)(a, b)(s)) \wedge addresses(a) \wedge states(s) \supset OK?[State, Unit](memory_write(pm)(a, b)(s)) \vee Exception?[State, Unit](memory_write(pm)(a, b)(s)))
-----	---

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `changed_memory_invariant?_TCC1`.
Q.E.D.

C.123.8 Memory_Change_2.changed_memory_invariant_mono

Terse proof for `changed_memory_invariant_mono`.

changed_memory_invariant_mono:

$\{1\} \quad \forall (\text{pm}: \text{Memory_struct}[\text{State}], \text{states}: \text{PRED}[\text{State}], \\ \text{addresses_1}, \text{addresses_2}: \text{PRED}[\text{Address}]): \\ (\text{addresses_1} \subseteq \text{addresses_2}) \wedge \text{changed_memory_invariant?}(\text{pm}, \text{states}, \text{ad-} \\ \text{resses_2}) \supset \\ \text{changed_memory_invariant?}(\text{pm}, \text{states}, \text{addresses_1})$

Repeatedly Skolemizing and flattening,

Expanding the definition of changed_memory_invariant?,

Applying disjunctive simplification to flatten sequent,

Applying propositional simplification,

we get 3 subgoals:

changed_memory_invariant_mono.1:

$\{-1\} \quad (\text{addresses}' \subseteq \text{addresses}'')$ $\{-2\} \quad \text{transformer_invariant?}(\text{states}', \text{memory_read_transformers}(\text{pm}', \text{addresses}''))$ $\{-3\} \quad \text{transformer_invariant?}(\text{states}', \text{memory_write_transformers}(\text{pm}', \text{addresses}''))$ $\{-4\} \quad \forall (s: \text{State}, a: \text{Address}, b: \text{Byte}): \\ \text{states}'(s) \wedge \\ \text{addresses}''(a) \wedge \\ \text{OK?}(\text{memory_write}(\text{pm}')(\text{a}, \text{b})(\text{s})) \wedge \\ \text{OK?}(\text{memory_read}(\text{pm}')(\text{a})(\text{state}(\text{memory_write}(\text{pm}')(\text{a}, \text{b})(\text{s})))) \\ \supset \text{data}(\text{memory_read}(\text{pm}')(\text{a})(\text{state}(\text{memory_write}(\text{pm}')(\text{a}, \text{b})(\text{s})))) = \text{b}$ <hr/> $\{1\} \quad \text{transformer_invariant?}(\text{states}', \text{memory_read_transformers}(\text{pm}', \text{addresses}'))$

Hiding formulas: -3, -4,

Installing automatic rewrites from: memory_read_transformers_mono

Using lemma transformer_invariant_mono_transformers,

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of changed_memory_invariant_mono.1.

changed_memory_invariant_mono.2:

$\{-1\} \quad (\text{addresses}' \subseteq \text{addresses}'')$ $\{-2\} \quad \text{transformer_invariant?}(\text{states}', \text{memory_read_transformers}(\text{pm}', \text{addresses}''))$ $\{-3\} \quad \text{transformer_invariant?}(\text{states}', \text{memory_write_transformers}(\text{pm}', \text{addresses}''))$ $\{-4\} \quad \forall (s: \text{State}, a: \text{Address}, b: \text{Byte}): \\ \text{states}'(s) \wedge \\ \text{addresses}''(a) \wedge \\ \text{OK?}(\text{memory_write}(\text{pm}')(\text{a}, \text{b})(\text{s})) \wedge \\ \text{OK?}(\text{memory_read}(\text{pm}')(\text{a})(\text{state}(\text{memory_write}(\text{pm}')(\text{a}, \text{b})(\text{s})))) \\ \supset \text{data}(\text{memory_read}(\text{pm}')(\text{a})(\text{state}(\text{memory_write}(\text{pm}')(\text{a}, \text{b})(\text{s})))) = \text{b}$ <hr/> $\{1\} \quad \text{transformer_invariant?}(\text{states}', \text{memory_write_transformers}(\text{pm}', \text{addresses}'))$
--

Hiding formulas: -2, -4,

Installing automatic rewrites from: memory_write_transformers_mono

Using lemma transformer_invariant_mono_transformers,

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of changed_memory_invariant_mono.2.

changed_memory_invariant_mono.3:

<pre> {-1} (addresses' ⊆ addresses'') {-2} transformer_invariant?(states', memory_read_transformers(pm', addresses'')) {-3} transformer_invariant?(states', memory_write_transformers(pm', addresses'')) {-4} ∀ (s: State, a: Address, b: Byte): states'(s) ∧ addresses''(a) ∧ OK?(memory_write(pm')(a, b)(s)) ∧ OK?(memory_read(pm')(a)(state(memory_write(pm')(a, b)(s)))) ⊃ data(memory_read(pm')(a)(state(memory_write(pm')(a, b)(s)))) = b </pre>	<hr style="border: 0.5px solid black;"/> <pre> {1} ∀ (s: State, a: Address, b: Byte): states'(s) ∧ addresses'(a) ∧ OK?(memory_write(pm')(a, b)(s)) ∧ OK?(memory_read(pm')(a)(state(memory_write(pm')(a, b)(s)))) ⊃ data(memory_read(pm')(a)(state(memory_write(pm')(a, b)(s)))) = b </pre>
--	--

Hiding formulas: -2, -3,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `changed_memory_invariant_mono.3`.

Q.E.D.

C.123.9 Memory_Change_2.memory_read_list_nse_ok

Terse proof for `memory_read_list_nse_ok`.

`memory_read_list_nse_ok`:

<pre> {1} ∀ (pm: Memory_struct [State], states: PRED [State], raddr: Ad- dress, size: nat, s: State): transformer_invariant?(states, memory_read_transformers(pm, ad- dress_block(raddr, size))) ∧ transformers_ok?(states, memory_read_transformers(pm, ad- dress_block(raddr, size))) ∧ states(s) ⊃ OK?(memory_read_list_nse(pm)(raddr, size)(s)) </pre>	<hr style="border: 0.5px solid black;"/>
--	--

For the top quantifier in 1, we introduce Skolem constants: `(pm' states' _ _)`,

Inducting on size on formula 1,

we get 2 subgoals:

`memory_read_list_nse_ok.1`:

<pre> {1} ∀ (raddr: Address, s: State): transformer_invariant?(states', memory_read_transformers(pm', ad- dress_block(raddr, 0))) ∧ transformers_ok?(states', memory_read_transformers(pm', ad- dress_block(raddr, 0))) ∧ states'(s) ⊃ OK?(memory_read_list_nse(pm')(raddr, 0)(s)) </pre>	<hr style="border: 0.5px solid black;"/>
---	--

Expanding the definition of `memory_read_list_nse`,

Expanding the definition of `ok_result`,

which is trivially true.

This completes the proof of `memory_read_list_nse_ok.1`.

`memory_read_list_nse_ok.2`:

<pre> {1} ∀ j: (∀ (raddr: Address, s: State): transformer_invariant?(states', memory_read_transformers(pm', ad- dress_block(raddr, j))) ∧ transformers_ok?(states', memory_read_transformers(pm', ad- dress_block(raddr, j))) ∧ states'(s) ⊃ OK?(memory_read_list_nse(pm')(raddr, j)(s))) ⊃ (∀ (raddr: Address, s: State): transformer_invariant?(states', memory_read_transformers(pm', ad- dress_block(raddr, j + 1))) ∧ transformers_ok?(states', memory_read_transformers(pm', ad- dress_block(raddr, j + 1))) ∧ states'(s) ⊃ OK?(memory_read_list_nse(pm')(raddr, j + 1)(s))) </pre>
--

Repeatedly Skolemizing and flattening,

Installing automatic rewrites from: `memory_read_transformers_memory_read address_block`

Expanding the definition of `memory_read_list_nse`,

Using lemma `expr_transformers_ok_ok[State, Byte]`,

Simplifying, rewriting, and recording with decision procedures,

Expanding the definition of `##`,

Instantiating quantified variables,

Applying propositional simplification,

we get 4 subgoals:

C Proof scripts

memory_read_list_nse_ok.2.1:

{-1}	OK?(memory_read_list_nse(pm')(raddr' + 1, j') (state(memory_read(pm')(raddr')(s'))))
{-2}	OK?(memory_read(pm')(raddr')(s'))
{-3}	$j' \geq 0$
{-4}	transformer_invariant?(states', memory_read_transformers(pm', ad- dress_block(raddr', 1 + j')))
{-5}	transformers_ok?(states', memory_read_transformers(pm', address_block(raddr', 1 + j')))
{-6}	states'(s')
{1}	OK?(CASES memory_read_list_nse(pm')(raddr' + 1, j') (state(memory_read(pm')(raddr')(s')))) OF OK(state_1, data_2): ok_result(cons(data(memory_read(pm')(raddr')(s')), data_2))(state_1), Exception(ex_type, state_1): Exception(ex_type, state_1), Fatal: Fatal, Hang: Hang ENDCASES)

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Expanding the definition of ok_result,
which is trivially true.

This completes the proof of memory_read_list_nse_ok.2.1.

memory_read_list_nse_ok.2.2:

{-1}	OK?(memory_read(pm')(raddr')(s'))
{-2}	$j' \geq 0$
{-3}	transformer_invariant?(states', memory_read_transformers(pm', ad- dress_block(raddr', 1 + j')))
{-4}	transformers_ok?(states', memory_read_transformers(pm', address_block(raddr', 1 + j')))
{-5}	states'(s')
{1}	transformer_invariant?(states', memory_read_transformers(pm', ad- dress_block(raddr' + 1, j')))
{2}	OK?(CASES memory_read_list_nse(pm')(raddr' + 1, j') (state(memory_read(pm')(raddr')(s')))) OF OK(state_1, data_2): ok_result(cons(data(memory_read(pm')(raddr')(s')), data_2))(state_1), Exception(ex_type, state_1): Exception(ex_type, state_1), Fatal: Fatal, Hang: Hang ENDCASES)

Keeping (-3 1) and hiding *,

Installing automatic rewrites from: memory_read_transformers_mono address_block_subset_1

Using lemma transformer_invariant_mono_transformers,

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of memory_read_list_nse_ok.2.2.

memory_read_list_nse_ok.2.3:

{-1}	OK?(memory_read(pm')(raddr')(s'))
{-2}	$j' \geq 0$
{-3}	transformer_invariant?(states', memory_read_transformers(pm', address_block(raddr', 1 + j')))
{-4}	transformers_ok?(states', memory_read_transformers(pm', address_block(raddr', 1 + j')))
{-5}	states'(s')
{1}	transformers_ok?(states', memory_read_transformers(pm', address_block(raddr' + 1, j')))
{2}	OK?(CASES memory_read_list_nse(pm')(raddr' + 1, j') (state(memory_read(pm')(raddr')(s'))) OF OK(state_1, data_2): ok_result(cons(data(memory_read(pm')(raddr')(s')), data_2))(state_1), Exception(ex_type, state_1): Exception(ex_type, state_1), Fatal: Fatal, Hang: Hang ENDCASES)

Keeping (-4 1) and hiding *,

Installing automatic rewrites from: memory_read_transformers_mono address_block_subset_1

Using lemma transformers_ok_mono_transformers,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of memory_read_list_nse_ok.2.3.

memory_read_list_nse_ok.2.4:

{-1}	OK?(memory_read(pm')(raddr')(s'))
{-2}	$j' \geq 0$
{-3}	transformer_invariant?(states', memory_read_transformers(pm', address_block(raddr', 1 + j')))
{-4}	transformers_ok?(states', memory_read_transformers(pm', address_block(raddr', 1 + j')))
{-5}	states'(s')
{1}	states'(state(memory_read(pm')(raddr')(s')))
{2}	OK?(CASES memory_read_list_nse(pm')(raddr' + 1, j') (state(memory_read(pm')(raddr')(s'))) OF OK(state_1, data_2): ok_result(cons(data(memory_read(pm')(raddr')(s')), data_2))(state_1), Exception(ex_type, state_1): Exception(ex_type, state_1), Fatal: Fatal, Hang: Hang ENDCASES)

Hiding formulas: 2,

Using lemma expr_transformer_invariant_next_ok [State, Byte],

Simplifying, rewriting, and recording with decision procedures,

Expanding the definition of has_next_state,

which is trivially true.

This completes the proof of memory_read_list_nse_ok.2.4.

Q.E.D.

C.123.10

Memory_Change_2.memory_read_list_nse_next_length_TCC1

Terse proof for memory_read_list_nse_next_length_TCC1.

memory_read_list_nse_next_length_TCC1:

$$\{1\} \quad \forall (pm: \text{Memory_struct}[\text{State}], \text{states}: \text{PRED}[\text{State}], \text{raddr}: \text{Address}, \text{size}: \text{nat}, s: \text{State}):$$

$$\quad \text{transformer_invariant?}(\text{states}, \text{memory_read_transformers}(pm, \text{address_block}(\text{raddr}, \text{size}))) \wedge$$

$$\quad \text{transformers_ok?}(\text{states}, \text{memory_read_transformers}(pm, \text{address_block}(\text{raddr}, \text{size}))) \wedge$$

$$\quad \text{states}(s)$$

$$\quad \supset$$

$$\quad \text{OK?}[\text{State}, \text{list}[\text{Byte}]](\text{memory_read_list_nse}[\text{State}](pm)(\text{raddr}, \text{size})(s)) \vee$$

$$\quad \text{Exception?}[\text{State}, \text{list}[\text{Byte}]](\text{memory_read_list_nse}[\text{State}](pm)(\text{raddr}, \text{size})(s))$$

Repeatedly Skolemizing and flattening,

Using lemma memory_read_list_nse_ok,

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of memory_read_list_nse_next_length_TCC1.

Q.E.D.

C.123.11

Memory_Change_2.memory_read_list_nse_next_length_TCC2

Terse proof for memory_read_list_nse_next_length_TCC2.

memory_read_list_nse_next_length_TCC2:

$$\{1\} \quad \forall (pm: \text{Memory_struct}[\text{State}], \text{states}: \text{PRED}[\text{State}], \text{raddr}: \text{Address}, \text{size}: \text{nat}, s: \text{State}):$$

$$\quad \text{states}(\text{state}(\text{memory_read_list_nse}(pm)(\text{raddr}, \text{size})(s))) \wedge$$

$$\quad \text{transformer_invariant?}(\text{states}, \text{memory_read_transformers}(pm, \text{address_block}(\text{raddr}, \text{size})))$$

$$\quad \wedge$$

$$\quad \text{transformers_ok?}(\text{states}, \text{memory_read_transformers}(pm, \text{address_block}(\text{raddr}, \text{size}))) \wedge$$

$$\quad \text{states}(s)$$

$$\quad \supset \text{OK?}[\text{State}, \text{list}[\text{Byte}]](\text{memory_read_list_nse}[\text{State}](pm)(\text{raddr}, \text{size})(s))$$

Repeatedly Skolemizing and flattening,

Using lemma memory_read_list_nse_ok,

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of memory_read_list_nse_next_length_TCC2.

Q.E.D.

C.123.12 Memory_Change_2.memory_read_list_nse_next_length

Terse proof for memory_read_list_nse_next_length.

memory_read_list_nse_next_length:

```
{1}  ∀ (pm: Memory_struct[State], states: PRED[State], raddr: Address, size: nat, s: State):
      transformer_invariant?(states, memory_read_transformers(pm, address_block(raddr, size))) ∧
      transformers_ok?(states, memory_read_transformers(pm, address_block(raddr, size))) ∧
      states(s)
      ⊃
      states(state(memory_read_list_nse(pm)(raddr, size)(s))) ∧
      length(data(memory_read_list_nse(pm)(raddr, size)(s))) = size
```

For the top quantifier in 1, we introduce Skolem constants: (pm' states' _ _ _),

Inducting on size on formula 1,

we get 4 subgoals:

memory_read_list_nse_next_length.1:

```
{1}  ∀ (raddr: Address, s: State):
      transformer_invariant?(states', memory_read_transformers(pm', address_block(raddr, 0)))
      ∧
      transformers_ok?(states', memory_read_transformers(pm', address_block(raddr, 0))) ∧
      states'(s)
      ⊃
      states'(state(memory_read_list_nse(pm')(raddr, 0)(s))) ∧
      length(data(memory_read_list_nse(pm')(raddr, 0)(s))) = 0
```

Expanding the definition of memory_read_list_nse,

Repeatedly Skolemizing and flattening,

Expanding the definition of ok_result,

Expanding the definition of length,

which is trivially true.

This completes the proof of memory_read_list_nse_next_length.1.

memory_read_list_nse_next_length.2:

$$\begin{array}{l}
 \{1\} \quad \forall j: \\
 \quad (\forall (\text{raddr}: \text{Address}, s: \text{State}): \\
 \quad \quad \text{transformer_invariant?}(\text{states}', \\
 \quad \quad \quad \text{memory_read_transformers}(\text{pm}', \text{ad-} \\
 \quad \quad \text{dress_block}(\text{raddr}, j))) \\
 \quad \quad \wedge \\
 \quad \quad \text{transformers_ok?}(\text{states}', \text{memory_read_transformers}(\text{pm}', \text{ad-} \\
 \quad \quad \text{dress_block}(\text{raddr}, j))) \\
 \quad \quad \wedge \text{states}'(s) \\
 \quad \quad \supset \\
 \quad \quad \text{states}'(\text{state}(\text{memory_read_list_nse}(\text{pm}')(\text{raddr}, j)(s))) \wedge \\
 \quad \quad \text{length}(\text{data}(\text{memory_read_list_nse}(\text{pm}')(\text{raddr}, j)(s))) = j \\
 \quad \quad \supset \\
 \quad (\forall (\text{raddr}: \text{Address}, s: \text{State}): \\
 \quad \quad \text{transformer_invariant?}(\text{states}', \\
 \quad \quad \quad \text{memory_read_transformers}(\text{pm}', \text{ad-} \\
 \quad \quad \text{dress_block}(\text{raddr}, j + 1))) \\
 \quad \quad \wedge \\
 \quad \quad \text{transformers_ok?}(\text{states}', \\
 \quad \quad \quad \text{memory_read_transformers}(\text{pm}', \text{ad-} \\
 \quad \quad \text{dress_block}(\text{raddr}, j + 1))) \\
 \quad \quad \wedge \text{states}'(s) \\
 \quad \quad \supset \\
 \quad \quad \text{states}'(\text{state}(\text{memory_read_list_nse}(\text{pm}')(\text{raddr}, j + 1)(s))) \wedge \\
 \quad \quad \text{length}(\text{data}(\text{memory_read_list_nse}(\text{pm}')(\text{raddr}, j + 1)(s))) = \\
 \quad \quad j + 1)
 \end{array}$$

Repeatedly Skolemizing and flattening,

Using lemma `memory_read_list_nse_ok`,

Simplifying, rewriting, and recording with decision procedures,

Installing automatic rewrites from: `memory_read_transformers_memory_read address_block`

Using lemma `expr_transformers_ok_ok`,

Simplifying, rewriting, and recording with decision procedures,

Instantiating the top quantifier in -4 with the terms: `raddr' + 1, state(memory_read(pm')(raddr')(s'))`,

Splitting conjunctions,

we get 4 subgoals:

memory_read_list_nse_next_length.2.1:

<pre> {-1} states'(state(memory_read_list_nse(pm')(raddr' + 1, j') (state(memory_read(pm')(raddr')(s'))))) ^ length(data(memory_read_list_nse(pm')(raddr' + 1, j') (state(memory_read(pm')(raddr')(s'))))) = j' {-2} OK?(memory_read(pm')(raddr')(s')) {-3} OK?(memory_read_list_nse(pm')(raddr', 1 + j')(s')) {-4} j' ≥ 0 {-5} transformer_invariant?(states', memory_read_transformers(pm', ad- dress_block(raddr', 1 + j')) {-6} transformers_ok?(states', memory_read_transformers(pm', address_block(raddr', 1 + j')) {-7} states'(s') </pre>	<hr style="border: 0.5px solid black;"/> <pre> {1} states'(state(memory_read_list_nse(pm')(raddr', 1 + j')(s')) ^ length(data(memory_read_list_nse(pm')(raddr', 1 + j')(s')) = 1 + j' </pre>
---	---

Applying disjunctive simplification to flatten sequent,

Installing automatic rewrites from: length ##! ok_result

Expanding the definition of memory_read_list_nse,

Expanding the definition of memory_read_list_nse,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of memory_read_list_nse_next_length.2.1.

memory_read_list_nse_next_length.2.2:

<pre> {-1} OK?(memory_read(pm')(raddr')(s')) {-2} OK?(memory_read_list_nse(pm')(raddr', 1 + j')(s')) {-3} j' ≥ 0 {-4} transformer_invariant?(states', memory_read_transformers(pm', ad- dress_block(raddr', 1 + j')) {-5} transformers_ok?(states', memory_read_transformers(pm', address_block(raddr', 1 + j')) {-6} states'(s') </pre>	<hr style="border: 0.5px solid black;"/> <pre> {1} transformer_invariant?(states', memory_read_transformers(pm', ad- dress_block(raddr' + 1, j')) {2} states'(state(memory_read_list_nse(pm')(raddr', 1 + j')(s')) ^ length(data(memory_read_list_nse(pm')(raddr', 1 + j')(s')) = 1 + j' </pre>
--	---

Installing automatic rewrites from: address_block_subset_1 memory_read_transformers_mono

Using lemma transformer_invariant_mono_transformers,

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of memory_read_list_nse_next_length.2.2.

C Proof scripts

memory_read_list_nse_next_length.2.3:

{-1}	OK?(memory_read(pm')(raddr')(s'))
{-2}	OK?(memory_read_list_nse(pm')(raddr', 1 + j')(s'))
{-3}	$j' \geq 0$
{-4}	transformer_invariant?(states', memory_read_transformers(pm', address_block(raddr', 1 + j')))
{-5}	transformers_ok?(states', memory_read_transformers(pm', address_block(raddr', 1 + j')))
{-6}	states'(s')
{1}	transformers_ok?(states', memory_read_transformers(pm', address_block(raddr' + 1, j')))
{2}	states'(state(memory_read_list_nse(pm')(raddr', 1 + j')(s'))) \wedge length(data(memory_read_list_nse(pm')(raddr', 1 + j')(s'))) = 1 + j'

Installing automatic rewrites from: address_block_subset_1 memory_read_transformers_mono

Using lemma transformers_ok_mono_transformers,

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of memory_read_list_nse_next_length.2.3.

memory_read_list_nse_next_length.2.4:

{-1}	OK?(memory_read(pm')(raddr')(s'))
{-2}	OK?(memory_read_list_nse(pm')(raddr', 1 + j')(s'))
{-3}	$j' \geq 0$
{-4}	transformer_invariant?(states', memory_read_transformers(pm', address_block(raddr', 1 + j')))
{-5}	transformers_ok?(states', memory_read_transformers(pm', address_block(raddr', 1 + j')))
{-6}	states'(s')
{1}	states'(state(memory_read(pm')(raddr')(s')))
{2}	states'(state(memory_read_list_nse(pm')(raddr', 1 + j')(s'))) \wedge length(data(memory_read_list_nse(pm')(raddr', 1 + j')(s'))) = 1 + j'

Hiding formulas: 2,

Using lemma expr_transformer_invariant_next_ok,

Simplifying, rewriting, and recording with decision procedures,

Expanding the definition of has_next_state,

which is trivially true.

This completes the proof of memory_read_list_nse_next_length.2.4.

memory_read_list_nse_next_length.3:

```

{1}  ∀ (size: nat, raddr: Address, s: State):
      states'(state(memory_read_list_nse(pm')(raddr, size)(s))) ∧
      transformer_invariant?(states',
                             memory_read_transformers(pm', ad-
address_block(raddr, size)))
      ∧
      transformers_ok?(states', memory_read_transformers(pm', ad-
address_block(raddr, size))) ∧
      states'(s)
      ⊃ OK?[State, list[Byte]](memory_read_list_nse[State](pm')(raddr, size)(s))
{2}  ∀ (raddr: Address, s: State):
      transformer_invariant?(states',
                             memory_read_transformers(pm', ad-
address_block(raddr, size')))
      ∧
      transformers_ok?(states', memory_read_transformers(pm', ad-
address_block(raddr, size')))
      ∧ states'(s)
      ⊃
      states'(state(memory_read_list_nse(pm')(raddr, size')(s))) ∧
      length(data(memory_read_list_nse(pm')(raddr, size')(s))) = size'

```

Hiding formulas: 2,

Repeatedly Skolemizing and flattening,

Using lemma memory_read_list_nse_ok,

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of memory_read_list_nse_next_length.3.

memory_read_list_nse_next_length.4:

```

{1}  ∀ (size: nat, raddr: Address, s: State):
      transformer_invariant?(states',
                             memory_read_transformers(pm', ad-
                             dress_block(raddr, size)))
      ∧
      transformers_ok?(states', memory_read_transformers(pm', ad-
      dress_block(raddr, size))) ∧
      states'(s)
      ⊃
      OK?[State, list[Byte]](memory_read_list_nse[State](pm')(raddr, size)(s)) ∨
      Exception?[State, list[Byte]](memory_read_list_nse[State](pm')(raddr, size)(s))
{2}  ∀ (raddr: Address, s: State):
      transformer_invariant?(states',
                             memory_read_transformers(pm', ad-
                             dress_block(raddr, size')))
      ∧
      transformers_ok?(states', memory_read_transformers(pm', ad-
      dress_block(raddr, size')))
      ∧ states'(s)
      ⊃
      states'(state(memory_read_list_nse(pm')(raddr, size')(s))) ∧
      length(data(memory_read_list_nse(pm')(raddr, size')(s))) = size'

```

Hiding formulas: 2,

Repeatedly Skolemizing and flattening,

Using lemma memory_read_list_nse_ok,

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of memory_read_list_nse_next_length.4.

Q.E.D.

C.123.13 Memory_Change_2.memory_read_list_ok

Terse proof for memory_read_list_ok.

memory_read_list_ok:

```

{1}  ∀ (pm: Memory_struct[State], states: PRED[State], raddr: Ad-
      dress, size: nat, s: State):
      transformer_invariant?(states,
                             (memory_read_transformers(pm, address_block(raddr, size)) ∪ mem
      ∧
      transformers_ok?(states,
                             (memory_read_transformers(pm, address_block(raddr, size)) ∪ memory_re
      ∧ states(s)
      ⊃ OK?(memory_read_list(pm)(raddr, size)(s))

```

Repeatedly Skolemizing and flattening,

Installing automatic rewrites from: union_subset1

Expanding the definition of memory_read_list,

Using lemma memory_read_list_nse_ok,

Using lemma memory_read_list_nse_next_length,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 3 subgoals:

memory_read_list_ok.1:

<div style="display: flex; flex-direction: column; gap: 5px;"> <div style="border-bottom: 1px solid black; padding-bottom: 5px;"> <p>{-1} states'(s')</p> <p>{-2} states'(state(memory_read_list_nse(pm')(raddr', size')(s')))</p> <p>{-3} length(data(memory_read_list_nse(pm')(raddr', size')(s'))) = size'</p> <p>{-4} OK?(memory_read_list_nse(pm')(raddr', size')(s'))</p> <p>{-5} size' ≥ 0</p> <p>{-6} transformer_invariant?(states',</p> </div> <div style="padding: 5px;"> <p>(memory_read_transformers(pm', address_block(raddr', size')) ∪ memory_read_side_effect_transformers(pm', address_block(raddr', size')))</p> </div> <div style="border-bottom: 1px solid black; padding-bottom: 5px;"> <p>{-7} transformers_ok?(states',</p> <p>(memory_read_transformers(pm', address_block(raddr', size')) ∪ memory_read_side_effect_transformers(pm', address_block(raddr', size')))</p> </div> </div>	<div style="border-bottom: 1px solid black; padding-bottom: 5px;"> <p>{1} OK?((memory_read_list_nse(pm')(raddr', size') ##</p> <p>(λ (bl1: list[Byte]): memory_read_side_effect(pm')(raddr', bl1, FALSE)))</p> <p>(s'))</p> </div>
--	--

Expanding the definition of ##,

Hiding formulas: -2, -5, -6,

Using lemma expr_transformers_ok_ok,

Simplifying, rewriting, and recording with decision procedures,

Hiding formulas: -4, 2,

Installing automatic rewrites from: union_right memory_read_side_effect_super_transformers_memory_read_side_effect_subset_equal

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of memory_read_list_ok.1.

memory_read_list_ok.2:

<div style="display: flex; flex-direction: column; gap: 5px;"> <div style="border-bottom: 1px solid black; padding-bottom: 5px;"> <p>{-1} states'(s')</p> <p>{-2} size' ≥ 0</p> <p>{-3} transformer_invariant?(states',</p> </div> <div style="padding: 5px;"> <p>(memory_read_transformers(pm', address_block(raddr', size')) ∪ memory_read_side_effect_transformers(pm', address_block(raddr', size')))</p> </div> <div style="border-bottom: 1px solid black; padding-bottom: 5px;"> <p>{-4} transformers_ok?(states',</p> <p>(memory_read_transformers(pm', address_block(raddr', size')) ∪ memory_read_side_effect_transformers(pm', address_block(raddr', size')))</p> </div> </div>	<div style="border-bottom: 1px solid black; padding-bottom: 5px;"> <p>{1} transformers_ok?(states', memory_read_transformers(pm', address_block(raddr', size')))</p> <p>{2} OK?((memory_read_list_nse(pm')(raddr', size') ##</p> <p>(λ (bl1: list[Byte]): memory_read_side_effect(pm')(raddr', bl1, FALSE)))</p> <p>(s'))</p> </div>
--	--

Hiding formulas: 2,

Keeping (-4 1) and hiding *,

Using lemma transformers_ok_mono_transformers,

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of memory_read_list_ok.2.

memory_read_list_ok.3:

{-1}	states'(s')
{-2}	size' ≥ 0
{-3}	transformer_invariant?(states', (memory_read_transformers(pm', address_block(raddr', size')) ∪ mem
{-4}	transformers_ok?(states', (memory_read_transformers(pm', address_block(raddr', size')) ∪ memory_rea
{1}	transformer_invariant?(states', memory_read_transformers(pm', ad- dress_block(raddr', size'))
{2}	OK?((memory_read_list_nse(pm')(raddr', size') ## (λ (bl1: list[Byte]): memory_read_side_effect(pm')(raddr', bl1, FALSE))) (s'))

Keeping (-3 1) and hiding *,

Using lemma transformer_invariant_mono_transformers,

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of memory_read_list_ok.3.

Q.E.D.

C.123.14 Memory_Change_2.memory_read_list_next_ok_TCC1

Terse proof for memory_read_list_next_ok_TCC1.

memory_read_list_next_ok_TCC1:

{1}	∨ (pm: Memory_struct[State], states: PRED[State], raddr: Ad- dress, size: nat, s: State): transformer_invariant?(states, (memory_read_transformers(pm, address_block(raddr, size)) ∪ mem ∧ transformers_ok?(states, (memory_read_transformers(pm, address_block(raddr, size)) ∪ memory_re ∧ states(s) ⊃ OK?[State, list[Byte]](memory_read_list[State](pm)(raddr, size)(s)) ∨ Exception?[State, list[Byte]](memory_read_list[State](pm)(raddr, size)(s))
-----	--

Repeatedly simplifying with decision procedures, rewriting, propositional reasoning, quantifier instantiation, skolemization, if-lifting and equality replacement,

Using lemma memory_read_list_ok,

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of memory_read_list_next_ok_TCC1.

Q.E.D.

C.123.15 Memory_Change_2.memory_read_list_next_ok

Terse proof for memory_read_list_next_ok.

memory_read_list_next_ok:

$\{1\} \quad \forall (pm: \text{Memory_struct}[\text{State}], \text{states}: \text{PRED}[\text{State}], \text{raddr}: \text{Address}, \text{size}: \text{nat}, s: \text{State}):$ $\quad \text{transformer_invariant?}(\text{states},$ $\quad \quad \quad (\text{memory_read_transformers}(pm, \text{address_block}(\text{raddr}, \text{size})) \cup \text{memory_read_side_effect}(pm, \text{address_block}(\text{raddr}, \text{size}))))$ $\quad \wedge$ $\quad \text{transformers_ok?}(\text{states},$ $\quad \quad \quad (\text{memory_read_transformers}(pm, \text{address_block}(\text{raddr}, \text{size})) \cup \text{memory_read_side_effect}(pm, \text{address_block}(\text{raddr}, \text{size}))))$ $\quad \wedge \text{states}(s)$ $\quad \supset \text{states}(\text{state}(\text{memory_read_list}(pm)(\text{raddr}, \text{size}))(s))$
--

Repeatedly Skolemizing and flattening,

Using lemma memory_read_list_ok,

Simplifying, rewriting, and recording with decision procedures,

Expanding the definition of memory_read_list,

Expanding the definition of ##,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Using lemma memory_read_list_nse_next_length,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 3 subgoals:

memory_read_list_next_ok.1:

$\{-1\} \quad \text{states}'(s')$ $\{-2\} \quad \text{states}'(\text{state}(\text{memory_read_list_nse}(pm')(\text{raddr}', \text{size}')(s')))$ $\{-3\} \quad \text{length}(\text{data}(\text{memory_read_list_nse}(pm')(\text{raddr}', \text{size}')(s'))) = \text{size}'$ $\{-4\} \quad \text{OK?}(\text{memory_read_list_nse}(pm')(\text{raddr}', \text{size}')(s'))$ $\{-5\} \quad \text{OK?}(\text{memory_read_side_effect}(pm')$ $\quad (\text{raddr}', \text{data}(\text{memory_read_list_nse}(pm')(\text{raddr}', \text{size}')(s')), \text{FALSE})$ $\quad (\text{state}(\text{memory_read_list_nse}(pm')(\text{raddr}', \text{size}')(s'))))$ $\{-6\} \quad \text{size}' \geq 0$ $\{-7\} \quad \text{transformer_invariant?}(\text{states}',$ $\quad \quad \quad (\text{memory_read_transformers}(pm', \text{address_block}(\text{raddr}', \text{size}')) \cup \text{memory_read_side_effect}(pm', \text{address_block}(\text{raddr}', \text{size}'))))$ $\{-8\} \quad \text{transformers_ok?}(\text{states}',$ $\quad \quad \quad (\text{memory_read_transformers}(pm', \text{address_block}(\text{raddr}', \text{size}')) \cup \text{memory_read_side_effect}(pm', \text{address_block}(\text{raddr}', \text{size}'))))$
$\{1\} \quad \text{states}'(\text{state}(\text{memory_read_side_effect}(pm')$ $\quad (\text{raddr}', \text{data}(\text{memory_read_list_nse}(pm')(\text{raddr}', \text{size}')(s')),$ $\quad \quad \quad \text{FALSE})$ $\quad (\text{state}(\text{memory_read_list_nse}(pm')(\text{raddr}', \text{size}')(s'))))$

Hiding formulas: -1, -6, -8,

Installing automatic rewrites from: has_next_state union_right memory_read_side_effect_super_transformers_memory_read_side_effect_subset_equal

Using lemma expr_transformer_invariant_next_ok,

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of memory_read_list_next_ok.1.

memory_read_list_next_ok.2:

{-1}	states'(s')
{-2}	OK?(memory_read_list_nse(pm')(raddr', size')(s'))
{-3}	OK?(memory_read_side_effect(pm') (raddr', data(memory_read_list_nse(pm')(raddr', size')(s')), FALSE) (state(memory_read_list_nse(pm')(raddr', size')(s'))))
{-4}	size' ≥ 0
{-5}	transformer_invariant?(states', (memory_read_transformers(pm', address_block(raddr', size')) ∪ mem
{-6}	transformers_ok?(states', (memory_read_transformers(pm', address_block(raddr', size')) ∪ memory_rea
{1}	transformers_ok?(states', memory_read_transformers(pm', address_block(raddr', size')))
{2}	states'(state(memory_read_side_effect(pm') (raddr', data(memory_read_list_nse(pm')(raddr', size')(s')), FALSE) (state(memory_read_list_nse(pm')(raddr', size')(s'))))

Keeping (-6 1) and hiding *,
 Installing automatic rewrites from: union_subset1
 Using lemma transformers_ok_mono_transformers,
 Simplifying, rewriting, and recording with decision procedures,
 This completes the proof of memory_read_list_next_ok.2.

memory_read_list_next_ok.3:

{-1}	states'(s')
{-2}	OK?(memory_read_list_nse(pm')(raddr', size')(s'))
{-3}	OK?(memory_read_side_effect(pm') (raddr', data(memory_read_list_nse(pm')(raddr', size')(s')), FALSE) (state(memory_read_list_nse(pm')(raddr', size')(s'))))
{-4}	size' ≥ 0
{-5}	transformer_invariant?(states', (memory_read_transformers(pm', address_block(raddr', size')) ∪ mem
{-6}	transformers_ok?(states', (memory_read_transformers(pm', address_block(raddr', size')) ∪ memory_rea
{1}	transformer_invariant?(states', memory_read_transformers(pm', ad-
{2}	states'(state(memory_read_side_effect(pm') (raddr', data(memory_read_list_nse(pm')(raddr', size')(s')), FALSE) (state(memory_read_list_nse(pm')(raddr', size')(s'))))

Keeping (-5 1) and hiding *,
 Installing automatic rewrites from: union_subset1
 Using lemma transformer_invariant_mono_transformers,
 Simplifying, rewriting, and recording with decision procedures,
 This completes the proof of memory_read_list_next_ok.3.
 Q.E.D.

C.123.16

Memory_Change_2.unchanged_memory_read_list_nse_TCC1

Terse proof for unchanged_memory_read_list_nse_TCC1.

unchanged_memory_read_list_nse_TCC1:

$\{1\} \quad \forall (\text{addresses: PRED}[\text{Address}], \text{pm: Memory_struct}[\text{State}], \text{states: PRED}[\text{State}], \text{raddr: Address}, \\ \text{size: nat}, \text{s: State}, \text{q: } [\text{State} \rightarrow \text{SuperResult}[\text{State}]]) : \\ \text{unchanged_memory_invariant?}(\text{pm}, \text{states}, \\ \text{(memory_read_transformers}(\text{pm}, \text{address_block}(\text{raddr}, \text{size})) \cup \text{singleton}(\text{addresses})) \\ \wedge \\ \text{transformers_ok?}(\text{states}, \text{memory_read_transformers}(\text{pm}, \text{address_block}(\text{raddr}, \text{size}))) \wedge \\ \text{(address_block}(\text{raddr}, \text{size}) \subseteq \text{addresses}) \wedge \text{states}(\text{s}) \wedge \text{OK?}(\text{q}(\text{s})) \\ \supset \text{OK?}[\text{State}](\text{q}(\text{s})) \vee \text{abnormal?}[\text{State}](\text{q}(\text{s})))$
--

Repeatedly Skolemizing and flattening,

This completes the proof of unchanged_memory_read_list_nse_TCC1.

Q.E.D.

C.123.17

Memory_Change_2.unchanged_memory_read_list_nse_TCC2

Terse proof for unchanged_memory_read_list_nse_TCC2.

unchanged_memory_read_list_nse_TCC2:

$\{1\} \quad \forall (\text{addresses: PRED}[\text{Address}], \text{pm: Memory_struct}[\text{State}], \text{states: PRED}[\text{State}], \text{raddr: Address}, \\ \text{size: nat}, \text{s: State}, \text{q: } [\text{State} \rightarrow \text{SuperResult}[\text{State}]]) : \\ \text{unchanged_memory_invariant?}(\text{pm}, \text{states}, \\ \text{(memory_read_transformers}(\text{pm}, \text{address_block}(\text{raddr}, \text{size})) \cup \text{singleton}(\text{addresses})) \\ \wedge \\ \text{transformers_ok?}(\text{states}, \text{memory_read_transformers}(\text{pm}, \text{address_block}(\text{raddr}, \text{size}))) \wedge \\ \text{(address_block}(\text{raddr}, \text{size}) \subseteq \text{addresses}) \wedge \text{states}(\text{s}) \wedge \text{OK?}(\text{q}(\text{s})) \\ \supset \\ \text{OK?}[\text{State}, \text{list}[\text{Byte}]] \\ \text{(memory_read_list_nse}[\text{State}](\text{pm})(\text{raddr}, \text{size})(\text{state}[\text{State}](\text{q}(\text{s}))))$
--

Repeatedly Skolemizing and flattening,

Installing automatic rewrites from: has_next_state union_right singleton union_subset1

Using lemma memory_read_list_nse_ok,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 2 subgoals:

unchanged_memory_read_list_nse_TCC2.1:

{-1}	transformers_ok?(states', memory_read_transformers(pm', address_block(raddr', size')))
{-2}	size' ≥ 0
{-3}	unchanged_memory_invariant?(pm', states', (memory_read_transformers(pm', address_block(raddr', size')) addresses'))
{-4}	(address_block(raddr', size') ⊆ addresses')
{-5}	states'(s')
{-6}	OK?(q'(s'))
{1}	states'(state[State](q'(s')))
{2}	OK?[State, list[Byte]] (memory_read_list_nse[State](pm')(raddr', size')(state[State](q'(s'))))

Using lemma unchanged_memory_invariant_next_ok[State],

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of unchanged_memory_read_list_nse_TCC2.1.

unchanged_memory_read_list_nse_TCC2.2:

{-1}	transformers_ok?(states', memory_read_transformers(pm', address_block(raddr', size')))
{-2}	size' ≥ 0
{-3}	unchanged_memory_invariant?(pm', states', (memory_read_transformers(pm', address_block(raddr', size')) addresses'))
{-4}	(address_block(raddr', size') ⊆ addresses')
{-5}	states'(s')
{-6}	OK?(q'(s'))
{1}	transformer_invariant?(states', memory_read_transformers(pm', ad- dress_block(raddr', size')))
{2}	OK?[State, list[Byte]] (memory_read_list_nse[State](pm')(raddr', size')(state[State](q'(s'))))

Forward chaining on unchanged_memory_invariant_invariant,

Keeping (-1 1) and hiding *,

Using lemma transformer_invariant_mono_transformers,

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of unchanged_memory_read_list_nse_TCC2.2.

Q.E.D.

C.123.18

Memory_Change_2.unchanged_memory_read_list_nse_TCC3

Terse proof for unchanged_memory_read_list_nse_TCC3.

unchanged_memory_read_list_nse_TCC3:

$$\{1\} \quad \forall (\text{addresses: PRED}[\text{Address}], \text{pm: Memory_struct}[\text{State}], \text{states: PRED}[\text{State}], \text{raddr: Address}, \\ \text{size: nat}, \text{s: State}, \text{q: } [\text{State} \rightarrow \text{SuperResult}[\text{State}]]) : \\ \text{unchanged_memory_invariant?}(\text{pm}, \text{states}, \\ \text{(memory_read_transformers}(\text{pm}, \text{address_block}(\text{raddr}, \text{size})) \cup \text{singleton}(\text{addresses})) \\ \wedge \\ \text{transformers_ok?}(\text{states}, \text{memory_read_transformers}(\text{pm}, \text{address_block}(\text{raddr}, \text{size}))) \wedge \\ \text{(address_block}(\text{raddr}, \text{size}) \subseteq \text{addresses}) \wedge \text{states}(\text{s}) \wedge \text{OK?}(\text{q}(\text{s})) \\ \supset \text{OK?}[\text{State}, \text{list}[\text{Byte}]](\text{memory_read_list_nse}[\text{State}](\text{pm})(\text{raddr}, \text{size})(\text{s}))$$

Repeatedly Skolemizing and flattening,

Installing automatic rewrites from: union_subset1

Using lemma memory_read_list_nse_ok,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Forward chaining on unchanged_memory_invariant_invariant,

Keeping (-1 1) and hiding *,

Using lemma transformer_invariant_mono_transformers,

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of unchanged_memory_read_list_nse_TCC3.

Q.E.D.

C.123.19 Memory_Change_2.unchanged_memory_read_list_nse

Terse proof for unchanged_memory_read_list_nse.

unchanged_memory_read_list_nse:

$$\{1\} \quad \forall (\text{addresses: PRED}[\text{Address}], \text{pm: Memory_struct}[\text{State}], \text{states: PRED}[\text{State}], \text{raddr: Address}, \\ \text{size: nat}, \text{s: State}, \text{q: } [\text{State} \rightarrow \text{SuperResult}[\text{State}]]) : \\ \text{unchanged_memory_invariant?}(\text{pm}, \text{states}, \\ \text{(memory_read_transformers}(\text{pm}, \text{address_block}(\text{raddr}, \text{size})) \cup \text{singleton}(\text{addresses})) \\ \wedge \\ \text{transformers_ok?}(\text{states}, \text{memory_read_transformers}(\text{pm}, \text{address_block}(\text{raddr}, \text{size}))) \wedge \\ \text{(address_block}(\text{raddr}, \text{size}) \subseteq \text{addresses}) \wedge \text{states}(\text{s}) \wedge \text{OK?}(\text{q}(\text{s})) \\ \supset \\ \text{data}(\text{memory_read_list_nse}(\text{pm})(\text{raddr}, \text{size})(\text{state}(\text{q}(\text{s})))) = \\ \text{data}(\text{memory_read_list_nse}(\text{pm})(\text{raddr}, \text{size})(\text{s}))$$

For the top quantifier in 1, we introduce Skolem constants: (addresses' pm' states' _ _ _ _),

Inducting on size on formula 1,

we get 5 subgoals:

unchanged_memory_read_list_nse.1:

$$\begin{array}{l}
 \{1\} \quad \forall (raddr: \text{Address}, s: \text{State}, q: [\text{State} \rightarrow \text{SuperResult}[\text{State}]]) : \\
 \quad \text{unchanged_memory_invariant?}(pm', \text{states}', \\
 \quad \quad \quad (\text{memory_read_transformers}(pm', \text{address_block}(raddr, 0)) \cup s \\
 \quad \quad \quad \text{addresses}')) \\
 \quad \wedge \\
 \quad \text{transformers_ok?}(\text{states}', \text{memory_read_transformers}(pm', \text{ad-} \\
 \quad \text{dress_block}(raddr, 0))) \wedge \\
 \quad (\text{address_block}(raddr, 0) \subseteq \text{addresses}') \wedge \text{states}'(s) \wedge \text{OK?}(q(s)) \\
 \quad \supset \\
 \quad \text{data}(\text{memory_read_list_nse}(pm')(raddr, 0)(\text{state}(q(s)))) = \\
 \quad \text{data}(\text{memory_read_list_nse}(pm')(raddr, 0)(s))
 \end{array}$$

Expanding the definition of memory_read_list_nse,

Expanding the definition of ok_result,

which is trivially true.

This completes the proof of unchanged_memory_read_list_nse.1.

unchanged_memory_read_list_nse.2:

$$\begin{array}{l}
 \{1\} \quad \forall j : \\
 \quad (\forall (raddr: \text{Address}, s: \text{State}, q: [\text{State} \rightarrow \text{SuperResult}[\text{State}]]) : \\
 \quad \quad \text{unchanged_memory_invariant?}(pm', \text{states}', \\
 \quad \quad \quad (\text{memory_read_transformers}(pm', \text{address_block}(raddr, j) \\
 \quad \quad \quad \text{addresses}')) \\
 \quad \quad \wedge \\
 \quad \quad \text{transformers_ok?}(\text{states}', \text{memory_read_transformers}(pm', \text{ad-} \\
 \quad \quad \text{dress_block}(raddr, j))) \\
 \quad \quad \wedge (\text{address_block}(raddr, j) \subseteq \text{addresses}') \wedge \text{states}'(s) \wedge \text{OK?}(q(s)) \\
 \quad \quad \supset \\
 \quad \quad \text{data}(\text{memory_read_list_nse}(pm')(raddr, j)(\text{state}(q(s)))) = \\
 \quad \quad \text{data}(\text{memory_read_list_nse}(pm')(raddr, j)(s))) \\
 \quad \supset \\
 \quad (\forall (raddr: \text{Address}, s: \text{State}, q: [\text{State} \rightarrow \text{SuperResult}[\text{State}]]) : \\
 \quad \quad \text{unchanged_memory_invariant?}(pm', \text{states}', \\
 \quad \quad \quad (\text{memory_read_transformers}(pm', \text{address_block}(raddr, \\
 \quad \quad \quad \text{addresses}')) \\
 \quad \quad \wedge \\
 \quad \quad \text{transformers_ok?}(\text{states}', \\
 \quad \quad \quad \text{memory_read_transformers}(pm', \text{ad-} \\
 \quad \quad \text{dress_block}(raddr, j + 1))) \\
 \quad \quad \wedge \\
 \quad \quad (\text{address_block}(raddr, j + 1) \subseteq \text{addresses}') \wedge \\
 \quad \quad \text{states}'(s) \wedge \text{OK?}(q(s)) \\
 \quad \quad \supset \\
 \quad \quad \text{data}(\text{memory_read_list_nse}(pm')(raddr, j + 1)(\text{state}(q(s)))) = \\
 \quad \quad \text{data}(\text{memory_read_list_nse}(pm')(raddr, j + 1)(s)))
 \end{array}$$

Repeatedly Skolemizing and flattening,

Simplifying, rewriting, and recording with decision procedures,

Using lemma unchanged_memory_read_list_nse_TCC2,

Using lemma unchanged_memory_read_list_nse_TCC3,

Simplifying, rewriting, and recording with decision procedures,

Expanding the definition of `memory_read_list_nse`,

Expanding the definition of `memory_read_list_nse`,

Expanding the definition of `memory_read_list_nse`,

Expanding the definition of `##`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Hiding formulas: -3, -6, -7,

Expanding the definition of `ok_result`,

Applying extensionality,

we get 2 subgoals:

`unchanged_memory_read_list_nse.2.1`:

{-1}	$\text{OK?}(\text{memory_read}(\text{pm}')(\text{raddr}')(\text{s}'))$
{-2}	$\text{OK?}(\text{memory_read_list_nse}(\text{pm}')(\text{raddr}' + 1, j')$ $\quad (\text{state}(\text{memory_read}(\text{pm}')(\text{raddr}')(\text{s}'))))$
{-3}	$\text{OK?}(\text{memory_read}(\text{pm}')(\text{raddr}')(\text{state}[\text{State}](q'(s'))))$
{-4}	$\text{OK?}(\text{memory_read_list_nse}(\text{pm}')(\text{raddr}' + 1, j')$ $\quad (\text{state}(\text{memory_read}(\text{pm}')$ $\quad \quad (\text{raddr}')(\text{state}[\text{State}](q'(s'))))))$
{-5}	$\forall (\text{raddr}: \text{Address}, \text{s}: \text{State}, \text{q}: [\text{State} \rightarrow \text{SuperResult}[\text{State}]]):$ $\quad \text{unchanged_memory_invariant?}(\text{pm}', \text{states}',$ $\quad \quad (\text{memory_read_transformers}(\text{pm}', \text{address_block}(\text{raddr}, j')) \cup \text{singleton}$ $\quad \quad \quad \text{addresses}'))$ $\quad \wedge$ $\quad \text{transformers_ok?}(\text{states}', \text{memory_read_transformers}(\text{pm}', \text{ad}$ $\quad \text{dress_block}(\text{raddr}, j')) \wedge$ $\quad \quad (\text{address_block}(\text{raddr}, j') \subseteq \text{addresses}') \wedge \text{states}'(\text{s}) \wedge \text{OK?}(\text{q}(\text{s})))$ $\quad \supset$ $\quad \text{data}(\text{memory_read_list_nse}(\text{pm}')(\text{raddr}, j')(\text{state}(\text{q}(\text{s})))) =$ $\quad \quad \text{data}(\text{memory_read_list_nse}(\text{pm}')(\text{raddr}, j')(\text{s})))$
{-6}	$\text{unchanged_memory_invariant?}(\text{pm}', \text{states}',$ $\quad (\text{memory_read_transformers}(\text{pm}', \text{address_block}(\text{raddr}', 1 + j')) \cup \text{singleton}$ $\quad \quad \text{addresses}'))$
{-7}	$\text{transformers_ok?}(\text{states}',$ $\quad \text{memory_read_transformers}(\text{pm}', \text{address_block}(\text{raddr}', 1 + j'))))$
{-8}	$(\text{address_block}(\text{raddr}', 1 + j') \subseteq \text{addresses}')$
{-9}	$\text{states}'(\text{s}')$
{-10}	$\text{OK?}(\text{q}'(\text{s}'))$
{1}	$\text{data}(\text{memory_read}(\text{pm}')(\text{raddr}')(\text{state}(\text{q}'(\text{s}')))) =$ $\quad \text{data}(\text{memory_read}(\text{pm}')(\text{raddr}')(\text{s}'))$

Hiding formulas: -2, -4, -5, -7,

Using lemma `unchanged_memory_invariant_unchanged[State]`,

Installing automatic rewrites from: `union_right singleton`

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Keeping (-6 1) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `unchanged_memory_read_list_nse.2.1`.

unchanged_memory_read_list_nse.2.2:

<div style="display: flex; flex-direction: column; gap: 5px;"> <div style="display: flex; align-items: flex-start;"> {-1} $OK?(memory_read(pm')(raddr')(s'))$ </div> <div style="display: flex; align-items: flex-start;"> {-2} $OK?(memory_read_list_nse(pm')(raddr' + 1, j')$ $(state(memory_read(pm')(raddr')(s'))))$ </div> <div style="display: flex; align-items: flex-start;"> {-3} $OK?(memory_read(pm')(raddr')(state[State](q'(s'))))$ </div> <div style="display: flex; align-items: flex-start;"> {-4} $OK?(memory_read_list_nse(pm')(raddr' + 1, j')$ $(state(memory_read(pm')$ $(raddr')(state[State](q'(s')))))$ </div> <div style="display: flex; align-items: flex-start;"> {-5} $\forall (raddr: Address, s: State, q: [State \rightarrow SuperResult[State]]):$ $unchanged_memory_invariant?(pm', states',$ $(memory_read_transformers(pm', address_block(raddr, j')$ $addresses'))$ \wedge $transformers_ok?(states', memory_read_transformers(pm', ad-$ $dress_block(raddr, j')) \wedge$ $(address_block(raddr, j') \subseteq addresses') \wedge states'(s) \wedge OK?(q(s))$ \supset $data(memory_read_list_nse(pm')(raddr, j')(state(q(s)))) =$ $data(memory_read_list_nse(pm')(raddr, j')(s))$ </div> <div style="display: flex; align-items: flex-start;"> {-6} $unchanged_memory_invariant?(pm', states',$ $(memory_read_transformers(pm', address_block(raddr', 1 + j')$ $addresses'))$ </div> <div style="display: flex; align-items: flex-start;"> {-7} $transformers_ok?(states',$ $memory_read_transformers(pm', address_block(raddr', 1 + j'))$ </div> <div style="display: flex; align-items: flex-start;"> {-8} $(address_block(raddr', 1 + j') \subseteq addresses')$ </div> <div style="display: flex; align-items: flex-start;"> {-9} $states'(s')$ </div> <div style="display: flex; align-items: flex-start;"> {-10} $OK?(q'(s'))$ </div> </div>	<hr style="border: 0.5px solid black; margin-bottom: 5px;"/> <div style="display: flex; flex-direction: column; gap: 5px;"> <div style="display: flex; align-items: flex-start;"> {1} $data(memory_read_list_nse(pm')(raddr' + 1, j')$ $(state(memory_read(pm')(raddr')(state(q'(s'))))))$ </div> <div style="display: flex; align-items: center; margin: 0 10px;"> $=$ </div> <div style="display: flex; align-items: flex-start;"> $data(memory_read_list_nse(pm')(raddr' + 1, j')$ $(state(memory_read(pm')(raddr')(s'))))$ </div> </div>
---	--

Installing automatic rewrites from: state_expr_2_super has_next_state address_block_subset_2 ok_expr_2_super subset_equal union_subset1 union_subset3 unchanged_memory_invariant_union_transformers union_right singleton memory_read_transformers_mono

Case splitting on $unchanged_memory_invariant?(pm!1, states!1, memory_read_transformers(pm!1, address_block(raddr!1 + 1, j!1)), addresses!1)$,

we get 2 subgoals:

unchanged_memory_read_list_nse.2.2.1:

{-1}	unchanged_memory_invariant?(pm', states', memory_read_transformers(pm', address_block(raddr' + 1, j')), addresses')
{-2}	OK?(memory_read(pm')(raddr')(s'))
{-3}	OK?(memory_read_list_nse(pm')(raddr' + 1, j') (state(memory_read(pm')(raddr')(s'))))
{-4}	OK?(memory_read(pm')(raddr')(state[State](q'(s'))))
{-5}	OK?(memory_read_list_nse(pm')(raddr' + 1, j') (state(memory_read(pm') (raddr')(state[State](q'(s'))))))
{-6}	\forall (raddr: Address, s: State, q: [State \rightarrow SuperResult[State]]): unchanged_memory_invariant?(pm', states', (memory_read_transformers(pm', address_block(raddr, j')) \cup singleton addresses') \wedge transformers_ok?(states', memory_read_transformers(pm', ad- dress_block(raddr, j'))) \wedge (address_block(raddr, j') \subseteq addresses') \wedge states'(s) \wedge OK?(q(s)) \supset data(memory_read_list_nse(pm')(raddr, j')(state(q(s)))) = data(memory_read_list_nse(pm')(raddr, j')(s))
{-7}	unchanged_memory_invariant?(pm', states', (memory_read_transformers(pm', address_block(raddr', 1 + j')) \cup singleton addresses')
{-8}	transformers_ok?(states', memory_read_transformers(pm', address_block(raddr', 1 + j')))
{-9}	(address_block(raddr', 1 + j') \subseteq addresses')
{-10}	states'(s')
{-11}	OK?(q'(s'))
{1}	data(memory_read_list_nse(pm')(raddr' + 1, j') (state(memory_read(pm')(raddr')(state(q'(s')))))) = data(memory_read_list_nse(pm')(raddr' + 1, j') (state(memory_read(pm')(raddr')(s'))))

Case splitting on unchanged_memory_invariant?(pm!1, states!1, singleton(expr_2_super (memory_read(pm!1)(raddr!1))), addresses!1),

we get 2 subgoals:

unchanged_memory_read_list_nse.2.2.1.1:

{-1}	$\text{unchanged_memory_invariant?}(\text{pm}', \text{states}',$ $\text{singleton}(\text{expr_2_super}(\text{memory_read}(\text{pm}')(\text{raddr}')),$ $\text{addresses}')$
{-2}	$\text{unchanged_memory_invariant?}(\text{pm}', \text{states}',$ $\text{memory_read_transformers}(\text{pm}',$ $\text{address_block}(\text{raddr}' + 1, j'),$ $\text{addresses}')$
{-3}	$\text{OK?}(\text{memory_read}(\text{pm}')(\text{raddr}')(s'))$
{-4}	$\text{OK?}(\text{memory_read_list_nse}(\text{pm}')(\text{raddr}' + 1, j')$ $(\text{state}(\text{memory_read}(\text{pm}')(\text{raddr}')(s'))))$
{-5}	$\text{OK?}(\text{memory_read}(\text{pm}')(\text{raddr}')(s[\text{State}](q'(s'))))$
{-6}	$\text{OK?}(\text{memory_read_list_nse}(\text{pm}')(\text{raddr}' + 1, j')$ $(\text{state}(\text{memory_read}(\text{pm}'$ $(\text{raddr}')(\text{state}[\text{State}](q'(s'))))))$
{-7}	$\forall (\text{raddr}: \text{Address}, s: \text{State}, q: [\text{State} \rightarrow \text{SuperResult}[\text{State}]]):$ $\text{unchanged_memory_invariant?}(\text{pm}', \text{states}',$ $\text{memory_read_transformers}(\text{pm}', \text{address_block}(\text{raddr}, j'),$ $\text{addresses}')$ \wedge $\text{transformers_ok?}(\text{states}', \text{memory_read_transformers}(\text{pm}', \text{ad-}$ $\text{dress_block}(\text{raddr}, j')) \wedge$ $(\text{address_block}(\text{raddr}, j') \subseteq \text{addresses}') \wedge \text{states}'(s) \wedge \text{OK?}(q(s))$ \supseteq $\text{data}(\text{memory_read_list_nse}(\text{pm}')(\text{raddr}, j')(\text{state}(q(s)))) =$ $\text{data}(\text{memory_read_list_nse}(\text{pm}')(\text{raddr}, j')(s))$
{-8}	$\text{unchanged_memory_invariant?}(\text{pm}', \text{states}',$ $\text{memory_read_transformers}(\text{pm}', \text{address_block}(\text{raddr}', 1 + j')$ $\text{addresses}')$
{-9}	$\text{transformers_ok?}(\text{states}',$ $\text{memory_read_transformers}(\text{pm}', \text{address_block}(\text{raddr}', 1 + j')))$
{-10}	$(\text{address_block}(\text{raddr}', 1 + j') \subseteq \text{addresses}')$
{-11}	$\text{states}'(s')$
{-12}	$\text{OK?}(q'(s'))$
{1}	$\text{data}(\text{memory_read_list_nse}(\text{pm}')(\text{raddr}' + 1, j')$ $(\text{state}(\text{memory_read}(\text{pm}')(\text{raddr}')(\text{state}(q'(s'))))))$ $=$ $\text{data}(\text{memory_read_list_nse}(\text{pm}')(\text{raddr}' + 1, j')$ $(\text{state}(\text{memory_read}(\text{pm}')(\text{raddr}')(\text{state}(q'(s'))))))$

Case splitting on $\text{transformers_ok?}(\text{states}'1, \text{memory_read_transformers}(\text{pm}'1, \text{address_block}(\text{raddr}'1 + 1, j'1)))$,

we get 2 subgoals:

unchanged_memory_read_list_nse.2.2.1.1.1:

{-1}	transformers_ok?(states', memory_read_transformers(pm', address_block(raddr' + 1, j')))
{-2}	unchanged_memory_invariant?(pm', states', singleton(expr_2_super(memory_read(pm')(raddr'))), addresses')
{-3}	unchanged_memory_invariant?(pm', states', memory_read_transformers(pm', address_block(raddr' + 1, j')), addresses')
{-4}	OK?(memory_read(pm')(raddr')(s'))
{-5}	OK?(memory_read_list_nse(pm')(raddr' + 1, j') (state(memory_read(pm')(raddr')(s'))))
{-6}	OK?(memory_read(pm')(raddr')(state[State](q'(s'))))
{-7}	OK?(memory_read_list_nse(pm')(raddr' + 1, j') (state(memory_read(pm') (raddr')(state[State](q'(s'))))))
{-8}	\forall (raddr: Address, s: State, q: [State \rightarrow SuperResult[State]]): unchanged_memory_invariant?(pm', states', (memory_read_transformers(pm', address_block(raddr, j')) \cup singleton addresses') \wedge transformers_ok?(states', memory_read_transformers(pm', ad- dress_block(raddr, j'))) \wedge (address_block(raddr, j') \subseteq addresses') \wedge states'(s) \wedge OK?(q(s)) \supset data(memory_read_list_nse(pm')(raddr, j')(state(q(s)))) = data(memory_read_list_nse(pm')(raddr, j')(s))
{-9}	unchanged_memory_invariant?(pm', states', (memory_read_transformers(pm', address_block(raddr', 1 + j')) \cup singleton addresses')
{-10}	transformers_ok?(states', memory_read_transformers(pm', address_block(raddr', 1 + j')))
{-11}	(address_block(raddr', 1 + j') \subseteq addresses')
{-12}	states'(s')
{-13}	OK?(q'(s'))
{1}	data(memory_read_list_nse(pm')(raddr' + 1, j') (state(memory_read(pm')(raddr')(state(q'(s')))))) = data(memory_read_list_nse(pm')(raddr' + 1, j') (state(memory_read(pm')(raddr')(s'))))

Copying formula number: -8

Instantiating the top quantifier in -1 with the terms: raddr' + 1, s', expr_2_super(memory_read(pm')(raddr')),

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Replacing using formula -3,

Hiding formulas: -3,

Copying formula number: -9

Instantiating the top quantifier in -1 with the terms: raddr' + 1, state(q'(s')), expr_2_super(memory_read(pm')(raddr')),

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 2 subgoals:

unchanged_memory_read_list_nse.2.2.1.1.1.1:

{-1}	transformers_ok?(states', memory_read_transformers(pm', address_block(raddr' + 1, j')))
{-2}	data(memory_read_list_nse(pm')(raddr' + 1, j') (state(memory_read(pm')(raddr')(state(q'(s')))))) = data(memory_read_list_nse(pm')(raddr' + 1, j')(state(q'(s'))))
{-3}	states'(s')
{-4}	unchanged_memory_invariant?(pm', states', singleton(expr_2_super(memory_read(pm')(raddr'))), addresses')
{-5}	unchanged_memory_invariant?(pm', states', memory_read_transformers(pm', ad- dress_block(raddr' + 1, j')), addresses')
{-6}	OK?(memory_read(pm')(raddr')(s'))
{-7}	OK?(memory_read_list_nse(pm')(raddr' + 1, j') (state(memory_read(pm')(raddr')(s'))))
{-8}	OK?(memory_read(pm')(raddr')(state[State](q'(s'))))
{-9}	OK?(memory_read_list_nse(pm')(raddr' + 1, j') (state(memory_read(pm') (raddr')(state[State](q'(s'))))))
{-10}	\forall (raddr: Address, s: State, q: [State \rightarrow SuperResult[State]]): unchanged_memory_invariant?(pm', states', (memory_read_transformers(pm', address_block(raddr, j') addresses') \wedge transformers_ok?(states', memory_read_transformers(pm', ad- dress_block(raddr, j')) \wedge (address_block(raddr, j') \subseteq addresses') \wedge states'(s) \wedge OK?(q(s)) \supset data(memory_read_list_nse(pm')(raddr, j')(state(q(s)))) = data(memory_read_list_nse(pm')(raddr, j')(s))
{-11}	unchanged_memory_invariant?(pm', states', (memory_read_transformers(pm', address_block(raddr', 1 + j') addresses')
{-12}	transformers_ok?(states', memory_read_transformers(pm', address_block(raddr', 1 + j')))
{-13}	(address_block(raddr', 1 + j') \subseteq addresses')
{-14}	OK?(q'(s'))
{1}	data(memory_read_list_nse(pm')(raddr' + 1, j') (state(memory_read(pm')(raddr')(state(q'(s')))))) = data(memory_read_list_nse(pm')(raddr' + 1, j')(s'))

Replacing using formula -2,

Hiding formulas: -2,

Case splitting on unchanged_memory_invariant?(pm!1, states!1, singleton(q!1), addresses!1),

we get 2 subgoals:

unchanged_memory_read_list_nse.2.2.1.1.1.1.1:

{-1}	unchanged_memory_invariant?(pm', states', singleton(q'), addresses')
{-2}	transformers_ok?(states', memory_read_transformers(pm', address_block(raddr' + 1, j')))
{-3}	states'(s')
{-4}	unchanged_memory_invariant?(pm', states', singleton(expr_2_super(memory_read(pm')(raddr'))), addresses')
{-5}	unchanged_memory_invariant?(pm', states', memory_read_transformers(pm', ad- dress_block(raddr' + 1, j')), addresses')
{-6}	OK?(memory_read(pm')(raddr')(s'))
{-7}	OK?(memory_read_list_nse(pm')(raddr' + 1, j') (state(memory_read(pm')(raddr')(s'))))
{-8}	OK?(memory_read(pm')(raddr')(state[State](q'(s'))))
{-9}	OK?(memory_read_list_nse(pm')(raddr' + 1, j') (state(memory_read(pm') (raddr')(state[State](q'(s'))))))
{-10}	\forall (raddr: Address, s: State, q: [State \rightarrow SuperResult[State]]): unchanged_memory_invariant?(pm', states', (memory_read_transformers(pm', address_block(raddr, j')) \cup singleton addresses') \wedge transformers_ok?(states', memory_read_transformers(pm', ad- dress_block(raddr, j')) \wedge (address_block(raddr, j') \subseteq addresses') \wedge states'(s) \wedge OK?(q(s)) \supset data(memory_read_list_nse(pm')(raddr, j')(state(q(s)))) = data(memory_read_list_nse(pm')(raddr, j')(s))
{-11}	unchanged_memory_invariant?(pm', states', (memory_read_transformers(pm', address_block(raddr', 1 + j')) \cup singleton addresses')
{-12}	transformers_ok?(states', memory_read_transformers(pm', address_block(raddr', 1 + j')))
{-13}	(address_block(raddr', 1 + j') \subseteq addresses')
{-14}	OK?(q'(s'))
{1}	data(memory_read_list_nse(pm')(raddr' + 1, j')(state(q'(s')))) = data(memory_read_list_nse(pm')(raddr' + 1, j')(s'))

Rewriting using -10, matching in *,

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of unchanged_memory_read_list_nse.2.2.1.1.1.1.1.

unchanged_memory_read_list_nse.2.2.1.1.1.1.2:

{-1}	transformers_ok?(states', memory_read_transformers(pm', address_block(raddr' + 1, j')))
{-2}	states'(s')
{-3}	unchanged_memory_invariant?(pm', states', singleton(expr_2_super(memory_read(pm')(raddr'))), addresses')
{-4}	unchanged_memory_invariant?(pm', states', memory_read_transformers(pm', ad- dress_block(raddr' + 1, j')), addresses')
{-5}	OK?(memory_read(pm')(raddr')(s'))
{-6}	OK?(memory_read_list_nse(pm')(raddr' + 1, j') (state(memory_read(pm')(raddr')(s'))))
{-7}	OK?(memory_read(pm')(raddr')(state[State](q'(s'))))
{-8}	OK?(memory_read_list_nse(pm')(raddr' + 1, j') (state(memory_read(pm') (raddr')(state[State](q'(s'))))))
{-9}	\forall (raddr: Address, s: State, q: [State \rightarrow SuperResult[State]]): unchanged_memory_invariant?(pm', states', (memory_read_transformers(pm', address_block(raddr, j')) addresses') \wedge transformers_ok?(states', memory_read_transformers(pm', ad- dress_block(raddr, j')) \wedge (address_block(raddr, j') \subseteq addresses') \wedge states'(s) \wedge OK?(q(s)) \supset data(memory_read_list_nse(pm')(raddr, j')(state(q(s)))) = data(memory_read_list_nse(pm')(raddr, j')(s))
{-10}	unchanged_memory_invariant?(pm', states', (memory_read_transformers(pm', address_block(raddr', 1 + j') addresses'))
{-11}	transformers_ok?(states', memory_read_transformers(pm', address_block(raddr', 1 + j')))
{-12}	(address_block(raddr', 1 + j') \subseteq addresses')
{-13}	OK?(q'(s'))
{1}	unchanged_memory_invariant?(pm', states', singleton(q'), addresses')
{2}	data(memory_read_list_nse(pm')(raddr' + 1, j')(state(q'(s')))) = data(memory_read_list_nse(pm')(raddr' + 1, j')(s'))

Keeping (-10 1) and hiding *,

Using lemma unchanged_memory_invariant_mono,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of unchanged_memory_read_list_nse.2.2.1.1.1.1.2.

unchanged_memory_read_list_nse.2.2.1.1.1.2:

{-1}	transformers_ok?(states', memory_read_transformers(pm', address_block(raddr' + 1, j')))
{-2}	states'(s')
{-3}	unchanged_memory_invariant?(pm', states', singleton(expr_2_super(memory_read(pm')(raddr'))), addresses')
{-4}	unchanged_memory_invariant?(pm', states', memory_read_transformers(pm', address_block(raddr' + 1, j')), addresses')
{-5}	OK?(memory_read(pm')(raddr')(s'))
{-6}	OK?(memory_read_list_nse(pm')(raddr' + 1, j') (state(memory_read(pm')(raddr')(s'))))
{-7}	OK?(memory_read(pm')(raddr')(state[State](q'(s'))))
{-8}	OK?(memory_read_list_nse(pm')(raddr' + 1, j') (state(memory_read(pm') (raddr')(state[State](q'(s'))))))
{-9}	\forall (raddr: Address, s: State, q: [State \rightarrow SuperResult[State]]): unchanged_memory_invariant?(pm', states', (memory_read_transformers(pm', address_block(raddr, j')) \cup singleton addresses') \wedge transformers_ok?(states', memory_read_transformers(pm', ad- dress_block(raddr, j'))) \wedge (address_block(raddr, j') \subseteq addresses') \wedge states'(s) \wedge OK?(q(s)) \supset data(memory_read_list_nse(pm')(raddr, j')(state(q(s)))) = data(memory_read_list_nse(pm')(raddr, j')(s))
{-10}	unchanged_memory_invariant?(pm', states', (memory_read_transformers(pm', address_block(raddr', 1 + j')) \cup singleton addresses')
{-11}	transformers_ok?(states', memory_read_transformers(pm', address_block(raddr', 1 + j')))
{-12}	(address_block(raddr', 1 + j') \subseteq addresses')
{-13}	OK?(q'(s'))
{1}	states'(state(q'(s')))
{2}	data(memory_read_list_nse(pm')(raddr' + 1, j') (state(memory_read(pm')(raddr')(state(q'(s')))))) = data(memory_read_list_nse(pm')(raddr' + 1, j')(s'))

Keeping (-10 1) and hiding *,

Using lemma unchanged_memory_invariant_next_ok[State],

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of unchanged_memory_read_list_nse.2.2.1.1.1.2.

unchanged_memory_read_list_nse.2.2.1.1.2:

{-1}	unchanged_memory_invariant?(pm', states', singleton(expr_2_super(memory_read(pm')(raddr'))), addresses')
{-2}	unchanged_memory_invariant?(pm', states', memory_read_transformers(pm', address_block(raddr' + 1, j')), addresses')
{-3}	OK?(memory_read(pm')(raddr')(s'))
{-4}	OK?(memory_read_list_nse(pm')(raddr' + 1, j')(state(memory_read(pm')(raddr')(s'))))
{-5}	OK?(memory_read(pm')(raddr')(state[State](q'(s'))))
{-6}	OK?(memory_read_list_nse(pm')(raddr' + 1, j')(state(memory_read(pm')(raddr')(state[State](q'(s'))))))
{-7}	\forall (raddr: Address, s: State, q: [State \rightarrow SuperResult[State]]): unchanged_memory_invariant?(pm', states', (memory_read_transformers(pm', address_block(raddr, j')), addresses') \wedge transformers_ok?(states', memory_read_transformers(pm', address_block(raddr, j'))) \wedge (address_block(raddr, j') \subseteq addresses') \wedge states'(s) \wedge OK?(q(s)) \supset data(memory_read_list_nse(pm')(raddr, j')(state(q(s)))) = data(memory_read_list_nse(pm')(raddr, j')(s)))
{-8}	unchanged_memory_invariant?(pm', states', (memory_read_transformers(pm', address_block(raddr', 1 + j')), addresses')
{-9}	transformers_ok?(states', memory_read_transformers(pm', address_block(raddr', 1 + j')))
{-10}	(address_block(raddr', 1 + j') \subseteq addresses')
{-11}	states'(s')
{-12}	OK?(q'(s'))
{1}	transformers_ok?(states', memory_read_transformers(pm', address_block(raddr' + 1, j')))
{2}	data(memory_read_list_nse(pm')(raddr' + 1, j')(state(memory_read(pm')(raddr')(state(q'(s')))))) = data(memory_read_list_nse(pm')(raddr' + 1, j')(state(memory_read(pm')(raddr')(s'))))

Keeping (-9 1) and hiding *,

Using lemma transformers_ok_mono_transformers,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of unchanged_memory_read_list_nse.2.2.1.1.2.

unchanged_memory_read_list_nse.2.2.1.2:

{-1}	$\text{unchanged_memory_invariant?}(\text{pm}', \text{states}',$ $\text{memory_read_transformers}(\text{pm}',$ $\text{address_block}(\text{raddr}' + 1, j'),$ $\text{addresses}')$
{-2}	$\text{OK?}(\text{memory_read}(\text{pm}')(\text{raddr}')(\text{s}'))$
{-3}	$\text{OK?}(\text{memory_read_list_nse}(\text{pm}')(\text{raddr}' + 1, j')$ $(\text{state}(\text{memory_read}(\text{pm}')(\text{raddr}')(\text{s}'))))$
{-4}	$\text{OK?}(\text{memory_read}(\text{pm}')(\text{raddr}')(\text{state}[\text{State}](q'(s'))))$
{-5}	$\text{OK?}(\text{memory_read_list_nse}(\text{pm}')(\text{raddr}' + 1, j')$ $(\text{state}(\text{memory_read}(\text{pm}'$ $(\text{raddr}')(\text{state}[\text{State}](q'(s'))))))$
{-6}	$\forall (\text{raddr}: \text{Address}, \text{s}: \text{State}, \text{q}: [\text{State} \rightarrow \text{SuperResult}[\text{State}]]):$ $\text{unchanged_memory_invariant?}(\text{pm}', \text{states}',$ $(\text{memory_read_transformers}(\text{pm}', \text{address_block}(\text{raddr}, j')) \cup \text{singleton}$ $\text{addresses}')$ \wedge $\text{transformers_ok?}(\text{states}', \text{memory_read_transformers}(\text{pm}', \text{ad}$ $\text{dress_block}(\text{raddr}, j')) \wedge$ $(\text{address_block}(\text{raddr}, j') \subseteq \text{addresses}') \wedge \text{states}'(\text{s}) \wedge \text{OK?}(q(\text{s}))$ \supseteq $\text{data}(\text{memory_read_list_nse}(\text{pm}')(\text{raddr}, j')(\text{state}(q(\text{s})))) =$ $\text{data}(\text{memory_read_list_nse}(\text{pm}')(\text{raddr}, j')(s))$
{-7}	$\text{unchanged_memory_invariant?}(\text{pm}', \text{states}',$ $(\text{memory_read_transformers}(\text{pm}', \text{address_block}(\text{raddr}', 1 + j')) \cup \text{singleton}$ $\text{addresses}')$
{-8}	$\text{transformers_ok?}(\text{states}',$ $\text{memory_read_transformers}(\text{pm}', \text{address_block}(\text{raddr}', 1 + j')))$
{-9}	$(\text{address_block}(\text{raddr}', 1 + j') \subseteq \text{addresses}')$
{-10}	$\text{states}'(s')$
{-11}	$\text{OK?}(q'(s'))$
{1}	$\text{unchanged_memory_invariant?}(\text{pm}', \text{states}',$ $\text{singleton}(\text{expr_2_super}(\text{memory_read}(\text{pm}')(\text{raddr}'))),$ $\text{addresses}')$
{2}	$\text{data}(\text{memory_read_list_nse}(\text{pm}')(\text{raddr}' + 1, j')$ $(\text{state}(\text{memory_read}(\text{pm}')(\text{raddr}')(\text{state}(q'(s'))))))$ $=$ $\text{data}(\text{memory_read_list_nse}(\text{pm}')(\text{raddr}' + 1, j')$ $(\text{state}(\text{memory_read}(\text{pm}')(\text{raddr}')(\text{s}'))))$

Keeping (-7 1) and hiding *,

Using lemma unchanged_memory_invariant_mono,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Hiding formulas: -1,

Using lemma unchanged_memory_invariant_mono,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Rewriting using singleton_subset, matching in *,

Expanding the definition of member,

Rewriting using memory_read_transformers_memory_read, matching in *,

Keeping 1 and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of unchanged_memory_read_list_nse.2.2.1.2.

`unchanged_memory_read_list_nse.2.2.2`:

{-1}	OK?(memory_read(pm')(raddr')(s'))
{-2}	OK?(memory_read_list_nse(pm')(raddr' + 1, j') (state(memory_read(pm')(raddr')(s'))))
{-3}	OK?(memory_read(pm')(raddr')(state[State](q'(s'))))
{-4}	OK?(memory_read_list_nse(pm')(raddr' + 1, j') (state(memory_read(pm') (raddr')(state[State](q'(s')))))
{-5}	\forall (raddr: Address, s: State, q: [State \rightarrow SuperResult[State]]): unchanged_memory_invariant?(pm', states', (memory_read_transformers(pm', address_block(raddr, j') addresses')) \wedge transformers_ok?(states', memory_read_transformers(pm', ad- dress_block(raddr, j')))) \wedge (address_block(raddr, j') \subseteq addresses') \wedge states'(s) \wedge OK?(q(s)) \supset data(memory_read_list_nse(pm')(raddr, j')(state(q(s)))) = data(memory_read_list_nse(pm')(raddr, j')(s))
{-6}	unchanged_memory_invariant?(pm', states', (memory_read_transformers(pm', address_block(raddr', 1 + j') addresses'))
{-7}	transformers_ok?(states', memory_read_transformers(pm', address_block(raddr', 1 + j'))
{-8}	(address_block(raddr', 1 + j') \subseteq addresses')
{-9}	states'(s')
{-10}	OK?(q'(s'))
{1}	unchanged_memory_invariant?(pm', states', memory_read_transformers(pm', ad- dress_block(raddr' + 1, j')), addresses')
{2}	data(memory_read_list_nse(pm')(raddr' + 1, j') (state(memory_read(pm')(raddr')(state(q'(s'))))) = data(memory_read_list_nse(pm')(raddr' + 1, j') (state(memory_read(pm')(raddr')(s'))))

Keeping {-6 1} and hiding *,

Using lemma unchanged_memory_invariant_mono,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Hiding formulas: -1,

Using lemma unchanged_memory_invariant_mono,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `unchanged_memory_read_list_nse.2.2.2`.

unchanged_memory_read_list_nse.3:

{1}	$\forall (\text{size: nat, raddr: Address, } s: \text{State, } q: [\text{State} \rightarrow \text{SuperResult}[\text{State}]]):$ $\text{unchanged_memory_invariant?}(\text{pm}', \text{states}',$ $\quad (\text{memory_read_transformers}(\text{pm}', \text{address_block}(\text{raddr}, \text{size})) \cup \text{singleton}$ $\quad \text{addresses}')$ \wedge $\text{transformers_ok?}(\text{states}', \text{memory_read_transformers}(\text{pm}', \text{ad-}$ $\text{dress_block}(\text{raddr}, \text{size}))) \wedge$ $(\text{address_block}(\text{raddr}, \text{size}) \subseteq \text{addresses}') \wedge \text{states}'(s) \wedge \text{OK?}(q(s))$ $\supset \text{OK?}[\text{State}, \text{list}[\text{Byte}]](\text{memory_read_list_nse}[\text{State}](\text{pm}')(\text{raddr}, \text{size})(s))$
{2}	$\forall (\text{raddr: Address, } s: \text{State, } q: [\text{State} \rightarrow \text{SuperResult}[\text{State}]]):$ $\text{unchanged_memory_invariant?}(\text{pm}', \text{states}',$ $\quad (\text{memory_read_transformers}(\text{pm}', \text{address_block}(\text{raddr}, \text{size}')) \cup \text{singleton}$ $\quad \text{addresses}')$ \wedge $\text{transformers_ok?}(\text{states}', \text{memory_read_transformers}(\text{pm}', \text{ad-}$ $\text{dress_block}(\text{raddr}, \text{size}'))$ $\wedge (\text{address_block}(\text{raddr}, \text{size}') \subseteq \text{addresses}') \wedge \text{states}'(s) \wedge \text{OK?}(q(s))$ \supset $\text{data}(\text{memory_read_list_nse}(\text{pm}')(\text{raddr}, \text{size}')(\text{state}(q(s)))) =$ $\text{data}(\text{memory_read_list_nse}(\text{pm}')(\text{raddr}, \text{size}')(\text{state}(q(s))))$

Hiding formulas: 2,

Repeatedly Skolemizing and flattening,

Using lemma unchanged_memory_read_list_nse_TCC3,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of unchanged_memory_read_list_nse.3.

unchanged_memory_read_list_nse.4:

{1}	$\forall (\text{size: nat, raddr: Address, } s: \text{State, } q: [\text{State} \rightarrow \text{SuperResult}[\text{State}]]):$ $\text{unchanged_memory_invariant?}(\text{pm}', \text{states}',$ $\quad (\text{memory_read_transformers}(\text{pm}', \text{address_block}(\text{raddr}, \text{size})) \cup \text{singleton}$ $\quad \text{addresses}')$ \wedge $\text{transformers_ok?}(\text{states}', \text{memory_read_transformers}(\text{pm}', \text{ad-}$ $\text{dress_block}(\text{raddr}, \text{size}))) \wedge$ $(\text{address_block}(\text{raddr}, \text{size}) \subseteq \text{addresses}') \wedge \text{states}'(s) \wedge \text{OK?}(q(s))$ \supset $\text{OK?}[\text{State}, \text{list}[\text{Byte}]]$ $\quad (\text{memory_read_list_nse}[\text{State}](\text{pm}')(\text{raddr}, \text{size})(\text{state}[\text{State}](q(s))))$
{2}	$\forall (\text{raddr: Address, } s: \text{State, } q: [\text{State} \rightarrow \text{SuperResult}[\text{State}]]):$ $\text{unchanged_memory_invariant?}(\text{pm}', \text{states}',$ $\quad (\text{memory_read_transformers}(\text{pm}', \text{address_block}(\text{raddr}, \text{size}')) \cup \text{singleton}$ $\quad \text{addresses}')$ \wedge $\text{transformers_ok?}(\text{states}', \text{memory_read_transformers}(\text{pm}', \text{ad-}$ $\text{dress_block}(\text{raddr}, \text{size}'))$ $\wedge (\text{address_block}(\text{raddr}, \text{size}') \subseteq \text{addresses}') \wedge \text{states}'(s) \wedge \text{OK?}(q(s))$ \supset $\text{data}(\text{memory_read_list_nse}(\text{pm}')(\text{raddr}, \text{size}')(\text{state}(q(s)))) =$ $\text{data}(\text{memory_read_list_nse}(\text{pm}')(\text{raddr}, \text{size}')(\text{state}(q(s))))$

Hiding formulas: 2,
 Repeatedly Skolemizing and flattening,
 Using lemma unchanged_memory_read_list_nse_TCC2,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 This completes the proof of unchanged_memory_read_list_nse.4.
 unchanged_memory_read_list_nse.5:

{1}	$\forall (\text{size: nat, raddr: Address, } s: \text{State, } q: [\text{State} \rightarrow \text{SuperResult}[\text{State}]]):$ $\text{unchanged_memory_invariant?}(\text{pm}', \text{states}',$ $\quad (\text{memory_read_transformers}(\text{pm}', \text{address_block}(\text{raddr}, \text{size}))$ $\quad \text{addresses}')$ \wedge $\text{transformers_ok?}(\text{states}', \text{memory_read_transformers}(\text{pm}', \text{ad-}$ $\text{dress_block}(\text{raddr}, \text{size}))) \wedge$ $(\text{address_block}(\text{raddr}, \text{size}) \subseteq \text{addresses}') \wedge \text{states}'(s) \wedge \text{OK?}(q(s))$ $\supset \text{OK?}[\text{State}](q(s)) \vee \text{abnormal?}[\text{State}](q(s))$
{2}	$\forall (\text{raddr: Address, } s: \text{State, } q: [\text{State} \rightarrow \text{SuperResult}[\text{State}]]):$ $\text{unchanged_memory_invariant?}(\text{pm}', \text{states}',$ $\quad (\text{memory_read_transformers}(\text{pm}', \text{address_block}(\text{raddr}, \text{size}'))$ $\quad \text{addresses}')$ \wedge $\text{transformers_ok?}(\text{states}', \text{memory_read_transformers}(\text{pm}', \text{ad-}$ $\text{dress_block}(\text{raddr}, \text{size}'))$ $\wedge (\text{address_block}(\text{raddr}, \text{size}') \subseteq \text{addresses}') \wedge \text{states}'(s) \wedge \text{OK?}(q(s))$ \supset $\text{data}(\text{memory_read_list_nse}(\text{pm}')(\text{raddr}, \text{size}')(\text{state}(q(s)))) =$ $\text{data}(\text{memory_read_list_nse}(\text{pm}')(\text{raddr}, \text{size}')(s))$

Hiding formulas: 2,
 Repeatedly Skolemizing and flattening,
 This completes the proof of unchanged_memory_read_list_nse.5.
 Q.E.D.

C.123.20 Memory_Change_2.unchanged_memory_read_list_TCC1

Terse proof for unchanged_memory_read_list_TCC1.
 unchanged_memory_read_list_TCC1:

{1}	$\forall (\text{addresses: PRED}[\text{Address}], \text{pm: Memory_struct}[\text{State}], \text{states: PRED}[\text{State}], \text{raddr: Ad-}$ dress, $\quad \text{size: nat, } s: \text{State, } q: [\text{State} \rightarrow \text{SuperResult}[\text{State}]]):$ $\text{unchanged_memory_invariant?}(\text{pm}, \text{states,}$ $\quad ((\text{memory_read_transformers}(\text{pm}, \text{address_block}(\text{raddr}, \text{size}))$ $\quad \text{addresses}))$ \wedge $\text{transformers_ok?}(\text{states,}$ $\quad (\text{memory_read_transformers}(\text{pm}, \text{address_block}(\text{raddr}, \text{size})) \cup \text{memory_re-}$ $\quad \text{ad_transformers}(\text{pm})))$ \wedge $\text{side_effect_content_unchanged}(\text{address_block}(\text{raddr}, \text{size}), \text{states,}$ $\quad \text{memory_read_side_effect}(\text{pm}))$ $\wedge (\text{address_block}(\text{raddr}, \text{size}) \subseteq \text{addresses}) \wedge \text{states}(s) \wedge \text{OK?}(q(s))$ $\supset \text{OK?}[\text{State}](q(s)) \vee \text{abnormal?}[\text{State}](q(s))$
-----	---

Repeatedly Skolemizing and flattening,

This completes the proof of `unchanged_memory_read_list_TCC1`.

Q.E.D.

C.123.21 Memory_Change_2.unchanged_memory_read_list_TCC2

Terse proof for `unchanged_memory_read_list_TCC2`.

`unchanged_memory_read_list_TCC2`:

$ \begin{aligned} & \{1\} \quad \forall (\text{addresses: PRED}[\text{Address}], \text{pm: Memory_struct}[\text{State}], \text{states: PRED}[\text{State}], \text{raddr: Ad-} \\ & \quad \text{dress,} \\ & \quad \text{size: nat, } s: \text{State}, q: [\text{State} \rightarrow \text{SuperResult}[\text{State}]]) : \\ & \quad \text{unchanged_memory_invariant?}(\text{pm}, \text{states}, \\ & \quad \quad \quad ((\text{memory_read_transformers}(\text{pm}, \text{address_block}(\text{raddr}, \text{size})) \cup \text{memory_} \\ & \quad \quad \quad \text{addresses})) \\ & \quad \wedge \\ & \quad \text{transformers_ok?}(\text{states}, \\ & \quad \quad \quad (\text{memory_read_transformers}(\text{pm}, \text{address_block}(\text{raddr}, \text{size})) \cup \text{memory_read_side_effe} \\ & \quad \wedge \\ & \quad \text{side_effect_content_unchanged}(\text{address_block}(\text{raddr}, \text{size}), \text{states}, \\ & \quad \quad \quad \text{memory_read_side_effect}(\text{pm})) \\ & \quad \wedge (\text{address_block}(\text{raddr}, \text{size}) \subseteq \text{addresses}) \wedge \text{states}(s) \wedge \text{OK?}(q(s)) \\ & \quad \supset \\ & \quad \text{OK?}[\text{State}, \text{list}[\text{Byte}]] \\ & \quad \quad (\text{memory_read_list}[\text{State}](\text{pm})(\text{raddr}, \text{size})(\text{state}[\text{State}](q(s)))) \end{aligned} $

Repeatedly Skolemizing and flattening,

Installing automatic rewrites from: union singleton member union_subset1 has_next_state

Using lemma `unchanged_memory_invariant_next_ok`,

Simplifying, rewriting, and recording with decision procedures,

Using lemma `memory_read_list_ok`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Keeping (-4 1) and hiding *,

Forward chaining on `unchanged_memory_invariant_invariant`,

Hiding formulas: -2,

Using lemma `transformer_invariant_mono_transformers`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `unchanged_memory_read_list_TCC2`.

Q.E.D.

C.123.22 Memory_Change_2.unchanged_memory_read_list_TCC3

Terse proof for `unchanged_memory_read_list_TCC3`.

unchanged_memory_read_list_TCC3:

$\{1\} \quad \forall (\text{addresses: PRED}[\text{Address}], \text{pm: Memory_struct}[\text{State}], \text{states: PRED}[\text{State}], \text{raddr: Ad-}$ dress, $\text{size: nat, } s: \text{State, } q: [\text{State} \rightarrow \text{SuperResult}[\text{State}]]):$ $\text{unchanged_memory_invariant?}(\text{pm, states,}$ $\text{((memory_read_transformers}(\text{pm, address_block}(\text{raddr, size}))$ $\text{addresses)})$ \wedge $\text{transformers_ok?}(\text{states,}$ $\text{(memory_read_transformers}(\text{pm, address_block}(\text{raddr, size})) \cup \text{memory_re}$ \wedge $\text{side_effect_content_unchanged}(\text{address_block}(\text{raddr, size}), \text{states,}$ $\text{memory_read_side_effect}(\text{pm}))$ $\wedge (\text{address_block}(\text{raddr, size}) \subseteq \text{addresses}) \wedge \text{states}(s) \wedge \text{OK?}(q(s))$ $\supset \text{OK?}[\text{State, list}[\text{Byte}]](\text{memory_read_list}[\text{State}](\text{pm})(\text{raddr, size})(s))$
--

Repeatedly Skolemizing and flattening,

Using lemma memory_read_list_ok,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Keeping (-4 1) and hiding *,

Forward chaining on unchanged_memory_invariant_invariant,

Hiding formulas: -2,

Using lemma transformer_invariant_mono_transformers,

Installing automatic rewrites from: union_subset1

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of unchanged_memory_read_list_TCC3.

Q.E.D.

C.123.23 Memory_Change_2.unchanged_memory_read_list

Terse proof for unchanged_memory_read_list.

unchanged_memory_read_list:

$\{1\} \quad \forall (\text{addresses: PRED}[\text{Address}], \text{pm: Memory_struct}[\text{State}], \text{states: PRED}[\text{State}], \text{raddr: Ad-}$ dress, $\text{size: nat, } s: \text{State, } q: [\text{State} \rightarrow \text{SuperResult}[\text{State}]]):$ $\text{unchanged_memory_invariant?}(\text{pm, states,}$ $\text{((memory_read_transformers}(\text{pm, address_block}(\text{raddr, size}))$ $\text{addresses)})$ \wedge $\text{transformers_ok?}(\text{states,}$ $\text{(memory_read_transformers}(\text{pm, address_block}(\text{raddr, size})) \cup \text{memory_re}$ \wedge $\text{side_effect_content_unchanged}(\text{address_block}(\text{raddr, size}), \text{states,}$ $\text{memory_read_side_effect}(\text{pm}))$ $\wedge (\text{address_block}(\text{raddr, size}) \subseteq \text{addresses}) \wedge \text{states}(s) \wedge \text{OK?}(q(s))$ \supset $\text{data}(\text{memory_read_list}(\text{pm})(\text{raddr, size})(\text{state}(q(s)))) =$ $\text{data}(\text{memory_read_list}(\text{pm})(\text{raddr, size})(s))$
--

Repeatedly Skolemizing and flattening,

Installing automatic rewrites from: subset_equal union_subset1 union_subset3 has_next_state

Using lemma unchanged_memory_read_list_TCC2,

Using lemma unchanged_memory_read_list_TCC3,

Simplifying, rewriting, and recording with decision procedures,

Expanding the definition of memory_read_list,

Expanding the definition of ##,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Using lemma transformers_ok_mono_transformers,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Using lemma memory_read_list_nse_next_length,

Using lemma memory_read_list_nse_next_length,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 3 subgoals:

unchanged_memory_read_list.1:

{-1}	transformers_ok?(states', memory_read_transformers(pm', address_block(raddr', size')))
{-2}	states'(s')
{-3}	states'(state(memory_read_list_nse(pm')(raddr', size')(s')))
{-4}	length(data(memory_read_list_nse(pm')(raddr', size')(s'))) = size'
{-5}	states'(state(memory_read_list_nse(pm')(raddr', size')(state(q'(s')))))
{-6}	length(data(memory_read_list_nse(pm')(raddr', size')(state(q'(s'))))) = size'
{-7}	transformers_ok?(states', <div style="margin-left: 40px;">(memory_read_transformers(pm', address_block(raddr', size')) ∪ memory_read_side_eff</div>
{-8}	OK?(memory_read_list_nse(pm')(raddr', size')(s'))
{-9}	OK?[State, list[Byte]] <div style="margin-left: 40px;">(memory_read_side_effect(pm') <div style="margin-left: 40px;">(raddr', data(memory_read_list_nse(pm')(raddr', size')(s')), FALSE) <div style="margin-left: 40px;">(state(memory_read_list_nse(pm')(raddr', size')(s')))</div> </div> </div>
{-10}	OK?(memory_read_list_nse(pm')(raddr', size')(state[State](q'(s'))))
{-11}	OK?[State, list[Byte]] <div style="margin-left: 40px;">(memory_read_side_effect(pm') <div style="margin-left: 40px;">(raddr', <div style="margin-left: 40px;">data(memory_read_list_nse(pm')(raddr', size') <div style="margin-left: 40px;">(state[State](q'(s'))),</div> </div> </div> </div>

C Proof scripts

Using lemma `side_effect_content_unchanged_content [State]`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Replacing using formula -4,

Hiding formulas: -4,

Using lemma `side_effect_content_unchanged_content [State]`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Replacing using formula -4,

Hiding formulas: -4,

Using lemma `unchanged_memory_read_list_nse`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Keeping (-16 1) and hiding *,

Rewriting using `union_commutative`, matching in -1,

Rewriting using `union_associative`, matching in *,

Using lemma `unchanged_memory_invariant_mono`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `unchanged_memory_read_list.1`.

unchanged_memory_read_list.2:

{-1}	transformers_ok?(states', memory_read_transformers(pm', address_block(raddr', size')))
{-2}	states'(s')
{-3}	states'(state(memory_read_list_nse(pm')(raddr', size')(s')))
{-4}	length(data(memory_read_list_nse(pm')(raddr', size')(s'))) = size'
{-5}	transformers_ok?(states', (memory_read_transformers(pm', address_block(raddr', size')) \cup memory_read_side_eff
{-6}	OK?(memory_read_list_nse(pm')(raddr', size')(s'))
{-7}	OK?[State, list[Byte]] (memory_read_side_effect(pm') (raddr', data(memory_read_list_nse(pm')(raddr', size')(s')), FALSE) (state(memory_read_list_nse(pm')(raddr', size')(s'))))
{-8}	OK?(memory_read_list_nse(pm')(raddr', size')(state[State](q'(s'))))
{-9}	OK?[State, list[Byte]] (memory_read_side_effect(pm') (raddr', data(memory_read_list_nse(pm')(raddr', size') (state[State](q'(s')))), FALSE) (state(memory_read_list_nse(pm')(raddr', size') (state[State](q'(s'))))))
{-10}	size' \geq 0
{-11}	unchanged_memory_invariant?(pm', states', (memory_read_transformers(pm', address_block(raddr', size')) \cup memory addresses')
{-12}	side_effect_content_unchanged(address_block(raddr', size'), states', memory_read_side_effect(pm'))
{-13}	(address_block(raddr', size') \subseteq addresses')
{-14}	OK?(q'(s'))
{1}	states'(state(q'(s')))
{2}	data(memory_read_side_effect(pm') (raddr', data(memory_read_list_nse(pm')(raddr', size')(state(q'(s')))), FALSE) (state(memory_read_list_nse(pm')(raddr', size')(state(q'(s')))))) = data(memory_read_side_effect(pm') (raddr', data(memory_read_list_nse(pm')(raddr', size')(s')), FALSE) (state(memory_read_list_nse(pm')(raddr', size')(s'))))

Keeping (-11 1) and hiding *,

Installing automatic rewrites from: union member singleton

Using lemma unchanged_memory_invariant_next_ok[State],

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of unchanged_memory_read_list.2.

unchanged_memory_read_list.3:

{-1}	transformers_ok?(states', memory_read_transformers(pm', address_block(raddr', size')))
{-2}	states'(s')
{-3}	transformers_ok?(states', (memory_read_transformers(pm', address_block(raddr', size')) \cup memory_read_transformers(pm', address_block(raddr', size'))))
{-4}	OK?(memory_read_list_nse(pm')(raddr', size')(s'))
{-5}	OK?[State, list[Byte]] (memory_read_side_effect(pm') (raddr', data(memory_read_list_nse(pm')(raddr', size')(s')), FALSE) (state(memory_read_list_nse(pm')(raddr', size')(s'))))
{-6}	OK?(memory_read_list_nse(pm')(raddr', size')(state[State](q'(s'))))
{-7}	OK?[State, list[Byte]] (memory_read_side_effect(pm') (raddr', data(memory_read_list_nse(pm')(raddr', size') (state[State](q'(s')))), FALSE) (state(memory_read_list_nse(pm')(raddr', size') (state[State](q'(s'))))))
{-8}	size' \geq 0
{-9}	unchanged_memory_invariant?(pm', states', (memory_read_transformers(pm', address_block(raddr', size')) addresses'))
{-10}	side_effect_content_unchanged(address_block(raddr', size'), states', memory_read_side_effect(pm'))
{-11}	(address_block(raddr', size') \subseteq addresses')
{-12}	OK?(q'(s'))
{1}	transformer_invariant?(states', memory_read_transformers(pm', address_block(raddr', size')))
{2}	data(memory_read_side_effect(pm') (raddr', data(memory_read_list_nse(pm')(raddr', size')(state(q'(s')))), FALSE) (state(memory_read_list_nse(pm')(raddr', size')(state(q'(s')))))) = data(memory_read_side_effect(pm') (raddr', data(memory_read_list_nse(pm')(raddr', size')(s')), FALSE) (state(memory_read_list_nse(pm')(raddr', size')(s'))))

Keeping (-9 1) and hiding *,

Forward chaining on unchanged_memory_invariant_invariant,

Hiding formulas: -2,

Using lemma transformer_invariant_mono_transformers,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Hiding formulas: -1,

Using lemma transformer_invariant_mono_transformers,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of unchanged_memory_read_list.3.

Q.E.D.

C.123.24 Mem- ory_Change_2.unchanged_memory_invariant_unchanged_read_list_nse_TCC1

Terse proof for `unchanged_memory_invariant_unchanged_read_list_nse_TCC1`.

`unchanged_memory_invariant_unchanged_read_list_nse_TCC1`:

<pre> {1} ∃ (addresses: PRED[Address], pm: Memory_struct[State], states: PRED[State], s: State, addr_1, addr_2: Address, size: nat): unchanged_memory_invariant?(pm, states, memory_read_transformers(pm, ad- dress_block(addr_2, size)), addresses) ∧ transformers_ok?(states, memory_read_transformers(pm, addresses)) ∧ (address_block(addr_2, size) ⊆ addresses) ∧ addresses(addr_1) ∧ states(s) ⊃ OK?[State, list[Byte]](memory_read_list_nse[State](pm)(addr_2, size)(s)) ∨ Exception?[State, list[Byte]](memory_read_list_nse[State](pm)(addr_2, size)(s)) </pre>

Repeatedly Skolemizing and flattening,

Installing automatic rewrites from: `memory_read_transformers_mono`

Using lemma `memory_read_list_nse_ok`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 2 subgoals:

`unchanged_memory_invariant_unchanged_read_list_nse_TCC1.1`:

<pre> {-1} states'(s') {-2} size' ≥ 0 {-3} unchanged_memory_invariant?(pm', states', memory_read_transformers(pm', ad- dress_block(addr'', size')), addresses') {-4} transformers_ok?(states', memory_read_transformers(pm', addresses')) {-5} (address_block(addr'', size') ⊆ addresses') {-6} addresses'(addr') </pre> <hr/> <pre> {1} transformers_ok?(states', memory_read_transformers(pm', address_block(addr'', size'))) {2} OK?[State, list[Byte]](memory_read_list_nse[State](pm')(addr'', size')(s')) {3} Exception?[State, list[Byte]](memory_read_list_nse[State](pm')(addr'', size')(s')) </pre>
--

Keeping (-3 -4 1) and hiding *,

Using lemma `transformers_ok_mono_transformers`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `unchanged_memory_invariant_unchanged_read_list_nse_TCC1.1`.

unchanged_memory_invariant_unchanged_read_list_nse_TCC1.2:

{-1}	states'(s')
{-2}	size' ≥ 0
{-3}	unchanged_memory_invariant?(pm', states', memory_read_transformers(pm', ad- dress_block(addr'', size')), addresses')
{-4}	transformers_ok?(states', memory_read_transformers(pm', addresses'))
{-5}	(address_block(addr'', size') ⊆ addresses')
{-6}	addresses'(addr')
{1}	transformer_invariant?(states', memory_read_transformers(pm', ad- dress_block(addr'', size')))
{2}	OK?[State, list[Byte]](memory_read_list_nse[State](pm')(addr'', size')(s'))
{3}	Exception?[State, list[Byte]](memory_read_list_nse[State](pm')(addr'', size')(s'))

Forward chaining on unchanged_memory_invariant_invariant,
This completes the proof of unchanged_memory_invariant_unchanged_read_list_nse_TCC1.2.
Q.E.D.

C.123.25 Memory_Change_2.unchanged_memory_invariant_unchanged_read_list_nse_TCC2

Terse proof for unchanged_memory_invariant_unchanged_read_list_nse_TCC2.

unchanged_memory_invariant_unchanged_read_list_nse_TCC2:

{1}	\forall (addresses: PRED[Address], pm: Memory_struct[State], states: PRED[State], s: State, addr_1, addr_2: Address, size: nat): unchanged_memory_invariant?(pm, states, memory_read_transformers(pm, ad- dress_block(addr_2, size)), addresses) \wedge transformers_ok?(states, memory_read_transformers(pm, addresses)) \wedge (address_block(addr_2, size) ⊆ addresses) \wedge addresses(addr_1) \wedge states(s) \supset OK?[State, Byte] (memory_read(pm) (addr_1) (state[State, list[Byte]] (memory_read_list_nse[State](pm)(addr_2, size)(s))))
-----	--

Repeatedly Skolemizing and flattening,
Installing automatic rewrites from: memory_read_transformers_memory_read
Using lemma expr_transformers_ok_ok,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Hiding formulas: 2,
Using lemma memory_read_list_nse_next_length,
Forward chaining on unchanged_memory_invariant_invariant,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Installing automatic rewrites from: memory_read_transformers_mono
Using lemma transformers_ok_mono_transformers,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `unchanged_memory_invariant_unchanged_read_list_nse_TCC2`.
Q.E.D.

C.123.26 Mem- ory_Change_2.unchanged_memory_invariant_unchanged_read_list_nse_TCC3

Terse proof for `unchanged_memory_invariant_unchanged_read_list_nse_TCC3`.

`unchanged_memory_invariant_unchanged_read_list_nse_TCC3`:

$\{1\} \quad \forall (\text{addresses: PRED}[\text{Address}], \text{pm: Memory_struct}[\text{State}], \text{states: PRED}[\text{State}], s: \text{State},$ $\text{addr}_1, \text{addr}_2: \text{Address}, \text{size: nat}):$ $\text{unchanged_memory_invariant?}(\text{pm}, \text{states},$ $\text{memory_read_transformers}(\text{pm}, \text{ad-}$ $\text{dress_block}(\text{addr}_2, \text{size})),$ $\text{addresses})$ \wedge $\text{transformers_ok?}(\text{states}, \text{memory_read_transformers}(\text{pm}, \text{addresses})) \wedge$ $(\text{address_block}(\text{addr}_2, \text{size}) \subseteq \text{addresses}) \wedge \text{addresses}(\text{addr}_1) \wedge \text{states}(s)$ $\supset \text{OK?}[\text{State}, \text{Byte}] (\text{memory_read}(\text{pm})(\text{addr}_1)(s))$
--

Repeatedly Skolemizing and flattening,
Using lemma `expr_transformers_ok_ok`[State, Byte],
Installing automatic rewrites from: `memory_read_transformers_memory_read`
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `unchanged_memory_invariant_unchanged_read_list_nse_TCC3`.
Q.E.D.

C.123.27 Mem- ory_Change_2.unchanged_memory_invariant_unchanged_read_list_nse

Terse proof for `unchanged_memory_invariant_unchanged_read_list_nse`.

`unchanged_memory_invariant_unchanged_read_list_nse`:

$\{1\} \quad \forall (\text{addresses: PRED}[\text{Address}], \text{pm: Memory_struct}[\text{State}], \text{states: PRED}[\text{State}], s: \text{State},$ $\text{addr}_1, \text{addr}_2: \text{Address}, \text{size: nat}):$ $\text{unchanged_memory_invariant?}(\text{pm}, \text{states},$ $\text{memory_read_transformers}(\text{pm}, \text{ad-}$ $\text{dress_block}(\text{addr}_2, \text{size})),$ $\text{addresses})$ \wedge $\text{transformers_ok?}(\text{states}, \text{memory_read_transformers}(\text{pm}, \text{addresses})) \wedge$ $(\text{address_block}(\text{addr}_2, \text{size}) \subseteq \text{addresses}) \wedge \text{addresses}(\text{addr}_1) \wedge \text{states}(s)$ \supset $\text{data}(\text{memory_read}(\text{pm})(\text{addr}_1)(\text{state}(\text{memory_read_list_nse}(\text{pm})(\text{addr}_2, \text{size})(s))))$ $= \text{data}(\text{memory_read}(\text{pm})(\text{addr}_1)(s))$
--

For the top quantifier in 1, we introduce Skolem constants: $(\text{addresses}' \text{ pm}' \text{ states}' \text{ — } \text{addr}' \text{ — } \text{—})$,
Inducting on size on formula 1,
we get 5 subgoals:

unchanged_memory_invariant_unchanged_read_list_nse.1:

```
{1}  ∀ (s: State, addr_2: Address):
      unchanged_memory_invariant?(pm', states',
                                   memory_read_transformers(pm', ad-
dress_block(addr_2, 0)),
                                   addresses')
      ∧
      transformers_ok?(states', memory_read_transformers(pm', addresses')) ∧
      (address_block(addr_2, 0) ⊆ addresses') ∧ addresses'(addr') ∧ states'(s)
      ⊃
      data(memory_read(pm')
            (addr')(state(memory_read_list_nse(pm')(addr_2, 0)(s))))
      = data(memory_read(pm')(addr')(s))
```

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of unchanged_memory_invariant_unchanged_read_list_nse.1.

unchanged_memory_invariant_unchanged_read_list_nse.2:

```
{1}  ∀ j:
      (∀ (s: State, addr_2: Address):
        unchanged_memory_invariant?(pm', states',
                                      memory_read_transformers(pm', ad-
dress_block(addr_2, j)),
                                      addresses')
        ∧
        transformers_ok?(states', memory_read_transformers(pm', addresses')) ∧
        (address_block(addr_2, j) ⊆ addresses') ∧ addresses'(addr') ∧ states'(s)
        ⊃
        data(memory_read(pm')
              (addr')(state(memory_read_list_nse(pm')(addr_2, j)(s))))
        = data(memory_read(pm')(addr')(s)))
      ⊃
      (∀ (s: State, addr_2: Address):
        unchanged_memory_invariant?(pm', states',
                                      memory_read_transformers(pm',
                                                                address_block
                                                                (addr_2, j + 1)),
                                      addresses')
        ∧
        transformers_ok?(states', memory_read_transformers(pm', addresses')) ∧
        (address_block(addr_2, j + 1) ⊆ addresses') ∧
        addresses'(addr') ∧ states'(s)
        ⊃
        data(memory_read(pm')
              (addr')
              (state(memory_read_list_nse(pm')(addr_2, j + 1)(s))))
        = data(memory_read(pm')(addr')(s)))
```

Repeatedly Skolemizing and flattening,

Forward chaining on unchanged_memory_invariant_invariant,

Installing automatic rewrites from: memory_read_transformers_mono ok_result address_block_subset_1

address_block_subset_2 address_block has_next_state memory_read_transformers_memory_read_subset_equal

Using lemma transformers_ok_mono_transformers,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 Using lemma memory_read_list_nse_ok,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 Expanding the definition of memory_read_list_nse,
 Expanding the definition of memory_read_list_nse,
 Expanding the definition of ##,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 Hiding formulas: -6,
 Using lemma expr_transformer_invariant_next_ok[State, Byte],
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 Instantiating quantified variables,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 we get 2 subgoals:

unchanged_memory_invariant_unchanged_read_list_nse.2.1:

{-1}	transformer_invariant?(states', memory_read_transformers(pm', ad- dress_block(addr'', 1 + j')))
{-2}	states'(s')
{-3}	states'(state(memory_read(pm')(addr'')(s')))
{-4}	transformers_ok?(states', memory_read_transformers(pm', address_block(addr'', 1 + j')))
{-5}	OK?(memory_read(pm')(addr'')(s'))
{-6}	OK?(memory_read_list_nse(pm')(addr'' + 1, j') (state(memory_read(pm')(addr'')(s'))))
{-7}	transformers_ok?(states', memory_read_transformers(pm', addresses'))
{-8}	j' ≥ 0
{-9}	data(memory_read(pm') (addr') (state(memory_read_list_nse(pm')(addr'' + 1, j') (state(memory_read(pm')(addr'')(s'))))))
{-10}	= data(memory_read(pm')(addr')(state(memory_read(pm')(addr'')(s')))) unchanged_memory_invariant?(pm', states', memory_read_transformers(pm', ad- dress_block(addr'', 1 + j')), addresses')
{-11}	(address_block(addr'', 1 + j') ⊆ addresses')
{-12}	addresses'(addr')
{1}	data(memory_read(pm') (addr') (state(memory_read_list_nse(pm')(addr'' + 1, j') (state(memory_read(pm')(addr'')(s')))))) = data(memory_read(pm')(addr')(s'))

Replacing using formula -9,
 Hiding formulas: -9,
 Using lemma expr_unchanged_memory_invariant_unchanged[State, Byte],
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 we get 2 subgoals:

unchanged_memory_invariant_unchanged_read_list_nse.2.1.1.1:

{-1}	unchanged_memory_invariant?(pm', states', memory_read_transformers(pm', ad- dress_block(addr'', 1 + j')), addresses')
{-2}	states'(s')
{-3}	addresses'(addr')
{-4}	OK?(memory_read(pm')(addr'')(s'))
{-5}	transformer_invariant?(states', memory_read_transformers(pm', ad- dress_block(addr'', 1 + j')))
{-6}	states'(state(memory_read(pm')(addr'')(s')))
{-7}	transformers_ok?(states', memory_read_transformers(pm', address_block(addr'', 1 + j')))
{-8}	OK?(memory_read_list_nse(pm')(addr'' + 1, j') (state(memory_read(pm')(addr'')(s'))))
{-9}	transformers_ok?(states', memory_read_transformers(pm', addresses'))
{-10}	$j' \geq 0$
{-11}	(address_block(addr'', 1 + j') \subseteq addresses')
{1}	OK?(memory_read(pm')(addr')(state(memory_read(pm')(addr'')(s'))))
{2}	data(memory_read(pm')(addr')(state(memory_read(pm')(addr'')(s')))) = data(memory_read(pm')(addr')(s'))

Keeping (-3 -6 -9 1) and hiding *,

Using lemma expr_transformers_ok_ok[State, Byte],

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of unchanged_memory_invariant_unchanged_read_list_nse.2.1.1.

unchanged_memory_invariant_unchanged_read_list_nse.2.1.1.2:

{-1}	unchanged_memory_invariant?(pm', states', memory_read_transformers(pm', ad- dress_block(addr'', 1 + j')), addresses')
{-2}	states'(s')
{-3}	addresses'(addr')
{-4}	OK?(memory_read(pm')(addr'')(s'))
{-5}	transformer_invariant?(states', memory_read_transformers(pm', ad- dress_block(addr'', 1 + j')))
{-6}	states'(state(memory_read(pm')(addr'')(s')))
{-7}	transformers_ok?(states', memory_read_transformers(pm', address_block(addr'', 1 + j')))
{-8}	OK?(memory_read_list_nse(pm')(addr'' + 1, j') (state(memory_read(pm')(addr'')(s'))))
{-9}	transformers_ok?(states', memory_read_transformers(pm', addresses'))
{-10}	$j' \geq 0$
{-11}	(address_block(addr'', 1 + j') \subseteq addresses')
{1}	OK?(memory_read(pm')(addr')(s'))
{2}	data(memory_read(pm')(addr')(state(memory_read(pm')(addr'')(s')))) = data(memory_read(pm')(addr')(s'))

Keeping (-3 -9 1) and hiding *,

Using lemma `expr_transformers_ok_ok`[State, Byte],

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `unchanged_memory_invariant_unchanged_read_list_nse.2.1.2`.

`unchanged_memory_invariant_unchanged_read_list_nse.2.2`:

{-1}	transformer_invariant?(states', memory_read_transformers(pm', ad- dress_block(addr'', 1 + j')))
{-2}	states'(s')
{-3}	states'(state(memory_read(pm')(addr'')(s')))
{-4}	transformers_ok?(states', memory_read_transformers(pm', address_block(addr'', 1 + j')))
{-5}	OK?(memory_read(pm')(addr'')(s'))
{-6}	OK?(memory_read_list_nse(pm')(addr'' + 1, j') (state(memory_read(pm')(addr'')(s'))))
{-7}	transformers_ok?(states', memory_read_transformers(pm', addresses'))
{-8}	$j' \geq 0$
{-9}	unchanged_memory_invariant?(pm', states', memory_read_transformers(pm', ad- dress_block(addr'', 1 + j')), addresses')
{-10}	(address_block(addr'', 1 + j') \subseteq addresses')
{-11}	addresses'(addr')
{1}	unchanged_memory_invariant?(pm', states', memory_read_transformers(pm', ad- dress_block(addr'' + 1, j')), addresses')
{2}	data(memory_read(pm') (addr') (state(memory_read_list_nse(pm')(addr'' + 1, j') (state(memory_read(pm')(addr'')(s')))))) = data(memory_read(pm')(addr')(s'))

Keeping (-9 1) and hiding *,

Using lemma `unchanged_memory_invariant_mono`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `unchanged_memory_invariant_unchanged_read_list_nse.2.2`.

unchanged_memory_invariant_unchanged_read_list_nse.3:

{1}	$\forall (\text{size: nat, } s: \text{State, } \text{addr_2: Address}):$ $\text{unchanged_memory_invariant?}(\text{pm}', \text{states}',$ $\text{memory_read_transformers}(\text{pm}', \text{ad-}$ $\text{dress_block}(\text{addr_2}, \text{size})),$ $\text{addresses}')$ \wedge $\text{transformers_ok?}(\text{states}', \text{memory_read_transformers}(\text{pm}', \text{addresses}')) \wedge$ $(\text{address_block}(\text{addr_2}, \text{size}) \subseteq \text{addresses}') \wedge \text{addresses}'(\text{addr}') \wedge \text{states}'(s)$ $\supset \text{OK?}[\text{State, Byte}](\text{memory_read}(\text{pm}')(\text{addr}') (s))$
{2}	$\forall (s: \text{State, } \text{addr_2: Address}):$ $\text{unchanged_memory_invariant?}(\text{pm}', \text{states}',$ $\text{memory_read_transformers}(\text{pm}', \text{ad-}$ $\text{dress_block}(\text{addr_2}, \text{size}')),$ $\text{addresses}')$ \wedge $\text{transformers_ok?}(\text{states}', \text{memory_read_transformers}(\text{pm}', \text{addresses}')) \wedge$ $(\text{address_block}(\text{addr_2}, \text{size}') \subseteq \text{addresses}') \wedge \text{addresses}'(\text{addr}') \wedge \text{states}'(s)$ \supset $\text{data}(\text{memory_read}(\text{pm}'$ $(\text{addr}')(\text{state}(\text{memory_read_list_nse}(\text{pm}')(\text{addr_2}, \text{size}') (s))))$ $= \text{data}(\text{memory_read}(\text{pm}')(\text{addr}') (s))$

Hiding formulas: 2,

Repeatedly Skolemizing and flattening,

Using lemma `unchanged_memory_invariant_unchanged_read_list_nse_TCC3`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `unchanged_memory_invariant_unchanged_read_list_nse.3`.

unchanged_memory_invariant_unchanged_read_list_nse.4:

```

{1}  ∀ (size: nat, s: State, addr_2: Address):
      unchanged_memory_invariant?(pm', states',
                                   memory_read_transformers(pm', ad-
dress_block(addr_2, size)),
                                   addresses')
      ∧
      transformers_ok?(states', memory_read_transformers(pm', addresses')) ∧
      (address_block(addr_2, size) ⊆ addresses') ∧ addresses'(addr') ∧ states'(s)
      ⊃
      OK?[State, Byte]
        (memory_read(pm')
         (addr')
         (state[State, list[Byte]]
          (memory_read_list_nse[State](pm')(addr_2, size)(s))))
{2}  ∀ (s: State, addr_2: Address):
      unchanged_memory_invariant?(pm', states',
                                   memory_read_transformers(pm', ad-
dress_block(addr_2, size')),
                                   addresses')
      ∧
      transformers_ok?(states', memory_read_transformers(pm', addresses')) ∧
      (address_block(addr_2, size') ⊆ addresses') ∧ addresses'(addr') ∧ states'(s)
      ⊃
      data(memory_read(pm')
           (addr')(state(memory_read_list_nse(pm')(addr_2, size')(s))))
      = data(memory_read(pm')(addr')(s))

```

Hiding formulas: 2,

Repeatedly Skolemizing and flattening,

Using lemma unchanged_memory_invariant_unchanged_read_list_nse_TCC2,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of unchanged_memory_invariant_unchanged_read_list_nse.4.

unchanged_memory_invariant_unchanged_read_list_nse.5:

```

{1}  ∃ (size: nat, s: State, addr_2: Address):
      unchanged_memory_invariant?(pm', states',
                                   memory_read_transformers(pm', ad-
dress_block(addr_2, size)),
                                   addresses')
      ∧
      transformers_ok?(states', memory_read_transformers(pm', addresses')) ∧
      (address_block(addr_2, size) ⊆ addresses') ∧ addresses'(addr') ∧ states'(s)
      ⊃
      OK?[State, list[Byte]](memory_read_list_nse[State](pm')(addr_2, size)(s)) ∨
      Exception?[State, list[Byte]](memory_read_list_nse[State](pm')(addr_2, size)(s))
{2}  ∃ (s: State, addr_2: Address):
      unchanged_memory_invariant?(pm', states',
                                   memory_read_transformers(pm', ad-
dress_block(addr_2, size')),
                                   addresses')
      ∧
      transformers_ok?(states', memory_read_transformers(pm', addresses')) ∧
      (address_block(addr_2, size') ⊆ addresses') ∧ addresses'(addr') ∧ states'(s)
      ⊃
      data(memory_read(pm')
            (addr')(state(memory_read_list_nse(pm')(addr_2, size')(s))))
      = data(memory_read(pm')(addr')(s))

```

Hiding formulas: 2,

Repeatedly Skolemizing and flattening,

Using lemma unchanged_memory_invariant_unchanged_read_list_nse_TCC1,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of unchanged_memory_invariant_unchanged_read_list_nse.5.

Q.E.D.

C.123.28 Memory_Change_2.memory_write_list_ok_nse

Terse proof for memory_write_list_ok_nse.

memory_write_list_ok_nse:

```

{1}  ∃ (pm: Memory_struct[State], states: PRED[State], waddr: Address, s: State,
      bl: list[Byte]):
      transformer_invariant?(states,
                              memory_write_transformers(pm, ad-
dress_block(waddr, length(bl))))
      ∧
      transformers_ok?(states,
                      memory_write_transformers(pm, ad-
dress_block(waddr, length(bl))))
      ∧ states(s)
      ⊃ OK?(memory_write_list_nse(pm)(waddr, bl)(s))

```

For the top quantifier in 1, we introduce Skolem constants: (pm' states'),

Inducting on bl on formula 1,

we get 2 subgoals:

`memory_write_list_ok_nse.1:`

$\{1\} \quad \forall (waddr: \text{Address}, s: \text{State}):$ $\quad \text{transformer_invariant?}(\text{states}',$ $\quad \quad \quad \text{memory_write_transformers}(\text{pm}',$ $\quad \quad \quad \quad \quad \text{ad-}$ $\quad \text{dress_block}(waddr, \text{length}(\text{null})))$ $\quad \wedge$ $\quad \text{transformers_ok?}(\text{states}',$ $\quad \quad \quad \text{memory_write_transformers}(\text{pm}', \text{ad-}$ $\quad \text{dress_block}(waddr, \text{length}(\text{null})))$ $\quad \wedge \text{states}'(s)$ $\quad \supset \text{OK?}(\text{memory_write_list_nse}(\text{pm}')(\text{waddr}, \text{null})(s))$

Expanding the definition of `memory_write_list_nse`,

Expanding the definition of `ok_result`,

which is trivially true.

This completes the proof of `memory_write_list_ok_nse.1`.

memory_write_list_ok_nse.2:

```

{1}  ∀ (cons1_var: Byte, cons2_var: list[Byte]):
      (∀ (waddr: Address, s: State):
        transformer_invariant?(states',
                                memory_write_transformers(pm',
                                                            address_block(waddr,
                                                                           length(cons2_
                                                                    &
                                                                    transformers_ok?(states',
                                                                memory_write_transformers(pm',
                                                                    ad-
                                                                    dress_block(waddr, length(cons2_var))))
                                                                    & states'(s))
                                                                    ⊃ OK?(memory_write_list_nse(pm')(waddr, cons2_var)(s)))
                                                                    ⊃
                                                                    (∀ (waddr: Address, s: State):
                                                                      transformer_invariant?(states',
                                                                          memory_write_transformers(pm',
                                                                              ad-
                                                                              dress_block(waddr,
                                                                                          length
                                                                                          (cons
                                                                                          (cons1_var,
                                                                                          cons2_v
                                                                    &
                                                                    transformers_ok?(states',
                                                                memory_write_transformers(pm',
                                                                    address_block(waddr,
                                                                           length
                                                                           (cons
                                                                           (cons1_var,
                                                                           cons2_var))))
                                                                    & states'(s))
                                                                    ⊃ OK?(memory_write_list_nse(pm')(waddr, cons(cons1_var, cons2_var))(s)))

```

Repeatedly Skolemizing and flattening,

Installing automatic rewrites from: memory_write_transformers_memory_write address_block length comp_expr_forget_expr has_next_state address_block_subset_1 memory_write_transformers_mono

Expanding the definition of memory_write_list_nse,

Using lemma expr_transformers_ok_ok,

Simplifying, rewriting, and recording with decision procedures,

Simplifying, rewriting, and recording with decision procedures,

Instantiating quantified variables,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 3 subgoals:

memory_write_list_ok_nse.2.1:

{-1}	OK?(memory_write(pm')(waddr', cons1_var')(s'))
{-2}	transformer_invariant?(states', memory_write_transformers(pm', address_block(waddr', 1 + length(cons2_var'))))
{-3}	transformers_ok?(states', memory_write_transformers(pm', address_block(waddr', 1 + length(cons2_var'))))
{-4}	states'(s')
{1}	states'(state(memory_write(pm')(waddr', cons1_var')(s')))
{2}	OK?(memory_write_list_nse(pm')(waddr' + 1, cons2_var') (state(memory_write(pm')(waddr', cons1_var')(s'))))

Using lemma `expr_transformer_invariant_next_ok` [State, Unit],

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `memory_write_list_ok_nse.2.1`.

memory_write_list_ok_nse.2.2:

{-1}	OK?(memory_write(pm')(waddr', cons1_var')(s'))
{-2}	transformer_invariant?(states', memory_write_transformers(pm', address_block(waddr', 1 + length(cons2_var'))))
{-3}	transformers_ok?(states', memory_write_transformers(pm', address_block(waddr', 1 + length(cons2_var'))))
{-4}	states'(s')
{1}	transformers_ok?(states', memory_write_transformers(pm', address_block(waddr' + 1, length(cons2_var'))))
{2}	OK?(memory_write_list_nse(pm')(waddr' + 1, cons2_var') (state(memory_write(pm')(waddr', cons1_var')(s'))))

Keeping (-3 1) and hiding *,

Using lemma `transformers_ok_mono_transformers`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `memory_write_list_ok_nse.2.2`.

memory_write_list_ok_nse.2.3:

{-1}	OK?(memory_write(pm')(waddr', cons1_var')(s'))
{-2}	transformer_invariant?(states', memory_write_transformers(pm', address_block(waddr', 1 + length(cons2_var'))
{-3}	transformers_ok?(states', memory_write_transformers(pm', address_block(waddr', 1 + length(cons2_var'))
{-4}	states'(s')
{1}	transformer_invariant?(states', memory_write_transformers(pm', address_block(waddr' + 1, length(cons2_var'))
{2}	OK?(memory_write_list_nse(pm')(waddr' + 1, cons2_var') (state(memory_write(pm')(waddr', cons1_var')(s'))))

Keeping (-2 1) and hiding *,
 Using lemma transformer_invariant_mono_transformers,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 This completes the proof of memory_write_list_ok_nse.2.3.
 Q.E.D.

C.123.29 Memory_Change_2.memory_write_list_ok

Terse proof for memory_write_list_ok.

memory_write_list_ok:

{1}	\forall (pm: Memory_struct[State], states: PRED[State], waddr: Address, s: State, bl: list[Byte]): transformer_invariant?(states, (memory_write_transformers(pm, address_block(waddr, length(bl))) ^ transformers_ok?(states, (memory_write_transformers(pm, address_block(waddr, length(bl)))) \cup mem- ^ side_effect_content_unchanged(address_block(waddr, length(bl)), states, memory_write_side_effect(pm)) ^ states(s) \supset OK?(memory_write_list(pm)(waddr, bl)(s))
-----	---

Repeatedly Skolemizing and flattening,
 Hiding formulas: -1,
 Installing automatic rewrites from: union member has_next_state union_subset1 mem-
 ory_write_side_effect_super_transformers_memory_write_side_effect subset_equal
 Expanding the definition of memory_write_list,
 Using lemma expr_transformers_ok_ok,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 Expanding the definition of ##,
 Using lemma side_effect_content_unchanged_content[State],
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Using lemma `memory_write_list_ok_nse`,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 we get 3 subgoals:

`memory_write_list_ok.1`:

{-1}	<code>side_effect_content_unchanged(address_block(waddr', length(bl')), states', memory_write_side_effect(pm'))</code>
{-2}	<code>states'(s')</code>
{-3}	<code>OK?(memory_write_side_effect(pm')(waddr', bl', FALSE)(s'))</code>
{-4}	<code>data(memory_write_side_effect(pm')(waddr', bl', FALSE)(s')) = bl'</code>
{-5}	<code>transformers_ok?(states', (memory_write_transformers(pm', address_block(waddr', length(bl')))) ∪ memory_write_</code>
{-6}	<code>transformer_invariant?(states', (memory_write_transformers(pm', address_block(waddr', length(bl')))) ∪ memory</code>
{1}	<code>states'(state(memory_write_side_effect(pm')(waddr', bl', FALSE)(s')))</code>
{2}	<code>OK?(memory_write_list_nse(pm') (waddr', data(memory_write_side_effect(pm') (waddr', bl', FALSE)(s')) (state(memory_write_side_effect(pm') (waddr', bl', FALSE)(s'))))</code>

Keeping (-6 1) and hiding *,
 Using lemma `expr_transformer_invariant_next_ok` [State, list[Byte]],
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 This completes the proof of `memory_write_list_ok.1`.

`memory_write_list_ok.2`:

{-1}	<code>side_effect_content_unchanged(address_block(waddr', length(bl')), states', memory_write_side_effect(pm'))</code>
{-2}	<code>states'(s')</code>
{-3}	<code>OK?(memory_write_side_effect(pm')(waddr', bl', FALSE)(s'))</code>
{-4}	<code>data(memory_write_side_effect(pm')(waddr', bl', FALSE)(s')) = bl'</code>
{-5}	<code>transformers_ok?(states', (memory_write_transformers(pm', address_block(waddr', length(bl')))) ∪ memory_write_</code>
{-6}	<code>transformer_invariant?(states', (memory_write_transformers(pm', address_block(waddr', length(bl')))) ∪ memory</code>
{1}	<code>transformers_ok?(states', memory_write_transformers(pm', address_block(waddr', length (data (memory_write_side_effect(pm') (waddr', bl', FALSE) (s'))))</code>
{2}	<code>OK?(memory_write_list_nse(pm') (waddr', data(memory_write_side_effect(pm') (waddr', bl', FALSE)(s')) (state(memory_write_side_effect(pm') (waddr', bl', FALSE)(s'))))</code>

Replacing using formula -4,
 Keeping (-5 1) and hiding *,
 Using lemma `transformers_ok_mono_transformers`,

C Proof scripts

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `memory_write_list_ok.2`.

`memory_write_list_ok.3`:

<pre> {-1} side_effect_content_unchanged(address_block(waddr', length(bl')), states', memory_write_side_effect(pm')) {-2} states'(s') {-3} OK?(memory_write_side_effect(pm')(waddr', bl', FALSE)(s')) {-4} data(memory_write_side_effect(pm')(waddr', bl', FALSE)(s')) = bl' {-5} transformers_ok?(states', (memory_write_transformers(pm', address_block(waddr', length(bl')))) ∪ mem {-6} transformer_invariant?(states', (memory_write_transformers(pm', address_block(waddr', length(bl')))) </pre>	<pre> </pre>
<pre> {1} transformer_invariant?(states', memory_write_transformers(pm', address_block(waddr', length (data (memory_write_si (pm') (waddr', bl', (s')))))) </pre>	<pre> </pre>
<pre> {2} OK?(memory_write_list_nse(pm') (waddr', data(memory_write_side_effect(pm') (waddr', bl', FALSE)(s')) (state(memory_write_side_effect(pm') (waddr', bl', FALSE)(s')))) </pre>	<pre> </pre>

Replacing using formula -4,

Keeping (-6 1) and hiding *,

Using lemma `transformer_invariant_mono_transformers`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `memory_write_list_ok.3`.

Q.E.D.

C.123.30 Mem- ory_Change_2.unchanged_memory_invariant_unchanged_write_list_nse_TCC1

Terse proof for `unchanged_memory_invariant_unchanged_write_list_nse_TCC1`.

unchanged_memory_invariant_unchanged_write_list_nse_TCC1:

<pre> {1} ∀ (pm: Memory_struct[State], states: PRED[State], s: State, raddr, waddr: Ad- dress, bl: list[Byte]): unchanged_memory_invariant?(pm, states, memory_write_transformers(pm, ad- dress_block(waddr, length(bl))), singleton(raddr)) ^ transformers_ok?(states, memory_write_transformers(pm, ad- dress_block(waddr, length(bl)))) ^ transformers_ok?(states, singleton(expr_2_super(memory_read(pm)(raddr)))) ^ states(s) ⊃ OK?[State, Unit](memory_write_list_nse[State](pm)(waddr, bl)(s)) ∨ Exception?[State, Unit](memory_write_list_nse[State](pm)(waddr, bl)(s)) </pre>
--

Repeatedly Skolemizing and flattening,

Using lemma `memory_write_list_ok_nse`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Forward chaining on `unchanged_memory_invariant_invariant`,

This completes the proof of `unchanged_memory_invariant_unchanged_write_list_nse_TCC1`.

Q.E.D.

C.123.31 Mem- ory_Change_2.unchanged_memory_invariant_unchanged_write_list_nse_TCC2

Terse proof for `unchanged_memory_invariant_unchanged_write_list_nse_TCC2`.

unchanged_memory_invariant_unchanged_write_list_nse_TCC2:

```

{1}  ∃ (pm: Memory_struct[State], states: PRED[State], s: State, raddr, waddr: Ad-
      dress,
      bl: list[Byte]):
      unchanged_memory_invariant?(pm, states,
                                  memory_write_transformers(pm, ad-
dress_block(waddr, length(bl))),
                                  singleton(raddr))
      ∧
      transformers_ok?(states,
                       memory_write_transformers(pm, ad-
dress_block(waddr, length(bl))))
      ∧
      transformers_ok?(states, singleton(expr_2_super(memory_read(pm)(raddr)))) ∧
      states(s)
      ⊃
      OK?[State, Byte]
        (memory_read(pm)
         (raddr)
         (state[State, Unit](memory_write_list_nse[State](pm)(waddr, bl)(s))))

```

Repeatedly Skolemizing and flattening,

Installing automatic rewrites from: singleton

Using lemma expr_transformers_ok_ok[State, Byte],

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Forward chaining on unchanged_memory_invariant_invariant,

Using lemma memory_write_list_ok_nse,

Simplifying, rewriting, and recording with decision procedures,

Using lemma transformer_invariant_write_list_nse,

Simplifying, rewriting, and recording with decision procedures,

Expanding the definition of result_pred,

Installing automatic rewrites from: has_next_state_expr_2_super has_next_state state_expr_2_super

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of unchanged_memory_invariant_unchanged_write_list_nse_TCC2.

Q.E.D.

C.123.32 Memory_Change_2.unchanged_memory_invariant_unchanged_write_list_nse_TCC3

Terse proof for unchanged_memory_invariant_unchanged_write_list_nse_TCC3.

unchanged_memory_invariant_unchanged_write_list_nse_TCC3:

```
{1}  ∀ (pm: Memory_struct[State], states: PRED[State], s: State, raddr, waddr: Ad-
dress,
      bl: list[Byte]):
      unchanged_memory_invariant?(pm, states,
                                  memory_write_transformers(pm,
                                                              ad-
dress_block(waddr, length(bl))),
                                  singleton(raddr))
      ∧
      transformers_ok?(states,
                       memory_write_transformers(pm, ad-
dress_block(waddr, length(bl))))
      ∧
      transformers_ok?(states, singleton(expr_2_super(memory_read(pm)(raddr)))) ∧
      states(s)
      ⊃ OK?[State, Byte](memory_read(pm)(raddr)(s))
```

Repeatedly Skolemizing and flattening,

Installing automatic rewrites from: singleton

Using lemma `expr_transformers_ok_ok` [State, Byte],

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `unchanged_memory_invariant_unchanged_write_list_nse_TCC3`.

Q.E.D.

C.123.33 Memory_Change_2.unchanged_memory_invariant_unchanged_write_list_nse

Terse proof for `unchanged_memory_invariant_unchanged_write_list_nse`.

unchanged_memory_invariant_unchanged_write_list_nse:

```
{1}  ∀ (pm: Memory_struct[State], states: PRED[State], s: State, raddr, waddr: Ad-
dress,
      bl: list[Byte]):
      unchanged_memory_invariant?(pm, states,
                                  memory_write_transformers(pm,
                                                              ad-
dress_block(waddr, length(bl))),
                                  singleton(raddr))
      ∧
      transformers_ok?(states,
                       memory_write_transformers(pm, ad-
dress_block(waddr, length(bl))))
      ∧
      transformers_ok?(states, singleton(expr_2_super(memory_read(pm)(raddr)))) ∧
      states(s)
      ⊃ data(memory_read(pm)(raddr)(state(memory_write_list_nse(pm)(waddr, bl)(s)))) =
      data(memory_read(pm)(raddr)(s))
```

C Proof scripts

For the top quantifier in 1, we introduce Skolem constants: (pm' states' — raddr' — —),

Inducting on bl on formula 1,

we get 5 subgoals:

unchanged_memory_invariant_unchanged_write_list_nse.1:

$$\begin{array}{l}
 \{1\} \quad \forall (s: \text{State}, waddr: \text{Address}): \\
 \quad \text{unchanged_memory_invariant?}(pm', \text{states}', \\
 \quad \quad \quad \text{memory_write_transformers}(pm', \\
 \quad \quad \quad \quad \quad \quad \text{ad-} \\
 \quad \text{dress_block}(waddr, \text{length}(\text{null}))), \\
 \quad \quad \quad \quad \quad \quad \text{singleton}(raddr')) \\
 \quad \wedge \\
 \quad \text{transformers_ok?}(\text{states}', \\
 \quad \quad \quad \text{memory_write_transformers}(pm', \text{ad-} \\
 \quad \text{dress_block}(waddr, \text{length}(\text{null})))) \\
 \quad \wedge \\
 \quad \text{transformers_ok?}(\text{states}', \text{singleton}(\text{expr_2_super}(\text{memory_read}(pm')(raddr')))) \wedge \\
 \quad \quad \text{states}'(s) \\
 \quad \supset \\
 \quad \text{data}(\text{memory_read}(pm') \\
 \quad \quad \quad (raddr'))(\text{state}(\text{memory_write_list_nse}(pm')(waddr, \text{null})(s))) \\
 \quad = \text{data}(\text{memory_read}(pm')(raddr'))(s)
 \end{array}$$

Expanding the definition of memory_write_list_nse,

Expanding the definition of ok_result,

which is trivially true.

This completes the proof of unchanged_memory_invariant_unchanged_write_list_nse.1.

unchanged_memory_invariant_unchanged_write_list_nse.2:

```

{1}  ∀ (cons1_var: Byte, cons2_var: list[Byte]):
      (∀ (s: State, waddr: Address):
        unchanged_memory_invariant?(pm', states',
                                     memory_write_transformers(pm',
                                                                address_block
                                                                (waddr, length(cons2_var))),
                                     singleton(raddr'))
          ∧
          transformers_ok?(states',
                           memory_write_transformers(pm',
                                                       address_block(waddr, length(cons2_var))))
          ∧
          transformers_ok?(states', singleton(expr_2_super(memory_read(pm')(raddr'))))
          ∧ states'(s)
        ⊃
        data(memory_read(pm')
              (raddr')(state(memory_write_list_nse(pm')(waddr, cons2_var)(s))))
          = data(memory_read(pm')(raddr')(s)))
        ⊃
        (∀ (s: State, waddr: Address):
          unchanged_memory_invariant?(pm', states',
                                       memory_write_transformers(pm',
                                                                  address_block
                                                                  (waddr,
                                                                    length
                                                                    (cons
                                                                      (cons1_var, cons2_var))))),
                                       singleton(raddr'))
            ∧
            transformers_ok?(states',
                              memory_write_transformers(pm',
                                                          address_block(waddr,
                                                                        length
                                                                        (cons
                                                                          (cons1_var,
                                                                           cons2_var))))))
            ∧
            transformers_ok?(states', singleton(expr_2_super(memory_read(pm')(raddr'))))
            ∧ states'(s)
          ⊃
          data(memory_read(pm')
                (raddr')
                (state(memory_write_list_nse(pm')(waddr, cons(cons1_var, cons2_var))
                    (s))))
            = data(memory_read(pm')(raddr')(s)))
        )
      )

```

Repeatedly Skolemizing and flattening,

Forward chaining on unchanged_memory_invariant_invariant,

Using lemma memory_write_list_ok_nse,

C Proof scripts

Simplifying, rewriting, and recording with decision procedures,

```
Installing automatic rewrites from: memory_write_transformers_memory_write singleton ad-
dress_block_subset_1 memory_write_transformers_mono length ## address_block has_next_state
subset_equal
```

Using lemma `expr_transformers_ok_ok`,

Simplifying, rewriting, and recording with decision procedures,

Using lemma `expr_transformers_ok_ok`,

Simplifying, rewriting, and recording with decision procedures,

Using lemma `expr_transformer_invariant_next_ok`,

Simplifying, rewriting, and recording with decision procedures,

Using lemma `expr_transformers_ok_ok`,

Simplifying, rewriting, and recording with decision procedures,

Expanding the definition of `memory_write_list_nse`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Rewriting using `-7`, matching in `*`,

we get 3 subgoals:

unchanged_memory_invariant_unchanged_write_list_nse.2.1:

{-1}	OK?(memory_read(pm') (raddr')(state(memory_write(pm')(waddr', cons1_var')(s')))))
{-2}	states'(state(memory_write(pm')(waddr', cons1_var')(s')))
{-3}	OK?(memory_write(pm')(waddr', cons1_var')(s'))
{-4}	OK?(memory_read(pm')(raddr')(s'))
{-5}	OK?(memory_write_list_nse(pm')(waddr', cons(cons1_var', cons2_var'))(s'))
{-6}	transformer_invariant?(states', memory_write_transformers(pm', address_block(waddr', 1 + length(cons2_var'))))
{-7}	$\forall (s: \text{State}, waddr: \text{Address}):$ unchanged_memory_invariant?(pm', states', memory_write_transformers(pm', address_block(waddr, length(cons2_var'))), singleton(raddr')) \wedge transformers_ok?(states', memory_write_transformers(pm', address_block(waddr, length(cons2_var')))) \wedge states'(s) \supset data(memory_read(pm') (raddr')(state(memory_write_list_nse(pm')(waddr, cons2_var')(s)))) = data(memory_read(pm')(raddr')(s))
{-8}	unchanged_memory_invariant?(pm', states', memory_write_transformers(pm', address_block(waddr', 1 + length(cons2_var'))), singleton(raddr'))
{-9}	transformers_ok?(states', memory_write_transformers(pm', address_block(waddr', 1 + length(cons2_var'))))
{-10}	transformers_ok?(states', singleton(expr_2_super(memory_read(pm')(raddr'))))
{-11}	states'(s')
{1}	data(memory_read(pm') (raddr')(state(memory_write(pm')(waddr', cons1_var')(s')))) = data(memory_read(pm')(raddr')(s'))

Hiding formulas: -7,

Using lemma expr_unchanged_memory_invariant_unchanged,

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of unchanged_memory_invariant_unchanged_write_list_nse.2.1.

unchanged_memory_invariant_unchanged_write_list_nse.2.2:

```

{-1} OK?(memory_read(pm')
      (raddr')(state(memory_write(pm')(waddr', cons1_var')(s'))))
{-2} states'(state(memory_write(pm')(waddr', cons1_var')(s'))))
{-3} OK?(memory_write(pm')(waddr', cons1_var')(s'))
{-4} OK?(memory_read(pm')(raddr')(s'))
{-5} OK?(memory_write_list_nse(pm')(waddr', cons(cons1_var', cons2_var'))(s'))
{-6} transformer_invariant?(states',
                              memory_write_transformers(pm',
                                                          address_block(waddr',
                                                                           1 + length(cons2_var')),
                              &
                              singleton(raddr'))
{-7} ̄ (s: State, waddr: Address):
      unchanged_memory_invariant?(pm', states',
                                   memory_write_transformers(pm',
                                                             address_block(waddr,
                                                                           length(cons2_var')),
                                   &
                                   singleton(raddr'))
      ^
      transformers_ok?(states',
                       memory_write_transformers(pm',
                                                  address_block(waddr, length(cons2_var')),
                       & states'(s)
      ̄
      data(memory_read(pm')
            (raddr')(state(memory_write_list_nse(pm')(waddr, cons2_var')(s))))
      = data(memory_read(pm')(raddr')(s))
{-8} unchanged_memory_invariant?(pm', states',
                                   memory_write_transformers(pm',
                                                             address_block(waddr',
                                                                           1 + length(cons2_var')),
                                   &
                                   singleton(raddr'))
{-9} transformers_ok?(states',
                     memory_write_transformers(pm',
                                               address_block(waddr',
                                                             1 + length(cons2_var'))))
{-10} transformers_ok?(states', singleton(expr_2_super(memory_read(pm')(raddr'))))
{-11} states'(s')
-----
{1}  unchanged_memory_invariant?(pm', states',
                                   memory_write_transformers(pm',
                                                             address_block(waddr' + 1,
                                                                           length(cons2_var')),
                                   &
                                   singleton(raddr'))
{2}  data(memory_read(pm')
          (raddr')
          (state(memory_write_list_nse(pm')(waddr' + 1, cons2_var')
              (state(memory_write(pm')
                (waddr', cons1_var')(s'))))))
      = data(memory_read(pm')(raddr')(s'))

```

Keeping (-8 1) and hiding *,

Using lemma unchanged_memory_invariant_mono,

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `unchanged_memory_invariant_unchanged_write_list_nse.2.2`.

`unchanged_memory_invariant_unchanged_write_list_nse.2.3`:

{-1}	OK?(memory_read(pm')
{-2}	(raddr')(state(memory_write(pm')(waddr', cons1_var')(s'))))
{-3}	states'(state(memory_write(pm')(waddr', cons1_var')(s')))
{-4}	OK?(memory_write(pm')(waddr', cons1_var')(s'))
{-5}	OK?(memory_read(pm')(raddr')(s'))
{-6}	OK?(memory_write_list_nse(pm')(waddr', cons(cons1_var', cons2_var'))(s'))
{-7}	transformer_invariant?(states',
	memory_write_transformers(pm',
	address_block(waddr',
	1 + length(cons2_var'))))
	∧ (s: State, waddr: Address):
	unchanged_memory_invariant?(pm', states',
	memory_write_transformers(pm',
	ad-
	dress_block(waddr,
	length(cons2_var'))),
	singleton(raddr'))
	∧
	transformers_ok?(states',
	memory_write_transformers(pm',
	address_block(waddr, length(cons2_var'))))
	∧ states'(s)
	⊃
	data(memory_read(pm')
	(raddr')(state(memory_write_list_nse(pm')(waddr, cons2_var')(s))))
	= data(memory_read(pm')(raddr')(s))
{-8}	unchanged_memory_invariant?(pm', states',
	memory_write_transformers(pm',
	address_block(waddr',
	1 + length(cons2_var'))))
	singleton(raddr'))
{-9}	transformers_ok?(states',
	memory_write_transformers(pm',
	address_block(waddr',
	1 + length(cons2_var'))))
{-10}	transformers_ok?(states', singleton(expr_2_super(memory_read(pm')(raddr'))))
{-11}	states'(s')
{1}	transformers_ok?(states',
	memory_write_transformers(pm',
	address_block(waddr' + 1,
	length(cons2_var'))))
{2}	data(memory_read(pm')
	(raddr')
	(state(memory_write_list_nse(pm')(waddr' + 1, cons2_var')
	(state(memory_write(pm')
	(waddr', cons1_var')(s'))))
	= data(memory_read(pm')(raddr')(s'))

Keeping (-9 1) and hiding *,

Using lemma `transformers_ok_mono_transformers`,

C Proof scripts

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `unchanged_memory_invariant_unchanged_write_list_nse.2.3`.

`unchanged_memory_invariant_unchanged_write_list_nse.3`:

```

{1}  ∀ (bl: list[Byte], s: State, waddr: Address):
      unchanged_memory_invariant?(pm', states',
                                   memory_write_transformers(pm', ad-
dress_block(waddr, length(bl))),
                                   singleton(raddr'))
      ∧
      transformers_ok?(states',
                       memory_write_transformers(pm', ad-
dress_block(waddr, length(bl))))
      ∧
      transformers_ok?(states', singleton(expr_2_super(memory_read(pm')(raddr')))) ∧
      states'(s)
      ⊃ OK?[State, Byte](memory_read(pm')(raddr')(s))
{2}  ∀ (s: State, waddr: Address):
      unchanged_memory_invariant?(pm', states',
                                   memory_write_transformers(pm', ad-
dress_block(waddr, length(bl'))),
                                   singleton(raddr'))
      ∧
      transformers_ok?(states',
                       memory_write_transformers(pm', ad-
dress_block(waddr, length(bl'))))
      ∧
      transformers_ok?(states', singleton(expr_2_super(memory_read(pm')(raddr')))) ∧
      states'(s)
      ⊃
      data(memory_read(pm')
            (raddr')(state(memory_write_list_nse(pm')(waddr, bl')(s))))
      = data(memory_read(pm')(raddr')(s))

```

Hiding formulas: 2,

Repeatedly Skolemizing and flattening,

Using lemma `unchanged_memory_invariant_unchanged_write_list_nse_TCC3`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `unchanged_memory_invariant_unchanged_write_list_nse.3`.

unchanged_memory_invariant_unchanged_write_list_nse.4:

{1}	$\forall (bl: \text{list}[\text{Byte}], s: \text{State}, waddr: \text{Address}):$ $\text{unchanged_memory_invariant?}(pm', \text{states}',$ $\text{memory_write_transformers}(pm', \text{ad-}$ $\text{dress_block}(waddr, \text{length}(bl))),$ $\text{singleton}(raddr'))$ \wedge $\text{transformers_ok?}(\text{states}',$ $\text{memory_write_transformers}(pm', \text{ad-}$ $\text{dress_block}(waddr, \text{length}(bl))))$ \wedge $\text{transformers_ok?}(\text{states}', \text{singleton}(\text{expr_2_super}(\text{memory_read}(pm')(raddr')))) \wedge$ $\text{states}'(s))$ \supset $\text{OK?}[\text{State}, \text{Byte}]$ $(\text{memory_read}(pm')$ $(raddr')$ $(\text{state}[\text{State}, \text{Unit}] (\text{memory_write_list_nse}[\text{State}](pm')(waddr, bl)(s))))$
{2}	$\forall (s: \text{State}, waddr: \text{Address}):$ $\text{unchanged_memory_invariant?}(pm', \text{states}',$ $\text{memory_write_transformers}(pm', \text{ad-}$ $\text{dress_block}(waddr, \text{length}(bl'))),$ $\text{singleton}(raddr'))$ \wedge $\text{transformers_ok?}(\text{states}',$ $\text{memory_write_transformers}(pm', \text{ad-}$ $\text{dress_block}(waddr, \text{length}(bl'))))$ \wedge $\text{transformers_ok?}(\text{states}', \text{singleton}(\text{expr_2_super}(\text{memory_read}(pm')(raddr')))) \wedge$ $\text{states}'(s))$ \supset $\text{data}(\text{memory_read}(pm')$ $(raddr'))(\text{state}(\text{memory_write_list_nse}(pm')(waddr, bl')(s))))$ $= \text{data}(\text{memory_read}(pm')(raddr'))(s)$

Hiding formulas: 2,

Repeatedly Skolemizing and flattening,

Using lemma `unchanged_memory_invariant_unchanged_write_list_nse_TCC2`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `unchanged_memory_invariant_unchanged_write_list_nse.4`.

unchanged_memory_invariant_unchanged_write_list_nse.5:

$$\begin{array}{l}
 \{1\} \quad \forall (bl: \text{list}[\text{Byte}], s: \text{State}, waddr: \text{Address}): \\
 \quad \text{unchanged_memory_invariant?}(pm', \text{states}', \\
 \quad \quad \quad \text{memory_write_transformers}(pm', \\
 \quad \quad \quad \quad \quad \text{ad-} \\
 \quad \text{dress_block}(waddr, \text{length}(bl))), \\
 \quad \quad \quad \quad \quad \text{singleton}(raddr')) \\
 \quad \wedge \\
 \quad \text{transformers_ok?}(\text{states}', \\
 \quad \quad \quad \text{memory_write_transformers}(pm', \text{ad-} \\
 \quad \text{dress_block}(waddr, \text{length}(bl)))) \\
 \quad \wedge \\
 \quad \text{transformers_ok?}(\text{states}', \text{singleton}(\text{expr_2_super}(\text{memory_read}(pm')(raddr')))) \wedge \\
 \quad \text{states}'(s) \\
 \quad \supset \\
 \quad \text{OK?}[\text{State}, \text{Unit}](\text{memory_write_list_nse}[\text{State}](pm')(waddr, bl)(s)) \vee \\
 \quad \text{Exception?}[\text{State}, \text{Unit}](\text{memory_write_list_nse}[\text{State}](pm')(waddr, bl)(s)) \\
 \{2\} \quad \forall (s: \text{State}, waddr: \text{Address}): \\
 \quad \text{unchanged_memory_invariant?}(pm', \text{states}', \\
 \quad \quad \quad \text{memory_write_transformers}(pm', \\
 \quad \quad \quad \quad \quad \text{ad-} \\
 \quad \text{dress_block}(waddr, \text{length}(bl'))), \\
 \quad \quad \quad \quad \quad \text{singleton}(raddr')) \\
 \quad \wedge \\
 \quad \text{transformers_ok?}(\text{states}', \\
 \quad \quad \quad \text{memory_write_transformers}(pm', \text{ad-} \\
 \quad \text{dress_block}(waddr, \text{length}(bl')))) \\
 \quad \wedge \\
 \quad \text{transformers_ok?}(\text{states}', \text{singleton}(\text{expr_2_super}(\text{memory_read}(pm')(raddr')))) \wedge \\
 \quad \text{states}'(s) \\
 \quad \supset \\
 \quad \text{data}(\text{memory_read}(pm') \\
 \quad \quad \quad (raddr')(state(\text{memory_write_list_nse}(pm')(waddr, bl')(s)))) \\
 \quad = \text{data}(\text{memory_read}(pm')(raddr')(s))
 \end{array}$$

Hiding formulas: 2,

Repeatedly Skolemizing and flattening,

Using lemma `unchanged_memory_invariant_unchanged_write_list_nse_TCC1`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `unchanged_memory_invariant_unchanged_write_list_nse.5`.

Q.E.D.

C.123.34 Memory_Change_2.unchanged_memory_invariant_unchanged_write_list_TCC1

Terse proof for `unchanged_memory_invariant_unchanged_write_list_TCC1`.

unchanged_memory_invariant_unchanged_write_list_TCC1:

{1}	$\forall (pm: \text{Memory_struct}[\text{State}], \text{states}: \text{PRED}[\text{State}], s: \text{State}, \text{raddr}, \text{waddr}: \text{Address}, \\ \text{bl}: \text{list}[\text{Byte}]): \\ \text{unchanged_memory_invariant?}(pm, \text{states}, \\ \quad (\text{memory_write_transformers}(pm, \text{address_block}(\text{waddr}, \text{length}(\text{bl})) \cup \text{singleton}(\text{raddr})) \\ \wedge \\ \text{transformers_ok?}(\text{states}, \\ \quad (\text{memory_write_transformers}(pm, \text{address_block}(\text{waddr}, \text{length}(\text{bl})) \cup \text{memory_write_transformers}(pm, \text{address_block}(\text{raddr}, \text{length}(\text{bl}))) \\ \wedge \\ \text{transformers_ok?}(\text{states}, \text{singleton}(\text{expr_2_super}(\text{memory_read}(pm)(\text{raddr})))) \wedge \\ \text{side_effect_content_unchanged}(\text{address_block}(\text{waddr}, \text{length}(\text{bl})), \text{states}, \\ \quad \text{memory_write_side_effect}(pm)) \\ \wedge \text{states}(s) \\ \supset \\ \text{OK?}[\text{State}, \text{Unit}](\text{memory_write_list}[\text{State}](pm)(\text{waddr}, \text{bl})(s)) \vee \\ \text{Exception?}[\text{State}, \text{Unit}](\text{memory_write_list}[\text{State}](pm)(\text{waddr}, \text{bl})(s))$
-----	---

Repeatedly Skolemizing and flattening,

Forward chaining on unchanged_memory_invariant_invariant,

Using lemma memory_write_list_ok,

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of unchanged_memory_invariant_unchanged_write_list_TCC1.

Q.E.D.

C.123.35 Memory_Change_2.unchanged_memory_invariant_unchanged_write_list_TCC2

Terse proof for unchanged_memory_invariant_unchanged_write_list_TCC2.

C Proof scripts

unchanged_memory_invariant_unchanged_write_list_TCC2:

$$\begin{array}{l}
 \{1\} \quad \forall (pm: \text{Memory_struct}[\text{State}], \text{states}: \text{PRED}[\text{State}], s: \text{State}, \text{raddr}, \text{waddr}: \text{Ad-} \\
 \quad \text{dress}, \\
 \quad \text{bl}: \text{list}[\text{Byte}]): \\
 \quad \text{unchanged_memory_invariant?}(pm, \text{states}, \\
 \quad \quad \quad (\text{memory_write_transformers}(pm, \text{address_block}(\text{waddr}, \text{length}(\text{bl}), \\
 \quad \quad \quad \text{singleton}(\text{raddr}))) \\
 \quad \wedge \\
 \quad \text{transformers_ok?}(\text{states}, \\
 \quad \quad \quad (\text{memory_write_transformers}(pm, \text{address_block}(\text{waddr}, \text{length}(\text{bl}))) \cup \text{me} \\
 \quad \wedge \\
 \quad \text{transformers_ok?}(\text{states}, \text{singleton}(\text{expr_2_super}(\text{memory_read}(pm)(\text{raddr})))) \wedge \\
 \quad \quad \quad \text{side_effect_content_unchanged}(\text{address_block}(\text{waddr}, \text{length}(\text{bl})), \text{states}, \\
 \quad \quad \quad \quad \quad \quad \text{memory_write_side_effect}(pm)) \\
 \quad \wedge \text{states}(s) \\
 \quad \supset \\
 \quad \text{OK?}[\text{State}, \text{Byte}] \\
 \quad \quad \quad (\text{memory_read}(pm) \\
 \quad \quad \quad \quad \quad \quad (\text{raddr}) \\
 \quad \quad \quad \quad \quad \quad (\text{state}[\text{State}, \text{Unit}](\text{memory_write_list}[\text{State}](pm)(\text{waddr}, \text{bl})(s))))
 \end{array}$$

Repeatedly Skolemizing and flattening,

Forward chaining on `unchanged_memory_invariant_invariant`,

Using lemma `memory_write_list_ok`,

Simplifying, rewriting, and recording with decision procedures,

Using lemma `expr_transformers_ok_ok`,

Installing automatic rewrites from: `singleton union_right has_next_state add_as_union mem-
memory_write_side_effect_super_transformers_memory_write_side_effect union_subset1 subset_equal sub-
set_singleton union_right`

Simplifying, rewriting, and recording with decision procedures,

Using lemma `transformer_invariant_write_list`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 2 subgoals:

unchanged_memory_invariant_unchanged_write_list_TCC2.1:

{-1}	side_effect_content_unchanged(address_block(waddr', length(bl')), states', memory_write_side_effect(pm'))
{-2}	states'(s')
{-3}	result_pred(states')(expr_2_super(memory_write_list(pm')(waddr', bl'))(s'))
{-4}	OK?(memory_write_list(pm')(waddr', bl')(s'))
{-5}	transformer_invariant?(states', (memory_write_transformers(pm', address_block(waddr', length(bl'))) ∪ memory_
{-6}	unchanged_memory_invariant?(pm', states', (memory_write_transformers(pm', address_block(waddr', length(bl'))) ∪ m singleton(raddr'))
{-7}	transformers_ok?(states', (memory_write_transformers(pm', address_block(waddr', length(bl'))) ∪ memory_write_
{-8}	transformers_ok?(states', singleton(expr_2_super(memory_read(pm')(raddr'))))
{1}	states'(state(memory_write_list[State](pm')(waddr', bl')(s')))
{2}	OK?[State, Byte] (memory_read(pm') (raddr') (state[State, Unit](memory_write_list[State](pm')(waddr', bl')(s'))))

Rewriting using expr_result_pred_next_state, matching in *,

This completes the proof of unchanged_memory_invariant_unchanged_write_list_TCC2.1.

unchanged_memory_invariant_unchanged_write_list_TCC2.2:

{-1}	side_effect_content_unchanged(address_block(waddr', length(bl')), states', memory_write_side_effect(pm'))
{-2}	states'(s')
{-3}	OK?(memory_write_list(pm')(waddr', bl')(s'))
{-4}	transformer_invariant?(states', (memory_write_transformers(pm', address_block(waddr', length(bl'))) ∪ memory_
{-5}	unchanged_memory_invariant?(pm', states', (memory_write_transformers(pm', address_block(waddr', length(bl'))) ∪ m singleton(raddr'))
{-6}	transformers_ok?(states', (memory_write_transformers(pm', address_block(waddr', length(bl'))) ∪ memory_write_
{-7}	transformers_ok?(states', singleton(expr_2_super(memory_read(pm')(raddr'))))
{1}	transformer_invariant?(states', (memory_write_transformers(pm', address_block(waddr', length(bl'))) ∪ singleton
{2}	states'(state(memory_write_list[State](pm')(waddr', bl')(s')))
{3}	OK?[State, Byte] (memory_read(pm') (raddr') (state[State, Unit](memory_write_list[State](pm')(waddr', bl')(s'))))

Keeping (-4 1) and hiding *,

Rewriting using transformer_invariant_union_transformers, matching in *,

we get 2 subgoals:

unchanged_memory_invariant_unchanged_write_list_TCC2.2.1:

{-1}	transformer_invariant?(states', (memory_write_transformers(pm', address_block(waddr', length(bl'))
{1}	transformer_invariant?(states', memory_write_transformers(pm', ad- dress_block(waddr', length(bl'))))
{2}	transformer_invariant?(states', memory_write_transformers(pm', address_block(waddr', length(bl'))

Hiding formulas: 2,

Using lemma transformer_invariant_mono_transformers,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of unchanged_memory_invariant_unchanged_write_list_TCC2.2.1.

unchanged_memory_invariant_unchanged_write_list_TCC2.2.2:

{-1}	transformer_invariant?(states', (memory_write_transformers(pm', address_block(waddr', length(bl'))
{1}	transformer_invariant?(states', singleton(expr_2_super(memory_write_side_effect(pm') (waddr', bl', FALSE))))
{2}	transformer_invariant?(states', memory_write_transformers(pm', address_block(waddr', length(bl'))

Hiding formulas: 2,

Using lemma transformer_invariant_mono_transformers,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of unchanged_memory_invariant_unchanged_write_list_TCC2.2.2.

Q.E.D.

C.123.36 Mem- ory_Change_2.unchanged_memory_invariant_unchanged_write_list_TCC3

Terse proof for unchanged_memory_invariant_unchanged_write_list_TCC3.

unchanged_memory_invariant_unchanged_write_list_TCC3:

{1}	\forall (pm: Memory_struct[State], states: PRED[State], s: State, raddr, waddr: Ad- dress, bl: list[Byte]): unchanged_memory_invariant?(pm, states, (memory_write_transformers(pm, address_block(waddr, length(bl)) singleton(raddr)) \wedge transformers_ok?(states, (memory_write_transformers(pm, address_block(waddr, length(bl))) \cup me \wedge transformers_ok?(states, singleton(expr_2_super(memory_read(pm)(raddr)))) \wedge side_effect_content_unchanged(address_block(waddr, length(bl)), states, memory_write_side_effect(pm)) \wedge states(s) \supset OK?[State, Byte](memory_read(pm)(raddr)(s))
-----	---

Repeatedly Skolemizing and flattening,

Installing automatic rewrites from: singleton

Using lemma `expr_transformers_ok_ok`[State, Byte],

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `unchanged_memory_invariant_unchanged_write_list_TCC3`.

Q.E.D.

C.123.37 Mem- ory_Change_2.unchanged_memory_invariant_unchanged_write_list

Terse proof for `unchanged_memory_invariant_unchanged_write_list`.

`unchanged_memory_invariant_unchanged_write_list`:

$\{1\} \quad \forall (pm: \text{Memory_struct}[\text{State}], \text{states}: \text{PRED}[\text{State}], s: \text{State}, \text{raddr}, \text{waddr}: \text{Ad-} \\ \text{dress}, \\ \text{bl}: \text{list}[\text{Byte}]): \\ \text{unchanged_memory_invariant?}(pm, \text{states}, \\ \text{memory_write_transformers}(pm, \text{address_block}(\text{waddr}, \text{length}(\text{bl}))) \cup \text{memory_write_} \\ \text{singleton}(\text{raddr})) \\ \wedge \\ \text{transformers_ok?}(\text{states}, \\ \text{memory_write_transformers}(pm, \text{address_block}(\text{waddr}, \text{length}(\text{bl}))) \cup \text{memory_write_} \\ \wedge \\ \text{transformers_ok?}(\text{states}, \text{singleton}(\text{expr_2_super}(\text{memory_read}(pm)(\text{raddr})))) \wedge \\ \text{side_effect_content_unchanged}(\text{address_block}(\text{waddr}, \text{length}(\text{bl})), \text{states}, \\ \text{memory_write_side_effect}(pm)) \\ \wedge \text{states}(s) \\ \supset \\ \text{data}(\text{memory_read}(pm)(\text{raddr})(\text{state}(\text{memory_write_list}(pm)(\text{waddr}, \text{bl})(s)))) = \\ \text{data}(\text{memory_read}(pm)(\text{raddr})(s))$

Repeatedly Skolemizing and flattening,

Hiding formulas: -1,

Expanding the definition of `memory_write_list`,

Using lemma `expr_transformers_ok_ok`,

Installing automatic rewrites from: `union_right memory_write_side_effect_super_transformers_memory_write_side_effect subset_equal singleton has_next_state union_subset1`

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Expanding the definition of `##`,

Using lemma `side_effect_content_unchanged_content`[State],

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Replacing using formula -4,

Hiding formulas: -4,

Using lemma `expr_unchanged_memory_invariant_next_ok`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Using lemma `unchanged_memory_invariant_unchanged_write_list_nse`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 3 subgoals:

C Proof scripts

unchanged_memory_invariant_unchanged_write_list.1:

{-1}	$\text{transformers_ok?}(\text{states}', \text{singleton}(\text{expr_2_super}(\text{memory_read}(\text{pm}')(\text{raddr}'))))$
{-2}	$\text{states}'(\text{state}(\text{memory_write_side_effect}(\text{pm}')(\text{waddr}', \text{bl}', \text{FALSE})(s')))$
{-3}	$\text{data}(\text{memory_read}(\text{pm}')(\text{raddr}')(\text{state}(\text{memory_write_list_nse}(\text{pm}')(\text{waddr}', \text{bl}')(\text{state}(\text{memory_write_side_effect}(\text{pm}')(\text{waddr}', \text{bl}', \text{FALSE})(s'))))))$
	$=$
	$\text{data}(\text{memory_read}(\text{pm}')(\text{raddr}')(\text{state}(\text{memory_write_side_effect}(\text{pm}')(\text{waddr}', \text{bl}', \text{FALSE})(s'))))$
{-4}	$\text{unchanged_memory_invariant?}(\text{pm}', \text{states}', (\text{memory_write_transformers}(\text{pm}', \text{address_block}(\text{waddr}', \text{length}(\text{bl}'), \text{length}(\text{bl}'), \text{singleton}(\text{raddr}'))))$
{-5}	$\text{states}'(s')$
{-6}	$\text{side_effect_content_unchanged}(\text{address_block}(\text{waddr}', \text{length}(\text{bl}'), \text{length}(\text{bl}'), \text{memory_write_side_effect}(\text{pm}')(\text{waddr}', \text{bl}', \text{FALSE})(s')))$
{-7}	$\text{OK?}(\text{memory_write_side_effect}(\text{pm}')(\text{waddr}', \text{bl}', \text{FALSE})(s'))$
{-8}	$\text{transformers_ok?}(\text{states}', (\text{memory_write_transformers}(\text{pm}', \text{address_block}(\text{waddr}', \text{length}(\text{bl}'), \text{length}(\text{bl}'), \text{singleton}(\text{raddr}')))) \cup \text{memory_write_side_effect}(\text{pm}')(\text{waddr}', \text{bl}', \text{FALSE})(s'))$
{1}	$\text{data}(\text{memory_read}(\text{pm}')(\text{raddr}')(\text{state}(\text{memory_write_list_nse}(\text{pm}')(\text{waddr}', \text{bl}')(\text{state}(\text{memory_write_side_effect}(\text{pm}')(\text{waddr}', \text{bl}', \text{FALSE})(s'))))))$
	$= \text{data}(\text{memory_read}(\text{pm}')(\text{raddr}')(\text{state}(\text{memory_write_list_nse}(\text{pm}')(\text{waddr}', \text{bl}')(\text{state}(\text{memory_write_side_effect}(\text{pm}')(\text{waddr}', \text{bl}', \text{FALSE})(s'))))))$

Hiding formulas: -2,

Using lemma `expr_unchanged_memory_invariant_unchanged`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 2 subgoals:

unchanged_memory_invariant_unchanged_write_list.1.1:

$$\begin{array}{l}
\{-1\} \text{ unchanged_memory_invariant?}(pm', \text{states}', \\
\qquad\qquad\qquad (\text{memory_write_transformers}(pm', \text{address_block}(\text{waddr}', \text{length}(bl')) \cup \text{m} \\
\qquad\qquad\qquad \text{singleton}(\text{raddr}')) \\
\{-2\} \text{ states}'(s') \\
\{-3\} \text{ OK?}(\text{memory_write_side_effect}(pm')(\text{waddr}', bl', \text{FALSE})(s')) \\
\{-4\} \text{ transformers_ok?}(\text{states}', \text{singleton}(\text{expr_2_super}(\text{memory_read}(pm')(\text{raddr}')))) \\
\{-5\} \text{ data}(\text{memory_read}(pm') \\
\qquad\qquad\qquad (\text{raddr}') \\
\qquad\qquad\qquad (\text{state}(\text{memory_write_list_nse}(pm')(\text{waddr}', bl') \\
\qquad\qquad\qquad\qquad\qquad\qquad (\text{state}(\text{memory_write_side_effect}(pm') \\
\qquad\qquad\qquad\qquad\qquad\qquad (\text{waddr}', bl', \text{FALSE})(s'))))) \\
= \\
\text{data}(\text{memory_read}(pm') \\
\qquad\qquad\qquad (\text{raddr}')(\text{state}(\text{memory_write_side_effect}(pm')(\text{waddr}', bl', \text{FALSE})(s')))) \\
\{-6\} \text{ side_effect_content_unchanged}(\text{address_block}(\text{waddr}', \text{length}(bl')), \text{states}', \\
\qquad\qquad\qquad \text{memory_write_side_effect}(pm')) \\
\{-7\} \text{ transformers_ok?}(\text{states}', \\
\qquad\qquad\qquad (\text{memory_write_transformers}(pm', \text{address_block}(\text{waddr}', \text{length}(bl')) \cup \text{memory_write-} \\
\hline
\{1\} \text{ OK?}(\text{memory_read}(pm') \\
\qquad\qquad\qquad (\text{raddr}')(\text{state}(\text{memory_write_side_effect}(pm')(\text{waddr}', bl', \text{FALSE})(s')))) \\
\{2\} \text{ data}(\text{memory_read}(pm') \\
\qquad\qquad\qquad (\text{raddr}') \\
\qquad\qquad\qquad (\text{state}(\text{memory_write_list_nse}(pm')(\text{waddr}', bl') \\
\qquad\qquad\qquad\qquad\qquad\qquad (\text{state}(\text{memory_write_side_effect}(pm') \\
\qquad\qquad\qquad\qquad\qquad\qquad (\text{waddr}', bl', \text{FALSE})(s'))))) \\
= \text{data}(\text{memory_read}(pm')(\text{raddr}')(s'))
\end{array}$$

Hiding formulas: -1, -5, 2,

Using lemma `expr_transformers_ok_ok`[State, Byte],

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `unchanged_memory_invariant_unchanged_write_list.1.1`.

unchanged_memory_invariant_unchanged_write_list.1.2:

{-1}	$\text{unchanged_memory_invariant?}(pm', \text{states}',$ $\quad (\text{memory_write_transformers}(pm', \text{address_block}(\text{waddr}', \text{length}(\text{bl}')),$ $\quad \text{singleton}(\text{raddr}'))$
{-2}	$\text{states}'(s')$
{-3}	$\text{OK?}(\text{memory_write_side_effect}(pm')(\text{waddr}', \text{bl}', \text{FALSE})(s'))$
{-4}	$\text{transformers_ok?}(\text{states}', \text{singleton}(\text{expr_2_super}(\text{memory_read}(pm')(\text{raddr}'))))$
{-5}	$\text{data}(\text{memory_read}(pm')$ $\quad (\text{raddr}')$ $\quad (\text{state}(\text{memory_write_list_nse}(pm')(\text{waddr}', \text{bl}')$ $\quad \quad (\text{state}(\text{memory_write_side_effect}(pm')$ $\quad \quad \quad (\text{waddr}', \text{bl}', \text{FALSE})(s'))))))$
	$=$
	$\text{data}(\text{memory_read}(pm')$ $\quad (\text{raddr}')(\text{state}(\text{memory_write_side_effect}(pm')(\text{waddr}', \text{bl}', \text{FALSE})(s'))))$
{-6}	$\text{side_effect_content_unchanged}(\text{address_block}(\text{waddr}', \text{length}(\text{bl}')), \text{states}',$ $\quad \text{memory_write_side_effect}(pm'))$
{-7}	$\text{transformers_ok?}(\text{states}',$ $\quad (\text{memory_write_transformers}(pm', \text{address_block}(\text{waddr}', \text{length}(\text{bl}')))) \cup \text{mem}$
{1}	$\text{OK?}(\text{memory_read}(pm')(\text{raddr}'))(s')$
{2}	$\text{data}(\text{memory_read}(pm')$ $\quad (\text{raddr}')$ $\quad (\text{state}(\text{memory_write_list_nse}(pm')(\text{waddr}', \text{bl}')$ $\quad \quad (\text{state}(\text{memory_write_side_effect}(pm')$ $\quad \quad \quad (\text{waddr}', \text{bl}', \text{FALSE})(s'))))))$
	$= \text{data}(\text{memory_read}(pm')(\text{raddr}'))(s')$

Hiding formulas: -1, -5, 2,

Using lemma `expr_transformers_ok_ok`[State, Byte],

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `unchanged_memory_invariant_unchanged_write_list.1.2`.

unchanged_memory_invariant_unchanged_write_list.2:

{-1}	$\text{transformers_ok?}(\text{states}', \text{singleton}(\text{expr_2_super}(\text{memory_read}(pm')(\text{raddr}'))))$
{-2}	$\text{states}'(\text{state}(\text{memory_write_side_effect}(pm')(\text{waddr}', \text{bl}', \text{FALSE})(s')))$
{-3}	$\text{unchanged_memory_invariant?}(pm', \text{states}',$ $\quad (\text{memory_write_transformers}(pm', \text{address_block}(\text{waddr}', \text{length}(\text{bl}')),$ $\quad \text{singleton}(\text{raddr}'))$
{-4}	$\text{states}'(s')$
{-5}	$\text{side_effect_content_unchanged}(\text{address_block}(\text{waddr}', \text{length}(\text{bl}')), \text{states}',$ $\quad \text{memory_write_side_effect}(pm'))$
{-6}	$\text{OK?}(\text{memory_write_side_effect}(pm')(\text{waddr}', \text{bl}', \text{FALSE})(s'))$
{-7}	$\text{transformers_ok?}(\text{states}',$ $\quad (\text{memory_write_transformers}(pm', \text{address_block}(\text{waddr}', \text{length}(\text{bl}')))) \cup \text{mem}$
{1}	$\text{transformers_ok?}(\text{states}',$ $\quad \text{memory_write_transformers}(pm', \text{address_block}(\text{waddr}', \text{length}(\text{bl}'))))$
{2}	$\text{data}(\text{memory_read}(pm')$ $\quad (\text{raddr}')$ $\quad (\text{state}(\text{memory_write_list_nse}(pm')(\text{waddr}', \text{bl}')$ $\quad \quad (\text{state}(\text{memory_write_side_effect}(pm')$ $\quad \quad \quad (\text{waddr}', \text{bl}', \text{FALSE})(s'))))))$
	$= \text{data}(\text{memory_read}(pm')(\text{raddr}'))(s')$

Keeping (-7 1) and hiding *,

Using lemma transformers_ok_mono_transformers,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of unchanged_memory_invariant_unchanged_write_list.2.

unchanged_memory_invariant_unchanged_write_list.3:

$$\begin{array}{l}
 \{-1\} \text{ transformers_ok?}(\text{states}', \text{singleton}(\text{expr_2_super}(\text{memory_read}(\text{pm}')(\text{raddr}')))) \\
 \{-2\} \text{ states}'(\text{state}(\text{memory_write_side_effect}(\text{pm}')(\text{waddr}', \text{bl}', \text{FALSE})(\text{s}')))) \\
 \{-3\} \text{ unchanged_memory_invariant?}(\text{pm}', \text{states}', \\
 \qquad \qquad \qquad (\text{memory_write_transformers}(\text{pm}', \text{address_block}(\text{waddr}', \text{length}(\text{bl}')) \cup \text{m} \\
 \qquad \qquad \qquad \text{singleton}(\text{raddr}')) \\
 \\
 \{-4\} \text{ states}'(\text{s}') \\
 \{-5\} \text{ side_effect_content_unchanged}(\text{address_block}(\text{waddr}', \text{length}(\text{bl}')), \text{states}', \\
 \qquad \qquad \qquad \text{memory_write_side_effect}(\text{pm}')) \\
 \{-6\} \text{ OK?}(\text{memory_write_side_effect}(\text{pm}')(\text{waddr}', \text{bl}', \text{FALSE})(\text{s}')) \\
 \{-7\} \text{ transformers_ok?}(\text{states}', \\
 \qquad \qquad \qquad (\text{memory_write_transformers}(\text{pm}', \text{address_block}(\text{waddr}', \text{length}(\text{bl}')))) \cup \text{memory_write_re} \\
 \hline
 \{1\} \text{ unchanged_memory_invariant?}(\text{pm}', \text{states}', \\
 \qquad \qquad \qquad \text{memory_write_transformers}(\text{pm}', \\
 \qquad \qquad \qquad \text{ad-} \\
 \qquad \qquad \qquad \text{dress_block}(\text{waddr}', \text{length}(\text{bl}'))), \\
 \qquad \qquad \qquad \text{singleton}(\text{raddr}')) \\
 \\
 \{2\} \text{ data}(\text{memory_read}(\text{pm}') \\
 \qquad \qquad \qquad (\text{raddr}') \\
 \qquad \qquad \qquad (\text{state}(\text{memory_write_list_nse}(\text{pm}')(\text{waddr}', \text{bl}') \\
 \qquad \qquad \qquad \qquad \qquad \qquad \text{state}(\text{memory_write_side_effect}(\text{pm}') \\
 \qquad \qquad \qquad \qquad \qquad \qquad \text{(waddr}', \text{bl}', \text{FALSE})(\text{s}'))))))) \\
 \\
 = \text{data}(\text{memory_read}(\text{pm}')(\text{raddr}')(\text{s}'))
 \end{array}$$

Keeping (-3 1) and hiding *,

Using lemma unchanged_memory_invariant_mono,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of unchanged_memory_invariant_unchanged_write_list.3.

Q.E.D.

C.123.38 Memory_Change_2.unchanged_memory_invariant_read_list

Terse proof for unchanged_memory_invariant_read_list.

unchanged_memory_invariant_read_list:

$$\begin{array}{l}
 \{1\} \forall (\text{addresses: PRED}[\text{Address}], \text{pm: Memory_struct}[\text{State}], \text{states: PRED}[\text{State}], \text{addr: Ad-} \\
 \text{dress,} \\
 \qquad \text{size: nat}): \\
 \text{unchanged_memory_invariant?}(\text{pm}, \text{states}, \\
 \qquad \qquad \qquad (\text{memory_read_transformers}(\text{pm}, \text{address_block}(\text{addr}, \text{size})) \cup \text{memory_re} \\
 \qquad \qquad \qquad \text{addresses}) \\
 \\
 \wedge \\
 \text{transformers_ok?}(\text{states}, \\
 \qquad \qquad \qquad (\text{memory_read_transformers}(\text{pm}, \text{addresses}) \cup \text{memory_read_side_effect_super_transfor} \\
 \wedge (\text{address_block}(\text{addr}, \text{size}) \subseteq \text{addresses}) \\
 \supset \\
 \text{unchanged_memory_invariant?}(\text{pm}, \text{states}, \\
 \qquad \qquad \qquad \text{singleton}(\text{expr_2_super}(\text{memory_read_list}(\text{pm})(\text{addr}, \text{size}))), \\
 \qquad \qquad \qquad \text{addresses})
 \end{array}$$

C Proof scripts

Repeatedly Skolemizing and flattening,

Installing automatic rewrites from: result_pred state_expr_2_super ok_expr_2_super has_next_state memory_read_transformers_mono subset_bigger_union_left memory_read_side_effect_super_transformers_mono subset_equal union_subset1 union_subset3 union_left union_right subset_singleton subset_bigger_union_right union_left union_right transformers_ok_union_transformers memory_read_transformers_memory_read memory_read_side_effect_super_transformers_memory_read_side_effect

Using lemma transformers_ok_mono_transformers,

Using lemma transformers_ok_mono_transformers,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Expanding the definition of unchanged_memory_invariant?,

Applying propositional simplification,

we get 2 subgoals:

unchanged_memory_invariant_read_list.1:

{-1}	transformers_ok?(states',
{-2}	transformers_ok?(states',
{-3}	transformers_ok?(states', memory_read_transformers(pm', address_block(addr', size')))
{-4}	size' ≥ 0
{-5}	unchanged_memory_invariant?(pm', states',
{-6}	(address_block(addr', size') ⊆ addresses')
{1}	transformer_invariant?(states',
	singleton(expr_2_super(memory_read_list(pm')(addr', size'))))

Expanding the definition of transformer_invariant?,

Repeatedly Skolemizing and flattening,

Expanding the definition of singleton,

Replacing using formula -2,

Hiding formulas: -2,

Forward chaining on unchanged_memory_invariant_invariant,

Using lemma memory_read_list_ok,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Using lemma memory_read_list_next_ok,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of unchanged_memory_invariant_read_list.1.

unchanged_memory_invariant_read_list.2:

{-1}	transformers_ok?(states',	
	(memory_read_transformers(pm', addresses') \cup memory_read_side_effect_super_transformers(pm',	
{-2}	transformers_ok?(states',	
	memory_read_side_effect_super_transformers(pm',	
	address_block(addr', size'))	
{-3}	transformers_ok?(states', memory_read_transformers(pm', address_block(addr', size')))	
{-4}	size' \geq 0	
{-5}	unchanged_memory_invariant?(pm', states',	
	(memory_read_transformers(pm', address_block(addr', size')) \cup memory_read_side_effect_super_transformers(pm',	
	address_block(addr', size')) \subseteq addresses')	
{-6}		
{1}	\forall (s: State, q: [State \rightarrow SuperResult[State]], a: Address):	
	states'(s) \wedge	
	singleton(expr_2_super(memory_read_list(pm')(addr', size')))(q) \wedge	
	addresses'(a) \wedge	
	OK?(q(s)) \wedge	
	OK?(memory_read(pm')(a)(s)) \wedge	
	OK?(memory_read(pm')(a)(state(q(s))))	
	\supset	
	data(memory_read(pm')(a)(state(q(s)))) = data(memory_read(pm')(a)(s))	

Using lemma unchanged_memory_invariant_mono,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Repeatedly Skolemizing and flattening,

Expanding the definition of singleton,

Replacing using formula -4,

Hiding formulas: -4,

Simplifying, rewriting, and recording with decision procedures,

Expanding the definition of memory_read_list,

Expanding the definition of ##,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Using lemma memory_read_list_nse_next_length,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 2 subgoals:

unchanged_memory_invariant_read_list.2.2:

{-1}	transformers_ok?(states', memory_read_transformers(pm', address_block(addr', size')))
{-2}	states'(s')
{-3}	unchanged_memory_invariant?(pm', states', (memory_read_transformers(pm', address_block(addr', size')) ∪ memory_read_transformers(pm', addresses'))
{-4}	unchanged_memory_invariant?(pm', states', memory_read_transformers(pm', address_block(addr', size')), addresses')
{-5}	addresses'(a')
{-6}	OK?(memory_read_list_nse(pm')(addr', size')(s'))
{-7}	OK?(memory_read_side_effect(pm') (addr', data(memory_read_list_nse(pm')(addr', size')(s')), FALSE) (state(memory_read_list_nse(pm')(addr', size')(s'))))
{-8}	OK?(memory_read(pm')(a')(s'))
{-9}	OK?(memory_read(pm') (a') (state(memory_read_side_effect(pm') (addr', data(memory_read_list_nse(pm')(addr', size')(s')), FALSE) (state(memory_read_list_nse(pm')(addr', size')(s'))))))
{-10}	transformers_ok?(states', (memory_read_transformers(pm', addresses') ∪ memory_read_side_effect_super_transformers(pm', addresses'))
{-11}	transformers_ok?(states', memory_read_side_effect_super_transformers(pm', address_block(addr', size'))
{-12}	size' ≥ 0
{-13}	(address_block(addr', size') ⊆ addresses')
{1}	transformer_invariant?(states', memory_read_transformers(pm', address_block(addr', size'))
{2}	data(memory_read(pm') (a') (state(memory_read_side_effect(pm') (addr', data(memory_read_list_nse(pm')(addr', size')(s')), FALSE) (state(memory_read_list_nse(pm')(addr', size')(s')))))) = data(memory_read(pm')(a')(s'))

Keeping (-4 1) and hiding *,

Forward chaining on unchanged_memory_invariant_invariant,

This completes the proof of unchanged_memory_invariant_read_list.2.2.

Q.E.D.

C.123.39 Mem- ory_Change_2.unchanged_memory_read_list_write_list_nse_TCC1

Terse proof for unchanged_memory_read_list_write_list_nse_TCC1.

unchanged_memory_read_list_write_list_nse_TCC1:

<pre> {1} ∃ (pm: Memory_struct[State], states: PRED[State], waddr: Ad- dress, bl: list[Byte], s: State): unchanged_memory_invariant?(pm, states, memory_read_transformers(pm, ad- dress_block(waddr, length(bl))), address_block(waddr, length(bl))) ∧ unchanged_memory_write_invariant?(pm, states, address_block(waddr, length(bl))) ∧ changed_memory_invariant?(pm, states, address_block(waddr, length(bl))) ∧ transformers_ok?(states, memory_read_transformers(pm, ad- dress_block(waddr, length(bl)))) ∧ transformers_ok?(states, memory_write_transformers(pm, ad- dress_block(waddr, length(bl)))) ∧ states(s) ⊃ OK?[State, Unit](memory_write_list_nse[State](pm)(waddr, bl)(s)) ∨ Exception?[State, Unit](memory_write_list_nse[State](pm)(waddr, bl)(s)) </pre>

Repeatedly Skolemizing and flattening,

Using lemma `memory_write_list_ok_nse`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Rewriting using `unchanged_memory_write_invariant_transformer_invariant`, matching in `*`,

This completes the proof of `unchanged_memory_read_list_write_list_nse_TCC1`.

Q.E.D.

C.123.40 Memory_Change_2.unchanged_memory_read_list_write_list_nse_TCC2

Terse proof for `unchanged_memory_read_list_write_list_nse_TCC2`.

unchanged_memory_read_list_write_list_nse_TCC2:

<pre> {1} ∀ (pm: Memory_struct[State], states: PRED[State], waddr: Ad- dress, bl: list[Byte], s: State): unchanged_memory_invariant?(pm, states, memory_read_transformers(pm, ad- dress_block(waddr, length(bl))), address_block(waddr, length(bl))) ∧ unchanged_memory_write_invariant?(pm, states, address_block(waddr, length(bl))) ∧ changed_memory_invariant?(pm, states, address_block(waddr, length(bl))) ∧ transformers_ok?(states, memory_read_transformers(pm, ad- dress_block(waddr, length(bl)))) ∧ transformers_ok?(states, memory_write_transformers(pm, ad- dress_block(waddr, length(bl)))) ∧ states(s) ⊃ OK?[State, list[Byte]] (memory_read_list_nse[State] (pm)(waddr, length[Byte](bl)) (state[State, Unit](memory_write_list_nse[State](pm)(waddr, bl)(s)))) </pre>

Repeatedly Skolemizing and flattening,

Using lemma memory_read_list_nse_ok,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 2 subgoals:

unchanged_memory_read_list_write_list_nse_TCC2.1:

<pre> {-1} transformers_ok?(states', memory_read_transformers(pm', address_block(waddr', length(bl')))) {-2} unchanged_memory_invariant?(pm', states', memory_read_transformers(pm', ad- dress_block(waddr', length(bl'))), address_block(waddr', length(bl'))) {-3} unchanged_memory_write_invariant?(pm', states', address_block(waddr', length(bl'))) {-4} changed_memory_invariant?(pm', states', address_block(waddr', length(bl'))) {-5} transformers_ok?(states', memory_write_transformers(pm', address_block(waddr', length(bl')))) {-6} states'(s') </pre>
<pre> {1} states'(state[State, Unit](memory_write_list_nse[State](pm')(waddr', bl')(s'))) {2} OK?[State, list[Byte]] (memory_read_list_nse[State] (pm')(waddr', length[Byte](bl')) (state[State, Unit] (memory_write_list_nse[State](pm')(waddr', bl')(s')))) </pre>

Hiding formulas: 2,

C Proof scripts

Using lemma `unchanged_memory_write_invariant_transformer_invariant`,
Simplifying, rewriting, and recording with decision procedures,
Using lemma `transformer_invariant_write_list_nse`,
Simplifying, rewriting, and recording with decision procedures,
Using lemma `memory_write_list_ok_nse`,
Simplifying, rewriting, and recording with decision procedures,
Installing automatic rewrites from: `has_next_state`
Rewriting using `expr_result_pred_next_state`, matching in `*`,
This completes the proof of `unchanged_memory_read_list_write_list_nse_TCC2.1`.
`unchanged_memory_read_list_write_list_nse_TCC2.2`:

{-1}	transformers_ok?(states',
{-2}	memory_read_transformers(pm', address_block(waddr', length(bl'))))
{-3}	unchanged_memory_invariant?(pm', states',
{-4}	memory_read_transformers(pm',
{-5}	address_block(waddr', length(bl'))),
{-6}	address_block(waddr', length(bl')))
{-3}	unchanged_memory_write_invariant?(pm', states', address_block(waddr', length(bl')))
{-4}	changed_memory_invariant?(pm', states', address_block(waddr', length(bl')))
{-5}	transformers_ok?(states',
{-6}	memory_write_transformers(pm', address_block(waddr', length(bl'))))
{-6}	states'(s')
{1}	transformer_invariant?(states',
	memory_read_transformers(pm',
	address_block(waddr',
	length[Byte](bl'))))
{2}	OK?[State, list[Byte]]
	(memory_read_list_nse[State]
	(pm')(waddr', length[Byte](bl'))
	(state[State, Unit]
	(memory_write_list_nse[State](pm')(waddr', bl')(s'))))

Forward chaining on `unchanged_memory_invariant_invariant`,
This completes the proof of `unchanged_memory_read_list_write_list_nse_TCC2.2`.
Q.E.D.

C.123.41 Memory_Change_2.unchanged_memory_read_list_write_list_nse

Terse proof for `unchanged_memory_read_list_write_list_nse`.

unchanged_memory_read_list_write_list_nse:

```

{1}  ∀ (pm: Memory_struct[State], states: PRED[State], waddr: Ad-
      dress, bl: list[Byte],
      s: State):
      unchanged_memory_invariant?(pm, states,
                                  memory_read_transformers(pm,
                                                            ad-
dress_block(waddr, length(bl))),
                                  address_block(waddr, length(bl)))
      ∧
      unchanged_memory_write_invariant?(pm, states, address_block(waddr, length(bl))) ∧
      changed_memory_invariant?(pm, states, address_block(waddr, length(bl))) ∧
      transformers_ok?(states,
                       memory_read_transformers(pm, ad-
dress_block(waddr, length(bl))))
      ∧
      transformers_ok?(states,
                       memory_write_transformers(pm, ad-
dress_block(waddr, length(bl))))
      ∧ states(s)
      ⊃
      data(memory_read_list_nse(pm)(waddr, length(bl))
           (state(memory_write_list_nse(pm)(waddr, bl)(s))))
      = bl

```

For the top quantifier in 1, we introduce Skolem constants: (pm' states' — — —),

Inducting on bl on formula 1,

we get 4 subgoals:

unchanged_memory_read_list_write_list_nse.1:

```

{1}  ∀ (waddr: Address, s: State):
      unchanged_memory_invariant?(pm', states',
                                  memory_read_transformers(pm',
                                                            ad-
dress_block(waddr, length(null))),
                                  address_block(waddr, length(null)))
      ∧
      unchanged_memory_write_invariant?(pm', states', address_block(waddr, length(null))) ∧
      changed_memory_invariant?(pm', states', address_block(waddr, length(null))) ∧
      transformers_ok?(states',
                       memory_read_transformers(pm', ad-
dress_block(waddr, length(null))))
      ∧
      transformers_ok?(states',
                       memory_write_transformers(pm', ad-
dress_block(waddr, length(null))))
      ∧ states'(s)
      ⊃
      data(memory_read_list_nse(pm')(waddr, length(null))
           (state(memory_write_list_nse(pm')(waddr, null)(s))))
      = null

```


C Proof scripts

Repeatedly Skolemizing and flattening,

Keeping 1 and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `unchanged_memory_read_list_write_list_nse.1`.

C Proof scripts

Repeatedly Skolemizing and flattening,

```
Installing automatic rewrites from: address_block address_block_subset_1 ##! length subset_equal union_subset1 union_subset3 union_left union_right subset_singleton subset_bigger_union_right subset_bigger_union_left union_left union_right memory_read_transformers_memory_read memory_read_transformers_mono memory_write_transformers_memory_write memory_write_transformers_mono transformer_invariant_mono_transformers expr_transformer_invariant_next_ok ok_result has_next_state state_expr_2_super ok_expr_2_super
```

Using lemma `unchanged_memory_write_invariant_transformer_invariant`,

Simplifying, rewriting, and recording with decision procedures,

Using lemma `memory_write_list_ok_nse`,

Simplifying, rewriting, and recording with decision procedures,

Using lemma `unchanged_memory_read_list_write_list_nse_TCC2`,

Simplifying, rewriting, and recording with decision procedures,

Expanding the definition of `memory_write_list_nse`,

Expanding the definition of `memory_read_list_nse`,

Expanding the definition of `memory_write_list_nse`,

Expanding the definition of `memory_write_list_nse`,

Expanding the definition of `memory_read_list_nse`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Applying extensionality,

we get 2 subgoals:

unchanged_memory_read_list_write_list_nse.2.1:

```

{-1} OK?(memory_write(pm')(waddr', cons1_var')(s'))
{-2} OK?(memory_read(pm')
      (waddr')
      (state[State, Unit]
        (memory_write_list_nse(pm')(waddr' + 1, cons2_var')
          (state(memory_write(pm')
            (waddr', cons1_var')(s'))))))))
{-3} OK?(memory_read_list_nse(pm')(waddr' + 1, length(cons2_var'))
      (state(memory_read(pm')
        (waddr')
        (state[State, Unit]
          (memory_write_list_nse(pm')
            (waddr' + 1,
              cons2_var')
            (state
              (memory_write(pm')
                (waddr', cons1_var')
                (s')))))))))
{-4} OK?[State, list[Byte]]
      (OK(state(memory_read_list_nse(pm')(waddr' + 1, length(cons2_var'))
        (state(memory_read(pm')
          (waddr')
          (state[State, Unit]
            (memory_write_list_nse(pm')
              (waddr' + 1,
                cons2_var')
              (state
                (memory_write
                  (pm')
                  (waddr',
                    cons1_var')
                    (s')))))))))
        cons(data(memory_read(pm')
          (waddr')
          (state[State, Unit]
            (memory_write_list_nse(pm')(waddr' + 1, cons2_var')
              (state(memory_write(pm')
                (waddr', cons1_var')
                (s'))))))),
          data(memory_read_list_nse(pm')(waddr' + 1, length(cons2_var'))
            (state(memory_read(pm')
              (waddr')
              (state[State, Unit]
                (memory_write_list_nse
                  (pm')
                  (waddr' + 1, cons2_var')
                  (state
                    (memory_write(pm')
                      (waddr', cons1_var')
                      (s'))))))))))))
{-5} OK?(memory_write_list_nse(pm')(waddr' + 1, cons2_var')
      (state(memory_write(pm')(waddr', cons1_var')(s'))))
{-6} transformer_invariant?(states',
      memory_write_transformers(pm',
        address_block(waddr',
          1 2507 length(cons2_var'))))
{-7} ∀ (waddr: Address, s: State):
      unchanged_memory_invariant?(pm', states',
        memory_read_transformers(pm',
          address_block(waddr,
            length(cons2_var'))),
        address_block(waddr, length(cons2_var'))))

```

C Proof scripts

Hiding formulas: -2, -3, -4, -7,

Using lemma `expr_transformer_invariant_next_ok`,

Simplifying, rewriting, and recording with decision procedures,

Using lemma `expr_transformers_ok_ok`,

Simplifying, rewriting, and recording with decision procedures,

Using lemma `unchanged_memory_invariant_unchanged_write_list_nse`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 4 subgoals:

`unchanged_memory_read_list_write_list_nse.2.1.1:`

<div style="display: flex; flex-direction: column; gap: 5px;"> <div style="display: flex; justify-content: space-between;"> {-1} states'(state(memory_write(pm')(waddr', cons1_var')(s'))) </div> <div style="display: flex; justify-content: space-between;"> {-2} data(memory_read(pm')(waddr')(state(memory_write_list_nse(pm')(waddr' + 1, cons2_var')(state(memory_write(pm')(waddr', cons1_var')(s')))))) </div> <div style="text-align: center; margin: 5px 0;">=</div> <div style="display: flex; justify-content: space-between;"> data(memory_read(pm')(waddr')(state(memory_write(pm')(waddr', cons1_var')(s')))) </div> <div style="display: flex; justify-content: space-between;"> {-3} OK?(memory_read(pm')(waddr')(state(memory_write(pm')(waddr', cons1_var')(s')))) </div> <div style="display: flex; justify-content: space-between;"> {-4} OK?(memory_write(pm')(waddr', cons1_var')(s')) </div> <div style="display: flex; justify-content: space-between;"> {-5} OK?(memory_write_list_nse(pm')(waddr' + 1, cons2_var')(state(memory_write(pm')(waddr', cons1_var')(s')))) </div> <div style="display: flex; justify-content: space-between;"> {-6} transformer_invariant?(states', memory_write_transformers(pm', address_block(waddr', 1 + length(cons2_var')))) </div> <div style="display: flex; justify-content: space-between;"> {-7} unchanged_memory_invariant?(pm', states', memory_read_transformers(pm', address_block(waddr', 1 + length(cons2_var')))) </div> <div style="display: flex; justify-content: space-between;"> {-8} unchanged_memory_write_invariant?(pm', states', address_block(waddr', 1 + length(cons2_var'))) </div> <div style="display: flex; justify-content: space-between;"> {-9} changed_memory_invariant?(pm', states', address_block(waddr', 1 + length(cons2_var'))) </div> <div style="display: flex; justify-content: space-between;"> {-10} transformers_ok?(states', memory_read_transformers(pm', address_block(waddr', 1 + length(cons2_var')))) </div> <div style="display: flex; justify-content: space-between;"> {-11} transformers_ok?(states', memory_write_transformers(pm', address_block(waddr', 1 + length(cons2_var')))) </div> <div style="display: flex; justify-content: space-between;"> {-12} states'(s') </div> </div> <hr style="border: 0.5px solid black; margin: 5px 0;"/> <div style="display: flex; justify-content: space-between;"> {1} data(memory_read(pm')(waddr')(state[State, Unit](memory_write_list_nse(pm')(waddr' + 1, cons2_var')(state(memory_write(pm')(waddr', cons1_var')(s')))))) </div> <div style="text-align: center; margin: 5px 0;">= cons1_var'</div>	
---	--

Replacing using formula -2,

Keeping (-9 1) and hiding *,

Expanding the definition of changed_memory_invariant?,

Applying disjunctive simplification to flatten sequent,

Rewriting using -2, matching in *,

This completes the proof of unchanged_memory_read_list_write_list_nse.2.1.1.

unchanged_memory_read_list_write_list_nse.2.1.2:

{-1}	states'(state(memory_write(pm')(waddr', cons1_var')(s')))
{-2}	OK?(memory_read(pm') (waddr')(state(memory_write(pm')(waddr', cons1_var')(s'))))
{-3}	OK?(memory_write(pm')(waddr', cons1_var')(s'))
{-4}	OK?(memory_write_list_nse(pm')(waddr' + 1, cons2_var') (state(memory_write(pm')(waddr', cons1_var')(s'))))
{-5}	transformer_invariant?(states', memory_write_transformers(pm', address_block(waddr', 1 + length(cons2_var'))))
{-6}	unchanged_memory_invariant?(pm', states', memory_read_transformers(pm', address_block(waddr', 1 + length(cons2_var'))), address_block(waddr', 1 + length(cons2_var')))
{-7}	unchanged_memory_write_invariant?(pm', states', address_block(waddr', 1 + length(cons2_var')))
{-8}	changed_memory_invariant?(pm', states', address_block(waddr', 1 + length(cons2_var')))
{-9}	transformers_ok?(states', memory_read_transformers(pm', address_block(waddr', 1 + length(cons2_var'))))
{-10}	transformers_ok?(states', memory_write_transformers(pm', address_block(waddr', 1 + length(cons2_var'))))
{-11}	states'(s')
{1}	transformers_ok?(states', singleton(expr_2_super(memory_read(pm')(waddr'))))
{2}	data(memory_read(pm') (waddr') (state[State, Unit] (memory_write_list_nse(pm')(waddr' + 1, cons2_var') (state(memory_write(pm') (waddr', cons1_var')(s'))))))
	= cons1_var'

Keeping (-9 1) and hiding *,

Using lemma transformers_ok_mono_transformers,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of unchanged_memory_read_list_write_list_nse.2.1.2.

unchanged_memory_read_list_write_list_nse.2.1.3:

```

{-1} states'(state(memory_write(pm')(waddr', cons1_var')(s')))
{-2} OK?(memory_read(pm')
      (waddr')(state(memory_write(pm')(waddr', cons1_var')(s'))))
{-3} OK?(memory_write(pm')(waddr', cons1_var')(s'))
{-4} OK?(memory_write_list_nse(pm')(waddr' + 1, cons2_var')
      (state(memory_write(pm')(waddr', cons1_var')(s'))))
{-5} transformer_invariant?(states',
      memory_write_transformers(pm',
      address_block(waddr',
      1 + length(cons2_var')))
{-6} unchanged_memory_invariant?(pm', states',
      memory_read_transformers(pm',
      address_block(waddr',
      1 + length(cons2_var')))
      address_block(waddr', 1 + length(cons2_var')))
{-7} unchanged_memory_write_invariant?(pm', states',
      address_block(waddr', 1 + length(cons2_var')))
{-8} changed_memory_invariant?(pm', states',
      address_block(waddr', 1 + length(cons2_var')))
{-9} transformers_ok?(states',
      memory_read_transformers(pm',
      address_block(waddr',
      1 + length(cons2_var')))
{-10} transformers_ok?(states',
      memory_write_transformers(pm',
      address_block(waddr',
      1 + length(cons2_var')))
{-11} states'(s')
-----
{1} transformers_ok?(states',
      memory_write_transformers(pm',
      address_block(waddr' + 1,
      length(cons2_var')))
{2} data(memory_read(pm')
      (waddr')
      (state[State, Unit]
      (memory_write_list_nse(pm')(waddr' + 1, cons2_var')
      (state(memory_write(pm')
      (waddr', cons1_var')(s'))))))
      = cons1_var'

```

Keeping (-10 1) and hiding *,

Using lemma transformers_ok_mono_transformers,

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of unchanged_memory_read_list_write_list_nse.2.1.3.

unchanged_memory_read_list_write_list_nse.2.1.4:

{-1}	states'(state(memory_write(pm')(waddr', cons1_var')(s')))
{-2}	OK?(memory_read(pm') (waddr')(state(memory_write(pm')(waddr', cons1_var')(s'))))
{-3}	OK?(memory_write(pm')(waddr', cons1_var')(s'))
{-4}	OK?(memory_write_list_nse(pm')(waddr' + 1, cons2_var') (state(memory_write(pm')(waddr', cons1_var')(s'))))
{-5}	transformer_invariant?(states', memory_write_transformers(pm', address_block(waddr', 1 + length(cons2_var'))))
{-6}	unchanged_memory_invariant?(pm', states', memory_read_transformers(pm', address_block(waddr', 1 + length(cons2_var'))), address_block(waddr', 1 + length(cons2_var')))
{-7}	unchanged_memory_write_invariant?(pm', states', address_block(waddr', 1 + length(cons2_var')))
{-8}	changed_memory_invariant?(pm', states', address_block(waddr', 1 + length(cons2_var')))
{-9}	transformers_ok?(states', memory_read_transformers(pm', address_block(waddr', 1 + length(cons2_var'))))
{-10}	transformers_ok?(states', memory_write_transformers(pm', address_block(waddr', 1 + length(cons2_var'))))
{-11}	states'(s')
{1}	unchanged_memory_invariant?(pm', states', memory_write_transformers(pm', ad- dress_block(waddr' + 1, length(cons2_var'))), singleton(waddr'))
{2}	data(memory_read(pm') (waddr') (state[State, Unit] (memory_write_list_nse(pm')(waddr' + 1, cons2_var') (state(memory_write(pm') (waddr', cons1_var')(s')))))) = cons1_var'

Hiding formulas: 2,

Using lemma unchanged_memory_write_invariant_unchanged_memory_invariant,

Simplifying, rewriting, and recording with decision procedures,

Keeping 1 and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of unchanged_memory_read_list_write_list_nse.2.1.4.

unchanged_memory_read_list_write_list_nse.2.2:

```

{-1} OK?(memory_write(pm')(waddr', cons1_var')(s'))
{-2} OK?(memory_read(pm')
      (waddr')
      (state[State, Unit]
        (memory_write_list_nse(pm')(waddr' + 1, cons2_var')
          (state(memory_write(pm')
            (waddr', cons1_var')(s'))))))))
{-3} OK?(memory_read_list_nse(pm')(waddr' + 1, length(cons2_var'))
      (state(memory_read(pm')
        (waddr')
        (state[State, Unit]
          (memory_write_list_nse(pm')
            (waddr' + 1,
              cons2_var')
            (state
              (memory_write
                (waddr', cons1_var')(s'))))))))
{-4} OK?[State, list[Byte]]
      (OK(state(memory_read_list_nse(pm')(waddr' + 1, length(cons2_var'))
        (state(memory_read(pm')
          (waddr')
          (state[State, Unit]
            (memory_write_list_nse(pm')
              (waddr' + 1,
                cons2_var')
              (state
                (memory_write
                  (waddr', cons1_var')(s')))))))))))

      cons(data(memory_read(pm')
        (waddr')
        (state[State, Unit]
          (memory_write_list_nse(pm')(waddr' + 1, cons2_var')
            (state(memory_write(pm')
              (waddr', cons1_var')(s'))))))),
        data(memory_read_list_nse(pm')(waddr' + 1, length(cons2_var'))
          (state(memory_read(pm')
            (waddr')
            (state[State, Unit]
              (memory_write_list_nse
                (pm')
                (waddr' + 1, cons2_var')
                (state
                  (memory_write(pm')
                    (waddr', cons1_var')(s')))))))))))

{-5} OK?(memory_write_list_nse(pm')(waddr' + 1, cons2_var')
      (state(memory_write(pm')(waddr', cons1_var')(s'))))
{-6} transformer_invariant?(states',
      memory_write_transformers(pm',
        address_block(waddr',
          1 + length(cons2_var'))
      )
2512
{-7} ∀ (waddr: Address, s: State):
      unchanged_memory_invariant?(pm', states',
        memory_read_transformers(pm',
          address_block(waddr,
            length(cons2_var'))
          address_block(waddr, length(cons2_var'))))
      ^

```

Hiding formulas: -4,

Using lemma unchanged_memory_read_list_nse,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 4 subgoals:

unchanged_memory_read_list_write_list_nse.2.2.1:

```

{-1}  data(memory_read_list_nse(pm')(waddr' + 1, length(cons2_var'))
      (state(memory_read(pm')
              (waddr')
              (state[State, Unit]
                (memory_write_list_nse(pm')
                  (waddr' + 1,
                    cons2_var')
                    (state
                      (memory_write
                        (waddr', cons1_var')
                        (s'))))))))
      =
      data(memory_read_list_nse(pm')(waddr' + 1, length(cons2_var'))
          (state[State, Unit]
            (memory_write_list_nse(pm')(waddr' + 1, cons2_var')
              (state(memory_write(pm')
                (waddr', cons1_var')
                (s'))))))))
{-2}  OK?(memory_write(pm')(waddr', cons1_var')(s'))
{-3}  OK?(memory_read(pm')
      (waddr')
      (state[State, Unit]
        (memory_write_list_nse(pm')(waddr' + 1, cons2_var')
          (state(memory_write(pm')
            (waddr', cons1_var')(s'))))))))
{-4}  OK?(memory_read_list_nse(pm')(waddr' + 1, length(cons2_var'))
      (state(memory_read(pm')
              (waddr')
              (state[State, Unit]
                (memory_write_list_nse(pm')
                  (waddr' + 1,
                    cons2_var')
                    (state
                      (memory_write
                        (waddr', cons1_var')
                        (s'))))))))
      (state
        (memory_write
          (waddr', cons1_var')
          (s'))))))))
{-5}  OK?(memory_write_list_nse(pm')(waddr' + 1, cons2_var')
      (state(memory_write(pm')(waddr', cons1_var')(s'))))
{-6}  transformer_invariant?(states',
      memory_write_transformers(pm',
        address_block(waddr',
          1 + length(cons2_var'))
      )
      (state
        (memory_write
          (waddr', cons1_var')
          (s'))))))))
{-7}  ∀ (waddr: Address, s: State):
      unchanged_memory_invariant?(pm', states',
        memory_read_transformers(pm',
          address_block(waddr,
            length(cons2_var'))
        )
        address_block(waddr, length(cons2_var')))
      ∧
      unchanged_memory_write_invariant?(pm', states',
        address_block(waddr, length(cons2_var')))
      ∧
      changed_memory_invariant?(pm', states', address_block(waddr, length(cons2_var'))) ∧
      transformers_ok?(states',
        memory_read_transformers(pm',
          address_block(waddr, length(cons2_var')))
        )
      ∧
      transformers_ok?(states',
        memory_write_transformers(pm',
          address_block(waddr,
            length(cons2_var'))
        )
      )
      ∧ states'(s)

```

Replacing using formula -1,

Rewriting using -7, matching in *,

we get 6 subgoals:

unchanged_memory_read_list_write_list_nse.2.2.1.1:

```

{-1}  data(memory_read_list_nse(pm')(waddr' + 1, length(cons2_var'))
      (state(memory_read(pm')
              (waddr')
              (state[State, Unit]
                (memory_write_list_nse(pm')
                  (waddr' + 1,
                    cons2_var')
                    (state
                      (memory_write
                        (waddr', cons1_var')
                        (s'))))))))
      =
      data(memory_read_list_nse(pm')(waddr' + 1, length(cons2_var'))
          (state[State, Unit]
            (memory_write_list_nse(pm')(waddr' + 1, cons2_var')
              (state(memory_write(pm')
                (waddr', cons1_var')
                (s'))))))))
{-2}  OK?(memory_write(pm')(waddr', cons1_var')(s'))
{-3}  OK?(memory_read(pm')
      (waddr')
      (state[State, Unit]
        (memory_write_list_nse(pm')(waddr' + 1, cons2_var')
          (state(memory_write(pm')
            (waddr', cons1_var')(s'))))))))
{-4}  OK?(memory_read_list_nse(pm')(waddr' + 1, length(cons2_var'))
      (state(memory_read(pm')
              (waddr')
              (state[State, Unit]
                (memory_write_list_nse(pm')
                  (waddr' + 1,
                    cons2_var')
                    (state
                      (memory_write
                        (waddr', cons1_var')
                        (s'))))))))
      (state
        (memory_write
          (waddr', cons1_var')
          (s'))))))))
{-5}  OK?(memory_write_list_nse(pm')(waddr' + 1, cons2_var')
      (state(memory_write(pm')(waddr', cons1_var')(s'))))
{-6}  transformer_invariant?(states',
      memory_write_transformers(pm',
        address_block(waddr',
          1 + length(cons2_var'))))
{-7}  ∀ (waddr: Address, s: State):
      unchanged_memory_invariant?(pm', states',
        memory_read_transformers(pm',
          address_block(waddr,
            length(cons2_var'))))
      ∧
      unchanged_memory_write_invariant?(pm', states',
        address_block(waddr, length(cons2_var')))
      ∧
      changed_memory_invariant?(pm', states', address_block(waddr, length(cons2_var'))) ∧
      transformers_ok?(states',
        memory_read_transformers(pm',
          address_block(waddr, length(cons2_var')))
        ∧
        transformers_ok?(states',
          memory_write_transformers(pm',
            address_block(waddr,
              length(cons2_var')))
          ∧ states'(s)

```

Keeping (-8 1) and hiding *,

Using lemma unchanged_memory_invariant_mono,

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of unchanged_memory_read_list_write_list_nse.2.2.1.1.

unchanged_memory_read_list_write_list_nse.2.2.1.2:

```

{-1}  data(memory_read_list_nse(pm')(waddr' + 1, length(cons2_var'))
      (state(memory_read(pm')
              (waddr')
              (state[State, Unit]
                (memory_write_list_nse(pm')
                  (waddr' + 1,
                    cons2_var')
                    (state
                      (memory_write
                        (waddr', cons1_var')
                        (s'))))))))
      =
      data(memory_read_list_nse(pm')(waddr' + 1, length(cons2_var'))
          (state[State, Unit]
            (memory_write_list_nse(pm')(waddr' + 1, cons2_var')
              (state(memory_write(pm')
                (waddr', cons1_var')
                (s'))))))))
{-2}  OK?(memory_write(pm')(waddr', cons1_var')(s'))
{-3}  OK?(memory_read(pm')
      (waddr')
      (state[State, Unit]
        (memory_write_list_nse(pm')(waddr' + 1, cons2_var')
          (state(memory_write(pm')
            (waddr', cons1_var')(s'))))))))
{-4}  OK?(memory_read_list_nse(pm')(waddr' + 1, length(cons2_var'))
      (state(memory_read(pm')
              (waddr')
              (state[State, Unit]
                (memory_write_list_nse(pm')
                  (waddr' + 1,
                    cons2_var')
                    (state
                      (memory_write
                        (waddr', cons1_var')
                        (s'))))))))
      (state
        (memory_write
          (waddr', cons1_var')
          (s'))))))))
{-5}  OK?(memory_write_list_nse(pm')(waddr' + 1, cons2_var')
      (state(memory_write(pm')(waddr', cons1_var')(s'))))
{-6}  transformer_invariant?(states',
      memory_write_transformers(pm',
        address_block(waddr',
          1 + length(cons2_var'))))
{-7}  ∀ (waddr: Address, s: State):
      unchanged_memory_invariant?(pm', states',
        memory_read_transformers(pm',
          address_block(waddr,
            length(cons2_var'))))
      ∧
      unchanged_memory_write_invariant?(pm', states',
        address_block(waddr, length(cons2_var')))
      ∧
      changed_memory_invariant?(pm', states', address_block(waddr, length(cons2_var'))) ∧
      transformers_ok?(states',
        memory_read_transformers(pm',
          address_block(waddr, length(cons2_var')))
        ∧
        memory_write_transformers(pm',
          address_block(waddr,
            length(cons2_var')))
        ∧ states'(s)

```

Keeping (-9 1) and hiding *,

Using lemma unchanged_memory_write_invariant_mono_addresses,

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of unchanged_memory_read_list_write_list_nse.2.2.1.2.

unchanged_memory_read_list_write_list_nse.2.2.1.3:

```

{-1}  data(memory_read_list_nse(pm')(waddr' + 1, length(cons2_var'))
      (state(memory_read(pm')
              (waddr')
              (state[State, Unit]
                (memory_write_list_nse(pm')
                  (waddr' + 1,
                    cons2_var')
                    (state
                      (memory_write
                        (waddr', cons1_var')
                        (s'))))))))
      =
      data(memory_read_list_nse(pm')(waddr' + 1, length(cons2_var'))
          (state[State, Unit]
            (memory_write_list_nse(pm')(waddr' + 1, cons2_var')
              (state(memory_write(pm')
                (waddr', cons1_var')
                (s'))))))))
{-2}  OK?(memory_write(pm')(waddr', cons1_var')(s'))
{-3}  OK?(memory_read(pm')
      (waddr')
      (state[State, Unit]
        (memory_write_list_nse(pm')(waddr' + 1, cons2_var')
          (state(memory_write(pm')
            (waddr', cons1_var')(s'))))))))
{-4}  OK?(memory_read_list_nse(pm')(waddr' + 1, length(cons2_var'))
      (state(memory_read(pm')
              (waddr')
              (state[State, Unit]
                (memory_write_list_nse(pm')
                  (waddr' + 1,
                    cons2_var')
                    (state
                      (memory_write
                        (waddr', cons1_var')
                        (s'))))))))
      (state
        (memory_write
          (waddr', cons1_var')
          (s'))))))))
{-5}  OK?(memory_write_list_nse(pm')(waddr' + 1, cons2_var')
      (state(memory_write(pm')(waddr', cons1_var')(s'))))
{-6}  transformer_invariant?(states',
      memory_write_transformers(pm',
        address_block(waddr',
          1 + length(cons2_var'))
      )
      (state
        (memory_write
          (waddr', cons1_var')
          (s'))))))))
{-7}  ∀ (waddr: Address, s: State):
      unchanged_memory_invariant?(pm', states',
        memory_read_transformers(pm',
          address_block(waddr,
            length(cons2_var'))
        )
        address_block(waddr, length(cons2_var')))
      ∧
      unchanged_memory_write_invariant?(pm', states',
        address_block(waddr, length(cons2_var')))
      ∧
      changed_memory_invariant?(pm', states', address_block(waddr, length(cons2_var'))) ∧
      transformers_ok?(states',
        memory_read_transformers(pm',
          address_block(waddr, length(cons2_var')))
        )
      ∧
      transformers_ok?(states',
        memory_write_transformers(pm',
          address_block(waddr,
            length(cons2_var'))
        )
      )
      ∧ states'(s)

```

Keeping (-10 1) and hiding *,

Using lemma changed_memory_invariant_mono,

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of unchanged_memory_read_list_write_list_nse.2.2.1.3.

unchanged_memory_read_list_write_list_nse.2.2.1.4:

```

{-1}  data(memory_read_list_nse(pm')(waddr' + 1, length(cons2_var'))
      (state(memory_read(pm')
              (waddr')
              (state[State, Unit]
                (memory_write_list_nse(pm')
                  (waddr' + 1,
                    cons2_var')
                    (state
                      (memory_write
                        (waddr', cons1_var')
                        (s'))))))))
      =
      data(memory_read_list_nse(pm')(waddr' + 1, length(cons2_var'))
          (state[State, Unit]
            (memory_write_list_nse(pm')(waddr' + 1, cons2_var')
              (state(memory_write(pm')
                (waddr', cons1_var')
                (s'))))))))
{-2}  OK?(memory_write(pm')(waddr', cons1_var')(s'))
{-3}  OK?(memory_read(pm')
      (waddr')
      (state[State, Unit]
        (memory_write_list_nse(pm')(waddr' + 1, cons2_var')
          (state(memory_write(pm')
            (waddr', cons1_var')(s'))))))))
{-4}  OK?(memory_read_list_nse(pm')(waddr' + 1, length(cons2_var'))
      (state(memory_read(pm')
              (waddr')
              (state[State, Unit]
                (memory_write_list_nse(pm')
                  (waddr' + 1,
                    cons2_var')
                    (state
                      (memory_write
                        (waddr', cons1_var')
                        (s'))))))))
      (state
        (memory_write
          (waddr', cons1_var')
          (s'))))))))
{-5}  OK?(memory_write_list_nse(pm')(waddr' + 1, cons2_var')
      (state(memory_write(pm')(waddr', cons1_var')(s'))))
{-6}  transformer_invariant?(states',
      memory_write_transformers(pm',
        address_block(waddr',
          1 + length(cons2_var'))))
{-7}  ∀ (waddr: Address, s: State):
      unchanged_memory_invariant?(pm', states',
        memory_read_transformers(pm',
          address_block(waddr,
            length(cons2_var'))))
      ∧
      unchanged_memory_write_invariant?(pm', states',
        address_block(waddr, length(cons2_var')))
      ∧
      changed_memory_invariant?(pm', states', address_block(waddr, length(cons2_var'))) ∧
      transformers_ok?(states',
        memory_read_transformers(pm',
          address_block(waddr, length(cons2_var')))
        ∧
        memory_write_transformers(pm',
          address_block(waddr,
            length(cons2_var')))
        ∧ states'(s)

```

Keeping (-11 1) and hiding *,

Using lemma transformers_ok_mono_transformers,

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of unchanged_memory_read_list_write_list_nse.2.2.1.4.

unchanged_memory_read_list_write_list_nse.2.2.1.5:

```

{-1}  data(memory_read_list_nse(pm')(waddr' + 1, length(cons2_var'))
      (state(memory_read(pm')
              (waddr')
              (state[State, Unit]
                (memory_write_list_nse(pm')
                  (waddr' + 1,
                    cons2_var')
                    (state
                      (memory_write
                        (waddr', cons1_var')
                        (s'))))))))
      =
      data(memory_read_list_nse(pm')(waddr' + 1, length(cons2_var'))
        (state[State, Unit]
          (memory_write_list_nse(pm')(waddr' + 1, cons2_var')
            (state(memory_write(pm')
                      (waddr', cons1_var')
                      (s'))))))))
{-2}  OK?(memory_write(pm')(waddr', cons1_var')(s'))
{-3}  OK?(memory_read(pm')
      (waddr')
      (state[State, Unit]
        (memory_write_list_nse(pm')(waddr' + 1, cons2_var')
          (state(memory_write(pm')
                    (waddr', cons1_var')(s'))))))))
{-4}  OK?(memory_read_list_nse(pm')(waddr' + 1, length(cons2_var'))
      (state(memory_read(pm')
              (waddr')
              (state[State, Unit]
                (memory_write_list_nse(pm')
                  (waddr' + 1,
                    cons2_var')
                    (state
                      (memory_write
                        (waddr', cons1_var')
                        (s'))))))))
      (state
        (memory_write
          (waddr', cons1_var')
          (s'))))))))
{-5}  OK?(memory_write_list_nse(pm')(waddr' + 1, cons2_var')
      (state(memory_write(pm')(waddr', cons1_var')(s'))))
{-6}  transformer_invariant?(states',
      memory_write_transformers(pm',
        address_block(waddr',
          1 + length(cons2_var'))
      )
      )
{-7}  ∀ (waddr: Address, s: State):
      unchanged_memory_invariant?(pm', states',
        memory_read_transformers(pm',
          address_block(waddr,
            length(cons2_var'))
        )
        address_block(waddr, length(cons2_var'))
      )
      ∧
      unchanged_memory_write_invariant?(pm', states',
        address_block(waddr, length(cons2_var'))
      )
      ∧
      changed_memory_invariant?(pm', states', address_block(waddr, length(cons2_var'))
      )
      ∧
      transformers_ok?(states',
        memory_read_transformers(pm',
          address_block(waddr, length(cons2_var'))
        )
        ∧
        transformers_ok?(states',
          memory_write_transformers(pm',
            address_block(waddr,
              length(cons2_var'))
          )
        )
      )
      ∧
      states'(s)

```

Keeping (-12 1) and hiding *,

Using lemma transformers_ok_mono_transformers,

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of unchanged_memory_read_list_write_list_nse.2.2.1.5.

unchanged_memory_read_list_write_list_nse.2.2.1.6:

```

{-1}  data(memory_read_list_nse(pm')(waddr' + 1, length(cons2_var'))
      (state(memory_read(pm')
              (waddr')
              (state[State, Unit]
                (memory_write_list_nse(pm')
                  (waddr' + 1,
                    cons2_var')
                    (state
                      (memory_write
                        (waddr', cons1_var')
                        (s'))))))))
      =
      data(memory_read_list_nse(pm')(waddr' + 1, length(cons2_var'))
        (state[State, Unit]
          (memory_write_list_nse(pm')(waddr' + 1, cons2_var')
            (state(memory_write(pm')
                      (waddr', cons1_var')
                      (s'))))))))
{-2}  OK?(memory_write(pm')(waddr', cons1_var')(s'))
{-3}  OK?(memory_read(pm')
      (waddr')
      (state[State, Unit]
        (memory_write_list_nse(pm')(waddr' + 1, cons2_var')
          (state(memory_write(pm')
                    (waddr', cons1_var')(s'))))))))
{-4}  OK?(memory_read_list_nse(pm')(waddr' + 1, length(cons2_var'))
      (state(memory_read(pm')
              (waddr')
              (state[State, Unit]
                (memory_write_list_nse(pm')
                  (waddr' + 1,
                    cons2_var')
                    (state
                      (memory_write
                        (waddr', cons1_var')
                        (s'))))))))
      (state
        (memory_write
          (waddr', cons1_var')
          (s'))))))))
{-5}  OK?(memory_write_list_nse(pm')(waddr' + 1, cons2_var')
      (state(memory_write(pm')(waddr', cons1_var')(s'))))
{-6}  transformer_invariant?(states',
      memory_write_transformers(pm',
        address_block(waddr',
          1 + length(cons2_var'))
      )
{-7}  ∀ (waddr: Address, s: State):
      unchanged_memory_invariant?(pm', states',
        memory_read_transformers(pm',
          address_block(waddr,
            length(cons2_var'))
        )
      )
      ∧
      unchanged_memory_write_invariant?(pm', states',
        address_block(waddr, length(cons2_var'))
      )
      ∧
      changed_memory_invariant?(pm', states', address_block(waddr, length(cons2_var'))
      ) ∧
      transformers_ok?(states',
        memory_read_transformers(pm',
          address_block(waddr, length(cons2_var'))
        )
      )
      ∧
      transformers_ok?(states',
        memory_write_transformers(pm',
          address_block(waddr,
            length(cons2_var'))
        )
      )
      ∧ states'(s)

```

Using lemma `expr_transformer_invariant_next_ok`,

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `unchanged_memory_read_list_write_list_nse.2.2.1.6`.

unchanged_memory_read_list_write_list_nse.2.2.2:

```

{-1} OK?(memory_write(pm')(waddr', cons1_var')(s'))
{-2} OK?(memory_read(pm')
      (waddr')
      (state[State, Unit]
        (memory_write_list_nse(pm')(waddr' + 1, cons2_var')
          (state(memory_write(pm')
            (waddr', cons1_var')(s'))))))))
{-3} OK?(memory_read_list_nse(pm')(waddr' + 1, length(cons2_var'))
      (state(memory_read(pm')
        (waddr')
        (state[State, Unit]
          (memory_write_list_nse(pm')
            (waddr' + 1,
              cons2_var')
            (state
              (memory_write
                (waddr', cons1_var')(s'))))))))
{-4} OK?(memory_write_list_nse(pm')(waddr' + 1, cons2_var')
      (state(memory_write(pm')(waddr', cons1_var')(s'))))
{-5} transformer_invariant?(states',
      memory_write_transformers(pm',
        address_block(waddr',
          1 + length(cons2_var')))
{-6} ∀ (waddr: Address, s: State):
      unchanged_memory_invariant?(pm', states',
        memory_read_transformers(pm',
          address_block(waddr,
            length(cons2_var')))
        address_block(waddr, length(cons2_var')))
      ^
      unchanged_memory_write_invariant?(pm', states',
        address_block(waddr, length(cons2_var')))
      ^
      changed_memory_invariant?(pm', states', address_block(waddr, length(cons2_var'))) ^
      transformers_ok?(states',
        memory_read_transformers(pm',
          address_block(waddr, length(cons2_var')))
        ^
        transformers_ok?(states',
          memory_write_transformers(pm',
            address_block(waddr,
              length(cons2_var')))
          ^ states'(s)
        )
      )
      ⊃
      data(memory_read_list_nse(pm')(waddr, length(cons2_var'))
        (state(memory_write_list_nse(pm')(waddr, cons2_var')
          (s))))
      = cons2_var'
{-7} unchanged_memory_invariant?(pm', states',
      memory_read_transformers(pm',
        address_block(waddr',
          1 + length(cons2_var')))
        address_block(waddr', 1 + length(cons2_var')))
2528 {-8} unchanged_memory_write_invariant?(pm', states',
      address_block(waddr', 1 + length(cons2_var')))
{-9} changed_memory_invariant?(pm', states',
      address_block(waddr', 1 + length(cons2_var')))
{-10} transformers_ok?(states',
      memory_read_transformers(pm',
        address_block(waddr',
          1 + length(cons2_var')))
      transformers_ok?(states',
        memory_write_transformers(pm',
          address_block(waddr',
            1 + length(cons2_var')))

```

Hiding formulas: -2, -3, -6, 2,

Using lemma transformer_invariant_write_list_nse,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 3 subgoals:

unchanged_memory_read_list_write_list_nse.2.2.2.1:

{-1}	result_pred(states') (expr_2_super(memory_write_list_nse(pm')(waddr' + 1, cons2_var')) (state(memory_write(pm')(waddr', cons1_var')(s'))))
{-2}	OK?(memory_write(pm')(waddr', cons1_var')(s'))
{-3}	OK?(memory_write_list_nse(pm')(waddr' + 1, cons2_var') (state(memory_write(pm')(waddr', cons1_var')(s'))))
{-4}	transformer_invariant?(states', memory_write_transformers(pm', address_block(waddr', 1 + length(cons2_var'))))
{-5}	unchanged_memory_invariant?(pm', states', memory_read_transformers(pm', address_block(waddr', 1 + length(cons2_var'))), address_block(waddr', 1 + length(cons2_var'))
{-6}	unchanged_memory_write_invariant?(pm', states', address_block(waddr', 1 + length(cons2_var'))
{-7}	changed_memory_invariant?(pm', states', address_block(waddr', 1 + length(cons2_var'))
{-8}	transformers_ok?(states', memory_read_transformers(pm', address_block(waddr', 1 + length(cons2_var'))))
{-9}	transformers_ok?(states', memory_write_transformers(pm', address_block(waddr', 1 + length(cons2_var'))))
{-10}	states'(s')
{1}	states'(state [State, Unit] (memory_write_list_nse(pm')(waddr' + 1, cons2_var') (state(memory_write(pm') (waddr', cons1_var')(s'))))

Rewriting using expr_result_pred_next_state, matching in *,

This completes the proof of unchanged_memory_read_list_write_list_nse.2.2.2.1.

C Proof scripts

unchanged_memory_read_list_write_list_nse.2.2.2.2:

{-1}	OK?(memory_write(pm')(waddr', cons1_var')(s'))
{-2}	OK?(memory_write_list_nse(pm')(waddr' + 1, cons2_var' (state(memory_write(pm')(waddr', cons1_var')(s')))))
{-3}	transformer_invariant?(states', memory_write_transformers(pm', address_block(waddr', 1 + length(cons2_var')))
{-4}	unchanged_memory_invariant?(pm', states', memory_read_transformers(pm', address_block(waddr', 1 + length(cons2_var')))
{-5}	unchanged_memory_write_invariant?(pm', states', address_block(waddr', 1 + length(cons2_var')))
{-6}	changed_memory_invariant?(pm', states', address_block(waddr', 1 + length(cons2_var')))
{-7}	transformers_ok?(states', memory_read_transformers(pm', address_block(waddr', 1 + length(cons2_var'))))
{-8}	transformers_ok?(states', memory_write_transformers(pm', address_block(waddr', 1 + length(cons2_var'))))
{-9}	states'(s')
{1}	states'(state(memory_write(pm')(waddr', cons1_var')(s')))
{2}	states'(state[State, Unit] (memory_write_list_nse(pm')(waddr' + 1, cons2_var') (state(memory_write(pm') (waddr', cons1_var')(s')))))

Keeping (-3 1) and hiding *,

Using lemma `expr_transformer_invariant_next_ok[State, Unit]`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `unchanged_memory_read_list_write_list_nse.2.2.2.2`.

unchanged_memory_read_list_write_list_nse.2.2.2.3:

{-1}	OK?(memory_write(pm')(waddr', cons1_var')(s'))
{-2}	OK?(memory_write_list_nse(pm')(waddr' + 1, cons2_var') (state(memory_write(pm')(waddr', cons1_var')(s'))))
{-3}	transformer_invariant?(states', memory_write_transformers(pm', address_block(waddr', 1 + length(cons2_var'))))
{-4}	unchanged_memory_invariant?(pm', states', memory_read_transformers(pm', address_block(waddr', 1 + length(cons2_var'))), address_block(waddr', 1 + length(cons2_var')))
{-5}	unchanged_memory_write_invariant?(pm', states', address_block(waddr', 1 + length(cons2_var')))
{-6}	changed_memory_invariant?(pm', states', address_block(waddr', 1 + length(cons2_var')))
{-7}	transformers_ok?(states', memory_read_transformers(pm', address_block(waddr', 1 + length(cons2_var'))))
{-8}	transformers_ok?(states', memory_write_transformers(pm', address_block(waddr', 1 + length(cons2_var'))))
{-9}	states'(s')
{1}	transformer_invariant?(states', memory_write_transformers(pm', address_block(waddr' + 1, length(cons2_var'))))
{2}	states'(state[State, Unit] (memory_write_list_nse(pm')(waddr' + 1, cons2_var') (state(memory_write(pm') (waddr', cons1_var')(s'))))

Keeping (-3 1) and hiding *,

Using lemma transformer_invariant_mono_transformers,

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of unchanged_memory_read_list_write_list_nse.2.2.2.3.

unchanged_memory_read_list_write_list_nse.2.2.3:

```

{-1} OK?(memory_write(pm')(waddr', cons1_var')(s'))
{-2} OK?(memory_read(pm')
      (waddr')
      (state[State, Unit]
        (memory_write_list_nse(pm')(waddr' + 1, cons2_var')
          (state(memory_write(pm')
            (waddr', cons1_var')(s'))))))))
{-3} OK?(memory_read_list_nse(pm')(waddr' + 1, length(cons2_var'))
      (state(memory_read(pm')
        (waddr')
        (state[State, Unit]
          (memory_write_list_nse(pm')
            (waddr' + 1,
              cons2_var')
            (state
              (memory_write
                (waddr', cons1_var')(s'))))))))
{-4} OK?(memory_write_list_nse(pm')(waddr' + 1, cons2_var')
      (state(memory_write(pm')(waddr', cons1_var')(s'))))
{-5} transformer_invariant?(states',
      memory_write_transformers(pm',
        address_block(waddr',
          1 + length(cons2_var')))
{-6} ∀ (waddr: Address, s: State):
      unchanged_memory_invariant?(pm', states',
        memory_read_transformers(pm',
          address_block(waddr,
            length(cons2_var')))
        address_block(waddr, length(cons2_var')))
      ^
      unchanged_memory_write_invariant?(pm', states',
        address_block(waddr, length(cons2_var')))
      ^
      changed_memory_invariant?(pm', states', address_block(waddr, length(cons2_var'))) ^
      transformers_ok?(states',
        memory_read_transformers(pm',
          address_block(waddr, length(cons2_var')))
        ^
        transformers_ok?(states',
          memory_write_transformers(pm',
            address_block(waddr,
              length(cons2_var')))
          ^ states'(s)
        )
      )
      ⊃
      data(memory_read_list_nse(pm')(waddr, length(cons2_var'))
        (state(memory_write_list_nse(pm')(waddr, cons2_var')
          (s))))
      = cons2_var'
{-7} unchanged_memory_invariant?(pm', states',
      memory_read_transformers(pm',
        address_block(waddr',
          1 + length(cons2_var')))
        address_block(waddr', 1 + length(cons2_var')))
2532 {-8} unchanged_memory_write_invariant?(pm', states',
      address_block(waddr', 1 + length(cons2_var')))
{-9} changed_memory_invariant?(pm', states',
      address_block(waddr', 1 + length(cons2_var')))
{-10} transformers_ok?(states',
      memory_read_transformers(pm',
        address_block(waddr',
          1 + length(cons2_var')))
      transformers_ok?(states',
        memory_write_transformers(pm',
          address_block(waddr',
            1 + length(cons2_var')))

```

Keeping (-10 1) and hiding *,

Using lemma transformers_ok_mono_transformers,

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of unchanged_memory_read_list_write_list_nse.2.2.3.

unchanged_memory_read_list_write_list_nse.2.2.4:

```

{-1} OK?(memory_write(pm')(waddr', cons1_var')(s'))
{-2} OK?(memory_read(pm')
      (waddr')
      (state[State, Unit]
        (memory_write_list_nse(pm')(waddr' + 1, cons2_var')
          (state(memory_write(pm')
            (waddr', cons1_var')(s'))))))))
{-3} OK?(memory_read_list_nse(pm')(waddr' + 1, length(cons2_var'))
      (state(memory_read(pm')
        (waddr')
        (state[State, Unit]
          (memory_write_list_nse(pm')
            (waddr' + 1,
              cons2_var')
            (state
              (memory_write
                (waddr', cons1_var')(s'))))))))
{-4} OK?(memory_write_list_nse(pm')(waddr' + 1, cons2_var')
      (state(memory_write(pm')(waddr', cons1_var')(s'))))
{-5} transformer_invariant?(states',
      memory_write_transformers(pm',
        address_block(waddr',
          1 + length(cons2_var')))
{-6} ∀ (waddr: Address, s: State):
      unchanged_memory_invariant?(pm', states',
        memory_read_transformers(pm',
          address_block(waddr,
            length(cons2_var')))
        address_block(waddr, length(cons2_var')))
      ^
      unchanged_memory_write_invariant?(pm', states',
        address_block(waddr, length(cons2_var')))
      ^
      changed_memory_invariant?(pm', states', address_block(waddr, length(cons2_var'))) ^
      transformers_ok?(states',
        memory_read_transformers(pm',
          address_block(waddr, length(cons2_var')))
        ^
        transformers_ok?(states',
          memory_write_transformers(pm',
            address_block(waddr,
              length(cons2_var')))
          ^ states'(s)
        )
      )
      ⊃
      data(memory_read_list_nse(pm')(waddr, length(cons2_var'))
        (state(memory_write_list_nse(pm')(waddr, cons2_var')
          (s))))
      = cons2_var'
{-7} unchanged_memory_invariant?(pm', states',
      memory_read_transformers(pm',
        address_block(waddr',
          1 + length(cons2_var')))
        address_block(waddr', 1 + length(cons2_var')))
2534 {-8} unchanged_memory_write_invariant?(pm', states',
      address_block(waddr', 1 + length(cons2_var')))
{-9} changed_memory_invariant?(pm', states',
      address_block(waddr', 1 + length(cons2_var')))
{-10} transformers_ok?(states',
      memory_read_transformers(pm',
        address_block(waddr',
          1 + length(cons2_var')))
      transformers_ok?(states',
        memory_write_transformers(pm',
          address_block(waddr',
            1 + length(cons2_var')))

```

Keeping (-7 1) and hiding *,

Rewriting using unchanged_memory_invariant_union_transformers, matching in *,

we get 2 subgoals:

unchanged_memory_read_list_write_list_nse.2.2.4.1:

{-1}	unchanged_memory_invariant?(pm', states', memory_read_transformers(pm', address_block(waddr', 1 + length(cons2_var'))), address_block(waddr', 1 + length(cons2_var')))
{1}	unchanged_memory_invariant?(pm', states', memory_read_transformers(pm', ad- dress_block(waddr' + 1, length(cons2_var'))), address_block(waddr', 1 + length(cons2_var')))
{2}	unchanged_memory_invariant?(pm', states', (memory_read_transformers(pm', address_block(waddr' + 1, length(cons2- address_block(waddr', 1 + length(cons2_var')))

Hiding formulas: 2,

Using lemma unchanged_memory_invariant_mono,

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of unchanged_memory_read_list_write_list_nse.2.2.4.1.

unchanged_memory_read_list_write_list_nse.2.2.4.2:

{-1}	unchanged_memory_invariant?(pm', states', memory_read_transformers(pm', address_block(waddr', 1 + length(cons2_var'))), address_block(waddr', 1 + length(cons2_var')))
{1}	unchanged_memory_invariant?(pm', states', singleton(expr_2_super(memory_read(pm')(waddr'))), address_block(waddr', 1 + length(cons2_var')))
{2}	unchanged_memory_invariant?(pm', states', (memory_read_transformers(pm', address_block(waddr' + 1, length(cons2- address_block(waddr', 1 + length(cons2_var')))

Hiding formulas: 2,

Using lemma unchanged_memory_invariant_mono,

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of unchanged_memory_read_list_write_list_nse.2.2.4.2.

unchanged_memory_read_list_write_list_nse.3:

$$\begin{array}{l}
\{1\} \quad \forall (bl: \text{list}[\text{Byte}], waddr: \text{Address}, s: \text{State}): \\
\quad \text{unchanged_memory_invariant?}(pm', \text{states}', \\
\quad \quad \quad \text{memory_read_transformers}(pm', \\
\quad \quad \quad \quad \quad \text{ad-} \\
\quad \text{dress_block}(waddr, \text{length}(bl))), \\
\quad \quad \quad \quad \quad \text{address_block}(waddr, \text{length}(bl))) \\
\quad \wedge \\
\quad \text{unchanged_memory_write_invariant?}(pm', \text{states}', \text{address_block}(waddr, \text{length}(bl))) \wedge \\
\quad \text{changed_memory_invariant?}(pm', \text{states}', \text{address_block}(waddr, \text{length}(bl))) \wedge \\
\quad \text{transformers_ok?}(\text{states}', \\
\quad \quad \quad \text{memory_read_transformers}(pm', \text{ad-} \\
\quad \text{dress_block}(waddr, \text{length}(bl)))) \\
\quad \wedge \\
\quad \text{transformers_ok?}(\text{states}', \\
\quad \quad \quad \text{memory_write_transformers}(pm', \text{ad-} \\
\quad \text{dress_block}(waddr, \text{length}(bl)))) \\
\quad \wedge \text{states}'(s) \\
\quad \supset \\
\quad \text{OK?}[\text{State}, \text{list}[\text{Byte}]] \\
\quad \quad \quad (\text{memory_read_list_nse}[\text{State}] \\
\quad \quad \quad \quad (pm')(waddr, \text{length}[\text{Byte}](bl)) \\
\quad \quad \quad \quad (\text{state}[\text{State}, \text{Unit}](\text{memory_write_list_nse}[\text{State}](pm')(waddr, bl)(s)))) \\
\{2\} \quad \forall (waddr: \text{Address}, s: \text{State}): \\
\quad \text{unchanged_memory_invariant?}(pm', \text{states}', \\
\quad \quad \quad \text{memory_read_transformers}(pm', \\
\quad \quad \quad \quad \quad \text{ad-} \\
\quad \text{dress_block}(waddr, \text{length}(bl'))), \\
\quad \quad \quad \quad \quad \text{address_block}(waddr, \text{length}(bl'))) \\
\quad \wedge \\
\quad \text{unchanged_memory_write_invariant?}(pm', \text{states}', \text{address_block}(waddr, \text{length}(bl'))) \wedge \\
\quad \text{changed_memory_invariant?}(pm', \text{states}', \text{address_block}(waddr, \text{length}(bl'))) \wedge \\
\quad \text{transformers_ok?}(\text{states}', \\
\quad \quad \quad \text{memory_read_transformers}(pm', \text{ad-} \\
\quad \text{dress_block}(waddr, \text{length}(bl')))) \\
\quad \wedge \\
\quad \text{transformers_ok?}(\text{states}', \\
\quad \quad \quad \text{memory_write_transformers}(pm', \text{ad-} \\
\quad \text{dress_block}(waddr, \text{length}(bl')))) \\
\quad \wedge \text{states}'(s) \\
\quad \supset \\
\quad \text{data}(\text{memory_read_list_nse}(pm')(waddr, \text{length}(bl'))) \\
\quad \quad \quad (\text{state}(\text{memory_write_list_nse}(pm')(waddr, bl')(s)))) \\
\quad = bl'
\end{array}$$

Hiding formulas: 2,

Repeatedly Skolemizing and flattening,

Using lemma unchanged_memory_read_list_write_list_nse_TCC2,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of unchanged_memory_read_list_write_list_nse.3.

unchanged_memory_read_list_write_list_nse.4:

{1}	$\forall (bl: \text{list}[\text{Byte}], waddr: \text{Address}, s: \text{State}):$ $\text{unchanged_memory_invariant?}(pm', \text{states}',$ $\qquad \text{memory_read_transformers}(pm',$ $\qquad \text{address_block}(waddr, \text{length}(bl))),$ $\qquad \text{address_block}(waddr, \text{length}(bl)))$ \wedge $\text{unchanged_memory_write_invariant?}(pm', \text{states}', \text{address_block}(waddr, \text{length}(bl))) \wedge$ $\text{changed_memory_invariant?}(pm', \text{states}', \text{address_block}(waddr, \text{length}(bl))) \wedge$ $\text{transformers_ok?}(\text{states}',$ $\qquad \text{memory_read_transformers}(pm', \text{ad-}$ $\text{dress_block}(waddr, \text{length}(bl))))$ \wedge $\text{transformers_ok?}(\text{states}',$ $\qquad \text{memory_write_transformers}(pm', \text{ad-}$ $\text{dress_block}(waddr, \text{length}(bl))))$ $\wedge \text{states}'(s)$ \supset $\text{OK?}[\text{State}, \text{Unit}](\text{memory_write_list_nse}[\text{State}](pm')(waddr, bl)(s)) \vee$ $\text{Exception?}[\text{State}, \text{Unit}](\text{memory_write_list_nse}[\text{State}](pm')(waddr, bl)(s))$
{2}	$\forall (waddr: \text{Address}, s: \text{State}):$ $\text{unchanged_memory_invariant?}(pm', \text{states}',$ $\qquad \text{memory_read_transformers}(pm',$ $\qquad \text{ad-}$ $\text{dress_block}(waddr, \text{length}(bl'))),$ $\qquad \text{address_block}(waddr, \text{length}(bl'))))$ \wedge $\text{unchanged_memory_write_invariant?}(pm', \text{states}', \text{address_block}(waddr, \text{length}(bl'))) \wedge$ $\text{changed_memory_invariant?}(pm', \text{states}', \text{address_block}(waddr, \text{length}(bl'))) \wedge$ $\text{transformers_ok?}(\text{states}',$ $\qquad \text{memory_read_transformers}(pm', \text{ad-}$ $\text{dress_block}(waddr, \text{length}(bl'))))$ \wedge $\text{transformers_ok?}(\text{states}',$ $\qquad \text{memory_write_transformers}(pm', \text{ad-}$ $\text{dress_block}(waddr, \text{length}(bl'))))$ $\wedge \text{states}'(s)$ \supset $\text{data}(\text{memory_read_list_nse}(pm')(waddr, \text{length}(bl'))$ $\qquad \text{(state}(\text{memory_write_list_nse}(pm')(waddr, bl')(s))))$ $= bl'$

Hiding formulas: 2,

Repeatedly Skolemizing and flattening,

Using lemma unchanged_memory_read_list_write_list_nse_TCC1,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of unchanged_memory_read_list_write_list_nse.4.

Q.E.D.

C.123.42

Memory_Change_2.unchanged_memory_read_list_write_list_TCC1

Terse proof for unchanged_memory_read_list_write_list_TCC1.

unchanged_memory_read_list_write_list_TCC1:

$$\begin{aligned}
 &\{1\} \quad \forall (pm: \text{Memory_struct}[\text{State}], \text{states}: \text{PRED}[\text{State}], \text{waddr}: \text{Ad-} \\
 &\quad \text{dress, bl}: \text{list}[\text{Byte}], \\
 &\quad \quad s: \text{State}): \\
 &\quad \text{unchanged_memory_invariant?}(pm, \text{states}, \\
 &\quad \quad \quad \text{(memory_read_transformers}(pm, \text{address_block}(\text{waddr}, \text{length}(bl)), \\
 &\quad \quad \quad \text{address_block}(\text{waddr}, \text{length}(bl))) \\
 &\quad \quad \wedge \\
 &\quad \text{unchanged_memory_write_invariant?}(pm, \text{states}, \text{address_block}(\text{waddr}, \text{length}(bl))) \wedge \\
 &\quad \text{changed_memory_invariant?}(pm, \text{states}, \text{address_block}(\text{waddr}, \text{length}(bl))) \wedge \\
 &\quad \text{transformers_ok?}(\text{states}, \\
 &\quad \quad \quad \text{(memory_read_transformers}(pm, \text{address_block}(\text{waddr}, \text{length}(bl))) \cup m \\
 &\quad \quad \wedge \\
 &\quad \quad \text{transformers_ok?}(\text{states}, \\
 &\quad \quad \quad \text{(memory_write_transformers}(pm, \text{address_block}(\text{waddr}, \text{length}(bl))) \cup \\
 &\quad \quad \wedge \\
 &\quad \quad \text{side_effect_content_unchanged}(\text{address_block}(\text{waddr}, \text{length}(bl)), \text{states}, \\
 &\quad \quad \quad \text{memory_read_side_effect}(pm)) \\
 &\quad \quad \wedge \\
 &\quad \quad \text{side_effect_content_unchanged}(\text{address_block}(\text{waddr}, \text{length}(bl)), \text{states}, \\
 &\quad \quad \quad \text{memory_write_side_effect}(pm)) \\
 &\quad \quad \wedge \text{states}(s) \\
 &\quad \supset \\
 &\quad \text{OK?}[\text{State}, \text{Unit}](\text{memory_write_list}[\text{State}](pm)(\text{waddr}, bl)(s)) \vee \\
 &\quad \text{Exception?}[\text{State}, \text{Unit}](\text{memory_write_list}[\text{State}](pm)(\text{waddr}, bl)(s))
 \end{aligned}$$

Repeatedly Skolemizing and flattening,

Using lemma memory_write_list_ok,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Installing automatic rewrites from: subset_equal union_subset1 union_subset3 union_left union_right subset_singleton subset_bigger_union_right subset_bigger_union_left union_left union_right transformer_invariant_union_transformers unchanged_memory_write_invariant_transformer_invariant

Rewriting using transformer_invariant_union_transformers, matching in *,

Hiding formulas: -1, 2,

Forward chaining on unchanged_memory_invariant_invariant,

Hiding formulas: -2,

Using lemma transformer_invariant_mono_transformers,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of unchanged_memory_read_list_write_list_TCC1.

Q.E.D.

C.123.43

Memory_Change_2.unchanged_memory_read_list_write_list_TCC2

Terse proof for unchanged_memory_read_list_write_list_TCC2.

unchanged_memory_read_list_write_list_TCC2:

$$\begin{array}{l}
\{1\} \quad \forall (pm: \text{Memory_struct}[\text{State}], \text{states}: \text{PRED}[\text{State}], \text{waddr}: \text{Ad-} \\
\quad \text{dress}, \text{bl}: \text{list}[\text{Byte}], \\
\quad \quad s: \text{State}): \\
\quad \text{unchanged_memory_invariant?}(pm, \text{states}, \\
\quad \quad \quad (\text{memory_read_transformers}(pm, \text{address_block}(\text{waddr}, \text{length}(\text{bl}))) \cup (\text{m} \\
\quad \quad \quad \text{address_block}(\text{waddr}, \text{length}(\text{bl}))) \\
\quad \wedge \\
\quad \text{unchanged_memory_write_invariant?}(pm, \text{states}, \text{address_block}(\text{waddr}, \text{length}(\text{bl}))) \wedge \\
\quad \text{changed_memory_invariant?}(pm, \text{states}, \text{address_block}(\text{waddr}, \text{length}(\text{bl}))) \wedge \\
\quad \quad \text{transformers_ok?}(\text{states}, \\
\quad \quad \quad (\text{memory_read_transformers}(pm, \text{address_block}(\text{waddr}, \text{length}(\text{bl}))) \cup \text{memory_read} \\
\quad \quad \wedge \\
\quad \quad \text{transformers_ok?}(\text{states}, \\
\quad \quad \quad (\text{memory_write_transformers}(pm, \text{address_block}(\text{waddr}, \text{length}(\text{bl}))) \cup \text{memory_w} \\
\quad \quad \wedge \\
\quad \quad \text{side_effect_content_unchanged}(\text{address_block}(\text{waddr}, \text{length}(\text{bl})), \text{states}, \\
\quad \quad \quad \text{memory_read_side_effect}(pm)) \\
\quad \quad \wedge \\
\quad \quad \text{side_effect_content_unchanged}(\text{address_block}(\text{waddr}, \text{length}(\text{bl})), \text{states}, \\
\quad \quad \quad \text{memory_write_side_effect}(pm)) \\
\quad \quad \wedge \text{states}(s) \\
\quad \supset \\
\quad \text{OK?}[\text{State}, \text{list}[\text{Byte}]] \\
\quad \quad (\text{memory_read_list}[\text{State}] \\
\quad \quad \quad (pm)(\text{waddr}, \text{length}[\text{Byte}](\text{bl})) \\
\quad \quad \quad (\text{state}[\text{State}, \text{Unit}](\text{memory_write_list}[\text{State}](pm)(\text{waddr}, \text{bl})(s))))
\end{array}$$

Repeatedly Skolemizing and flattening,

Installing automatic rewrites from: subset_equal union_subset1 union_subset3 union_left union_right
subset_singleton subset_bigger_union_right subset_bigger_union_left union_left union_right trans-
former_invariant_union_transformers has_next_state memory_write_side_effect_super_transformers_memory_write_side_effect

Forward chaining on unchanged_memory_invariant_invariant,

Using lemma memory_read_list_ok,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 2 subgoals:

unchanged_memory_read_list_write_list_TCC2.1:

{-1}	transformers_ok?(states', (memory_read_transformers(pm', address_block(waddr', length(bl')))) ∪ mem
{-2}	transformer_invariant?(states', (memory_read_transformers(pm', address_block(waddr', length(bl'))))
{-3}	unchanged_memory_invariant?(pm', states', (memory_read_transformers(pm', address_block(waddr', length address_block(waddr', length(bl'))
{-4}	unchanged_memory_write_invariant?(pm', states', address_block(waddr', length(bl')))
{-5}	changed_memory_invariant?(pm', states', address_block(waddr', length(bl')))
{-6}	transformers_ok?(states', (memory_write_transformers(pm', address_block(waddr', length(bl')))) ∪ mem
{-7}	side_effect_content_unchanged(address_block(waddr', length(bl')), states', memory_read_side_effect(pm'))
{-8}	side_effect_content_unchanged(address_block(waddr', length(bl')), states', memory_write_side_effect(pm'))
{-9}	states'(s')
{1}	states'(state[State, Unit](memory_write_list[State](pm')(waddr', bl')(s')))
{2}	OK?[State, list[Byte]] (memory_read_list[State] (pm')(waddr', length[Byte](bl')) (state[State, Unit](memory_write_list[State](pm')(waddr', bl')(s'))))

Hiding formulas: 2,

Using lemma unchanged_memory_write_invariant_transformer_invariant,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Using lemma memory_write_list_ok,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 2 subgoals:

unchanged_memory_read_list_write_list_TCC2.1.1:

{-1}	transformers_ok?(states', (memory_write_transformers(pm', address_block(waddr', length(bl')))) ∪ mem
{-2}	side_effect_content_unchanged(address_block(waddr', length(bl')), states', memory_write_side_effect(pm'))
{-3}	states'(s')
{-4}	OK?(memory_write_list(pm')(waddr', bl')(s'))
{-5}	unchanged_memory_write_invariant?(pm', states', address_block(waddr', length(bl')))
{-6}	transformer_invariant?(states', memory_write_transformers(pm', ad- dress_block(waddr', length(bl'))))
{-7}	transformers_ok?(states', (memory_read_transformers(pm', address_block(waddr', length(bl')))) ∪ mem
{-8}	transformer_invariant?(states', (memory_read_transformers(pm', address_block(waddr', length(bl'))))
{-9}	unchanged_memory_invariant?(pm', states', (memory_read_transformers(pm', address_block(waddr', lengt address_block(waddr', length(bl'))
{-10}	changed_memory_invariant?(pm', states', address_block(waddr', length(bl')))
{-11}	side_effect_content_unchanged(address_block(waddr', length(bl')), states', memory_read_side_effect(pm'))
{1}	states'(state[State, Unit](memory_write_list[State](pm')(waddr', bl')(s')))

Using lemma `transformer_invariant_write_list`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 2 subgoals:

`unchanged_memory_read_list_write_list_TCC2.1.1.1`:

{-1}	<code>side_effect_content_unchanged(address_block(waddr', length(bl')), states', memory_write_side_effect(pm'))</code>
{-2}	<code>states'(s')</code>
{-3}	<code>result_pred(states')(expr_2_super(memory_write_list(pm')(waddr', bl')(s'))</code>
{-4}	<code>transformers_ok?(states', (memory_write_transformers(pm', address_block(waddr', length(bl')))) ∪ memory_write</code>
{-5}	<code>OK?(memory_write_list(pm')(waddr', bl')(s'))</code>
{-6}	<code>unchanged_memory_write_invariant?(pm', states', address_block(waddr', length(bl'))</code>
{-7}	<code>transformer_invariant?(states', memory_write_transformers(pm', ad- dress_block(waddr', length(bl'))))</code>
{-8}	<code>transformers_ok?(states', (memory_read_transformers(pm', address_block(waddr', length(bl')))) ∪ memory_read</code>
{-9}	<code>transformer_invariant?(states', (memory_read_transformers(pm', address_block(waddr', length(bl')))) ∪ (memor</code>
{-10}	<code>unchanged_memory_invariant?(pm', states', (memory_read_transformers(pm', address_block(waddr', length(bl')))) ∪ (r address_block(waddr', length(bl'))</code>
{-11}	<code>changed_memory_invariant?(pm', states', address_block(waddr', length(bl'))</code>
{-12}	<code>side_effect_content_unchanged(address_block(waddr', length(bl')), states', memory_read_side_effect(pm'))</code>
{1}	<code>states'(state[State, Unit](memory_write_list[State](pm')(waddr', bl')(s'))</code>

Rewriting using `expr_result_pred_next_state`, matching in `*`,

This completes the proof of `unchanged_memory_read_list_write_list_TCC2.1.1.1`.

C Proof scripts

unchanged_memory_read_list_write_list_TCC2.1.1.2:

{-1}	side_effect_content_unchanged(address_block(waddr', length(bl')), states', memory_write_side_effect(pm'))
{-2}	states'(s')
{-3}	transformers_ok?(states', (memory_write_transformers(pm', address_block(waddr', length(bl'))) ∪ me
{-4}	OK?(memory_write_list(pm')(waddr', bl')(s'))
{-5}	unchanged_memory_write_invariant?(pm', states', address_block(waddr', length(bl')))
{-6}	transformer_invariant?(states', memory_write_transformers(pm', ad- dress_block(waddr', length(bl'))))
{-7}	transformers_ok?(states', (memory_read_transformers(pm', address_block(waddr', length(bl'))) ∪ me
{-8}	transformer_invariant?(states', (memory_read_transformers(pm', address_block(waddr', length(bl'))
{-9}	unchanged_memory_invariant?(pm', states', (memory_read_transformers(pm', address_block(waddr', lengt address_block(waddr', length(bl'))
{-10}	changed_memory_invariant?(pm', states', address_block(waddr', length(bl')))
{-11}	side_effect_content_unchanged(address_block(waddr', length(bl')), states', memory_read_side_effect(pm'))
{1}	transformer_invariant?(states', (memory_write_transformers(pm', address_block(waddr', length(bl'))
{2}	states'(state[State, Unit](memory_write_list[State](pm')(waddr', bl')(s')))

Rewriting using add_as_union, matching in *,

Rewriting using transformer_invariant_union_transformers, matching in *,

Keeping (-8 1) and hiding *,

Using lemma transformer_invariant_mono_transformers,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of unchanged_memory_read_list_write_list_TCC2.1.1.2.

unchanged_memory_read_list_write_list_TCC2.1.2:

{-1}	transformers_ok?(states', (memory_write_transformers(pm', address_block(waddr', length(bl')))) ∪ memory_write_
{-2}	side_effect_content_unchanged(address_block(waddr', length(bl')), states', memory_write_side_effect(pm'))
{-3}	states'(s')
{-4}	unchanged_memory_write_invariant?(pm', states', address_block(waddr', length(bl')))
{-5}	transformer_invariant?(states', memory_write_transformers(pm', ad- dress_block(waddr', length(bl'))))
{-6}	transformers_ok?(states', (memory_read_transformers(pm', address_block(waddr', length(bl')))) ∪ memory_read_
{-7}	transformer_invariant?(states', (memory_read_transformers(pm', address_block(waddr', length(bl')))) ∪ (memor
{-8}	unchanged_memory_invariant?(pm', states', (memory_read_transformers(pm', address_block(waddr', length(bl')))) ∪ (memor
	address_block(waddr', length(bl'))
{-9}	changed_memory_invariant?(pm', states', address_block(waddr', length(bl')))
{-10}	side_effect_content_unchanged(address_block(waddr', length(bl')), states', memory_read_side_effect(pm'))
{1}	transformer_invariant?(states', (memory_write_transformers(pm', address_block(waddr', length(bl')))) ∪ memor
{2}	states'(state[State, Unit](memory_write_list[State](pm')(waddr', bl')(s')))

Keeping (-5 -7 1) and hiding *,

Rewriting using transformer_invariant_union_transformers, matching in *,

Hiding formulas: -1, 2,

Using lemma transformer_invariant_mono_transformers,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of unchanged_memory_read_list_write_list_TCC2.1.2.

C Proof scripts

unchanged_memory_read_list_write_list_TCC2.2:

{-1}	transformers_ok?(states', (memory_read_transformers(pm', address_block(waddr', length(bl')))) ∪ mem
{-2}	transformer_invariant?(states', (memory_read_transformers(pm', address_block(waddr', length(bl'))))
{-3}	unchanged_memory_invariant?(pm', states', (memory_read_transformers(pm', address_block(waddr', length address_block(waddr', length(bl'))
{-4}	unchanged_memory_write_invariant?(pm', states', address_block(waddr', length(bl')))
{-5}	changed_memory_invariant?(pm', states', address_block(waddr', length(bl')))
{-6}	transformers_ok?(states', (memory_write_transformers(pm', address_block(waddr', length(bl')))) ∪ mem
{-7}	side_effect_content_unchanged(address_block(waddr', length(bl')), states', memory_read_side_effect(pm'))
{-8}	side_effect_content_unchanged(address_block(waddr', length(bl')), states', memory_write_side_effect(pm'))
{-9}	states'(s')
{1}	transformer_invariant?(states', (memory_read_transformers(pm', address_block(waddr', length [Byte
{2}	OK?[State, list[Byte]] (memory_read_list[State] (pm')(waddr', length[Byte](bl')) (state[State, Unit](memory_write_list[State](pm')(waddr', bl')(s'))))

Keeping (-2 1) and hiding *,

Rewriting using transformer_invariant_union_transformers, matching in *,

we get 2 subgoals:

unchanged_memory_read_list_write_list_TCC2.2.1:

{-1}	transformer_invariant?(states', (memory_read_transformers(pm', address_block(waddr', length(bl'))))
{1}	transformer_invariant?(states', memory_read_transformers(pm', address_block(waddr', length[Byte](bl'))))
{2}	transformer_invariant?(states', (memory_read_transformers(pm', address_block(waddr', length [Byte

Hiding formulas: 2,

Using lemma transformer_invariant_mono_transformers,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of unchanged_memory_read_list_write_list_TCC2.2.1.

unchanged_memory_read_list_write_list_TCC2.2.2:

{-1}	transformer_invariant?(states', (memory_read_transformers(pm', address_block(waddr', length(bl'))))
{1}	transformer_invariant?(states', memory_read_side_effect_super_transformers(pm', ad- dress_block (waddr', length[Byte](b
{2}	transformer_invariant?(states', (memory_read_transformers(pm', address_block(waddr', length [Byte

Hiding formulas: 2,

Using lemma transformer_invariant_mono_transformers,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of unchanged_memory_read_list_write_list_TCC2.2.2.

Q.E.D.

C.123.44 Memory_Change_2.unchanged_memory_read_list_write_list

Terse proof for unchanged_memory_read_list_write_list.

unchanged_memory_read_list_write_list:

$$\begin{array}{l}
 \{1\} \quad \forall (pm: \text{Memory_struct}[\text{State}], \text{states}: \text{PRED}[\text{State}], \text{waddr}: \text{Address}, \\
 \quad \text{dress}, \text{bl}: \text{list}[\text{Byte}], \\
 \quad \quad s: \text{State}): \\
 \quad \text{unchanged_memory_invariant?}(pm, \text{states}, \\
 \quad \quad \quad (\text{memory_read_transformers}(pm, \text{address_block}(\text{waddr}, \text{length}(\text{bl}))) \cup (\text{memory_read_transformers}(pm, \text{address_block}(\text{waddr}, \text{length}(\text{bl})))) \\
 \quad \quad \quad \wedge \\
 \quad \quad \text{unchanged_memory_write_invariant?}(pm, \text{states}, \text{address_block}(\text{waddr}, \text{length}(\text{bl}))) \wedge \\
 \quad \quad \text{changed_memory_invariant?}(pm, \text{states}, \text{address_block}(\text{waddr}, \text{length}(\text{bl}))) \wedge \\
 \quad \quad \quad \text{transformers_ok?}(\text{states}, \\
 \quad \quad \quad \quad (\text{memory_read_transformers}(pm, \text{address_block}(\text{waddr}, \text{length}(\text{bl}))) \cup \text{memory_read_transformers}(pm, \text{address_block}(\text{waddr}, \text{length}(\text{bl})))) \\
 \quad \quad \quad \wedge \\
 \quad \quad \quad \text{transformers_ok?}(\text{states}, \\
 \quad \quad \quad \quad (\text{memory_write_transformers}(pm, \text{address_block}(\text{waddr}, \text{length}(\text{bl}))) \cup \text{memory_write_transformers}(pm, \text{address_block}(\text{waddr}, \text{length}(\text{bl})))) \\
 \quad \quad \quad \wedge \\
 \quad \quad \quad \text{side_effect_content_unchanged}(\text{address_block}(\text{waddr}, \text{length}(\text{bl})), \text{states}, \\
 \quad \quad \quad \quad \text{memory_read_side_effect}(pm)) \\
 \quad \quad \quad \wedge \\
 \quad \quad \quad \text{side_effect_content_unchanged}(\text{address_block}(\text{waddr}, \text{length}(\text{bl})), \text{states}, \\
 \quad \quad \quad \quad \text{memory_write_side_effect}(pm)) \\
 \quad \quad \quad \wedge \text{states}(s) \\
 \quad \quad \supset \\
 \quad \quad \text{data}(\text{memory_read_list}(pm)(\text{waddr}, \text{length}(\text{bl})) \\
 \quad \quad \quad (\text{state}(\text{memory_write_list}(pm)(\text{waddr}, \text{bl})(s)))) \\
 \quad \quad = \text{bl}
 \end{array}$$

Repeatedly Skolemizing and flattening,

Using lemma unchanged_memory_read_list_write_list_TCC2,

Simplifying, rewriting, and recording with decision procedures,

Installing automatic rewrites from: address_block address_block_subset_1 ##! length subset_equal union_subset1 union_subset3 union_left union_right subset_singleton subset_bigger_union_right subset_bigger_union_left union_left union_right memory_read_transformers_memory_read memory_read_transformers_mono memory_write_transformers_memory_write memory_write_transformers_mono transformer_invariant_mono_transformers expr_transformer_invariant_next_ok ok_result has_next_state state_expr_2_super ok_expr_2_super memory_write_side_effect_super transformers_memory_write_side_effect memory_read_side_effect_super transformers_memory_read_side_effect

Using lemma expr_transformers_ok_ok,

Simplifying, rewriting, and recording with decision procedures,

Forward chaining on unchanged_memory_invariant_invariant,

Using lemma expr_transformer_invariant_next_ok,

Simplifying, rewriting, and recording with decision procedures,

C Proof scripts

Expanding the definition of `memory_read_list`,

Expanding the definition of `memory_write_list`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Using lemma `side_effect_content_unchanged_content`,

Simplifying, rewriting, and recording with decision procedures,

Replacing using formula -1,

Using lemma `unchanged_memory_read_list_write_list_nse`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 4 subgoals:

unchanged_memory_read_list_write_list.1:

```

{-1}  unchanged_memory_write_invariant?(pm', states', address_block(waddr', length(bl')))
{-2}  changed_memory_invariant?(pm', states', address_block(waddr', length(bl')))
{-3}  states'(state(memory_write_side_effect(pm')(waddr', bl', FALSE)(s')))
{-4}  data(memory_read_list_nse(pm')(waddr', length(bl'))
      (state(memory_write_list_nse(pm')(waddr', bl')
      (state(memory_write_side_effect(pm')
      (waddr', bl', FALSE)
      (s'))))))
      = bl'
{-5}  data(memory_write_side_effect(pm')(waddr', bl', FALSE)(s')) = bl'
{-6}  transformer_invariant?(states',
      (memory_read_transformers(pm', address_block(waddr', length(bl'))) ∪ (memor
{-7}  OK?(memory_write_side_effect(pm')(waddr', bl', FALSE)(s'))
{-8}  OK?(memory_read_list_nse(pm')(waddr', length[Byte](bl'))
      (state[State, Unit]
      (memory_write_list_nse(pm')(waddr', bl')
      (state(memory_write_side_effect(pm')
      (waddr', bl', FALSE)
      (s'))))))
{-9}  OK?[State, list[Byte]]
      (memory_read_side_effect(pm')
      (waddr',
      data(memory_read_list_nse(pm')(waddr', length[Byte](bl'))
      (state[State, Unit]
      (memory_write_list_nse(pm')
      (waddr', bl')
      (state
      (memory_write_side_effect
      (pm')
      (waddr', bl', FALSE)
      (s'))))))),
      FALSE)
      (state(memory_read_list_nse(pm')(waddr', length[Byte](bl'))
      (state[State, Unit]
      (memory_write_list_nse(pm')
      (waddr', bl')
      (state
      (memory_write_side_effect
      (pm')
      (waddr', bl', FALSE)
      (s'))))))))
{-10} unchanged_memory_invariant?(pm', states',
      (memory_read_transformers(pm', address_block(waddr', length(bl'))) ∪ (r
      address_block(waddr', length(bl')))
{-11} transformers_ok?(states',
      (memory_read_transformers(pm', address_block(waddr', length(bl'))) ∪ memory_read_s
{-12} transformers_ok?(states',
      (memory_write_transformers(pm', address_block(waddr', length(bl'))) ∪ memory_writ
{-13} side_effect_content_unchanged(address_block(waddr', length(bl')), states',
      memory_read_side_effect(pm'))
{-14} side_effect_content_unchanged(address_block(waddr', length(bl')), states',
      memory_write_side_effect(pm'))
{-15} states'(s')
-----
{1}  data(memory_read_side_effect(pm')
      (waddr',
      data(memory_read_list_nse(pm')(waddr', length[Byte](bl'))
      (state[State, Unit]
      (memory_write_list_nse(pm')
      (waddr', bl')
      (state
      (memory_write_side_effect
      (pm')
      (waddr', bl', FALSE)
      (s'))))))
      2547

```

C Proof scripts

Replacing using formula -4,

Using lemma `side_effect_content_unchanged_content`,

Simplifying, rewriting, and recording with decision procedures,

Hiding formulas: -4, -5, -9, -13, -14, 2,

Using lemma `memory_read_list_nse_next_length`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 3 subgoals:

`unchanged_memory_read_list_write_list.1.1:`

{-1}	<code>unchanged_memory_write_invariant?(pm', states', address_block(waddr', length(bl'))</code>
{-2}	<code>changed_memory_invariant?(pm', states', address_block(waddr', length(bl'))</code>
{-3}	<code>states'(state(memory_write_side_effect(pm')(waddr', bl', FALSE)(s'))</code>
{-4}	<code>transformer_invariant?(states',</code> $\text{(memory_read_transformers(pm', address_block(waddr', length(bl'))$
{-5}	<code>OK?(memory_write_side_effect(pm')(waddr', bl', FALSE)(s'))</code>
{-6}	<code>OK?(memory_read_list_nse(pm')(waddr', length[Byte](bl'))</code> $\text{(state[State, Unit]$ $\text{(memory_write_list_nse(pm')(waddr', bl')$ $\text{(state(memory_write_side_effect(pm')$ $\text{(waddr', bl', FALSE)$ $\text{(s'))))))))$
{-7}	<code>unchanged_memory_invariant?(pm', states',</code> $\text{(memory_read_transformers(pm', address_block(waddr', length(bl'))$ $\text{address_block(waddr', length(bl'))$
{-8}	<code>transformers_ok?(states',</code> $\text{(memory_read_transformers(pm', address_block(waddr', length(bl'))} \cup \text{mem}$
{-9}	<code>transformers_ok?(states',</code> $\text{(memory_write_transformers(pm', address_block(waddr', length(bl'))} \cup \text{mem}$
{-10}	<code>states'(s')</code>
{1}	<code>states'(state[State, Unit]</code> $\text{(memory_write_list_nse(pm')(waddr', bl')$ $\text{(state(memory_write_side_effect(pm')$ $\text{(waddr', bl', FALSE)(s'))))$
{2}	<code>states'(state(memory_read_list_nse(pm')(waddr', length[Byte](bl'))</code> $\text{(state[State, Unit]$ $\text{(memory_write_list_nse(pm')$ $\text{(waddr', bl')$ $\text{(state$ $\text{(memory_write_list_nse(pm')$ $\text{(pm')$ $\text{(waddr', bl',$ $\text{(s'))))))))$

Hiding formulas: 2,

Using lemma `transformer_invariant_write_list_nse`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 2 subgoals:

unchanged_memory_read_list_write_list.1.1.1:

{-1}	states'(state(memory_write_side_effect(pm')(waddr', bl', FALSE)(s')))
{-2}	result_pred(states') (expr_2_super(memory_write_list_nse(pm')(waddr', bl') (state(memory_write_side_effect(pm') (waddr', bl', FALSE)(s')))))
{-3}	unchanged_memory_write_invariant?(pm', states', address_block(waddr', length(bl')))
{-4}	changed_memory_invariant?(pm', states', address_block(waddr', length(bl')))
{-5}	transformer_invariant?(states', (memory_read_transformers(pm', address_block(waddr', length(bl'))) ∪ (memor
{-6}	OK?(memory_write_side_effect(pm')(waddr', bl', FALSE)(s'))
{-7}	OK?(memory_read_list_nse(pm')(waddr', length[Byte](bl')) (state[State, Unit] (memory_write_list_nse(pm')(waddr', bl') (state(memory_write_side_effect(pm') (waddr', bl', FALSE) (s'))))))
{-8}	unchanged_memory_invariant?(pm', states', (memory_read_transformers(pm', address_block(waddr', length(bl'))) ∪ (n address_block(waddr', length(bl')))
{-9}	transformers_ok?(states', (memory_read_transformers(pm', address_block(waddr', length(bl'))) ∪ memory_read_s
{-10}	transformers_ok?(states', (memory_write_transformers(pm', address_block(waddr', length(bl'))) ∪ memory_write
{-11}	states'(s')
{1}	states'(state[State, Unit] (memory_write_list_nse(pm')(waddr', bl') (state(memory_write_side_effect(pm') (waddr', bl', FALSE)(s')))))

Using lemma `expr_result_pred_next_state[State, Unit]`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Hiding formulas: -1, -7, 2,

Using lemma `memory_write_list_ok_nse`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 2 subgoals:

unchanged_memory_read_list_write_list.1.1.1.1:

{-1}	states'(state(memory_write_side_effect(pm')(waddr', bl', FALSE)(s'))
{-2}	unchanged_memory_write_invariant?(pm', states', address_block(waddr', length(bl')))
{-3}	changed_memory_invariant?(pm', states', address_block(waddr', length(bl')))
{-4}	transformer_invariant?(states', (memory_read_transformers(pm', address_block(waddr', length(bl'))))
{-5}	OK?(memory_write_side_effect(pm')(waddr', bl', FALSE)(s'))
{-6}	unchanged_memory_invariant?(pm', states', (memory_read_transformers(pm', address_block(waddr', length(bl')) address_block(waddr', length(bl'))))
{-7}	transformers_ok?(states', (memory_read_transformers(pm', address_block(waddr', length(bl')))) ∪ mem
{-8}	transformers_ok?(states', (memory_write_transformers(pm', address_block(waddr', length(bl')))) ∪ mem
{-9}	states'(s')
{1}	transformers_ok?(states', memory_write_transformers(pm', address_block(waddr', length(bl'))))
{2}	has_next_state(memory_write_list_nse(pm')(waddr', bl') (state(memory_write_side_effect(pm') (waddr', bl', FALSE)(s'))))

Keeping (-8 1) and hiding *,

Using lemma transformers_ok_mono_transformers,

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of unchanged_memory_read_list_write_list.1.1.1.1.

unchanged_memory_read_list_write_list.1.1.1.2:

{-1}	states'(state(memory_write_side_effect(pm')(waddr', bl', FALSE)(s'))
{-2}	unchanged_memory_write_invariant?(pm', states', address_block(waddr', length(bl')))
{-3}	changed_memory_invariant?(pm', states', address_block(waddr', length(bl')))
{-4}	transformer_invariant?(states', (memory_read_transformers(pm', address_block(waddr', length(bl'))))
{-5}	OK?(memory_write_side_effect(pm')(waddr', bl', FALSE)(s'))
{-6}	unchanged_memory_invariant?(pm', states', (memory_read_transformers(pm', address_block(waddr', length(bl')) address_block(waddr', length(bl'))))
{-7}	transformers_ok?(states', (memory_read_transformers(pm', address_block(waddr', length(bl')))) ∪ mem
{-8}	transformers_ok?(states', (memory_write_transformers(pm', address_block(waddr', length(bl')))) ∪ mem
{-9}	states'(s')
{1}	transformer_invariant?(states', memory_write_transformers(pm', ad- dress_block(waddr', length(bl'))))
{2}	has_next_state(memory_write_list_nse(pm')(waddr', bl') (state(memory_write_side_effect(pm') (waddr', bl', FALSE)(s'))))

Forward chaining on unchanged_memory_write_invariant_transformer_invariant,

This completes the proof of unchanged_memory_read_list_write_list.1.1.1.2.

unchanged_memory_read_list_write_list.1.1.2:

{-1}	states'(state(memory_write_side_effect(pm')(waddr', bl', FALSE)(s'))
{-2}	unchanged_memory_write_invariant?(pm', states', address_block(waddr', length(bl')))
{-3}	changed_memory_invariant?(pm', states', address_block(waddr', length(bl')))
{-4}	transformer_invariant?(states', <div style="margin-left: 150px;">(memory_read_transformers(pm', address_block(waddr', length(bl')) ∪ (memor</div>
{-5}	OK?(memory_write_side_effect(pm')(waddr', bl', FALSE)(s'))
{-6}	OK?(memory_read_list_nse(pm')(waddr', length[Byte](bl')) <div style="margin-left: 150px;">(state[State, Unit] <div style="margin-left: 150px;">(memory_write_list_nse(pm')(waddr', bl') <div style="margin-left: 150px;">(state(memory_write_side_effect(pm') <div style="margin-left: 150px;">(waddr', bl', FALSE) <div style="margin-left: 150px;">(s'))))))))</div> </div> </div> </div> </div>
{-7}	unchanged_memory_invariant?(pm', states', <div style="margin-left: 150px;">(memory_read_transformers(pm', address_block(waddr', length(bl')) ∪ (n <div style="margin-left: 150px;">address_block(waddr', length(bl'))</div> </div>
{-8}	transformers_ok?(states', <div style="margin-left: 150px;">(memory_read_transformers(pm', address_block(waddr', length(bl')) ∪ memory_read_s</div>
{-9}	transformers_ok?(states', <div style="margin-left: 150px;">(memory_write_transformers(pm', address_block(waddr', length(bl')) ∪ memory_write</div>
{-10}	states'(s')
{1}	transformer_invariant?(states', <div style="margin-left: 150px;">memory_write_transformers(pm', <div style="margin-left: 150px;">ad- <div style="margin-left: 150px;">dress_block(waddr', length(bl'))</div> </div> </div>
{2}	states'(state[State, Unit] <div style="margin-left: 150px;">(memory_write_list_nse(pm')(waddr', bl') <div style="margin-left: 150px;">(state(memory_write_side_effect(pm') <div style="margin-left: 150px;">(waddr', bl', FALSE)(s'))))))</div> </div> </div>

Forward chaining on unchanged_memory_write_invariant_transformer_invariant,

This completes the proof of unchanged_memory_read_list_write_list.1.1.2.

unchanged_memory_read_list_write_list.1.2:

{-1}	unchanged_memory_write_invariant?(pm', states', address_block(waddr', length(bl')))
{-2}	changed_memory_invariant?(pm', states', address_block(waddr', length(bl')))
{-3}	states'(state(memory_write_side_effect(pm')(waddr', bl', FALSE)(s')))
{-4}	transformer_invariant?(states', (memory_read_transformers(pm', address_block(waddr', length(bl'))))
{-5}	OK?(memory_write_side_effect(pm')(waddr', bl', FALSE)(s'))
{-6}	OK?(memory_read_list_nse(pm')(waddr', length[Byte](bl')) (state[State, Unit] (memory_write_list_nse(pm')(waddr', bl') (state(memory_write_side_effect(pm')(waddr', bl', FALSE)(s')))))))
{-7}	unchanged_memory_invariant?(pm', states', (memory_read_transformers(pm', address_block(waddr', length(bl'))))
{-8}	transformers_ok?(states', (memory_read_transformers(pm', address_block(waddr', length(bl')))) ∪ me
{-9}	transformers_ok?(states', (memory_write_transformers(pm', address_block(waddr', length(bl')))) ∪ me
{-10}	states'(s')
{1}	transformers_ok?(states', memory_read_transformers(pm', address_block(waddr', length[Byte](bl')))
{2}	states'(state(memory_read_list_nse(pm')(waddr', length[Byte](bl')) (state[State, Unit] (memory_write_list_nse(pm') (waddr', bl') (state (memory_write_side_effect(pm')(waddr', bl', (s'))))))))

Keeping (-8 1) and hiding *,

Using lemma transformers_ok_mono_transformers,

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of unchanged_memory_read_list_write_list.1.2.

unchanged_memory_read_list_write_list.1.3:

{-1}	unchanged_memory_write_invariant?(pm', states', address_block(waddr', length(bl')))
{-2}	changed_memory_invariant?(pm', states', address_block(waddr', length(bl')))
{-3}	states'(state(memory_write_side_effect(pm')(waddr', bl', FALSE)(s')))
{-4}	transformer_invariant?(states', <div style="margin-left: 150px;">(memory_read_transformers(pm', address_block(waddr', length(bl'))) ∪ (memor</div>
{-5}	OK?(memory_write_side_effect(pm')(waddr', bl', FALSE)(s'))
{-6}	OK?(memory_read_list_nse(pm')(waddr', length[Byte](bl')) <div style="margin-left: 150px;">(state[State, Unit] <div style="margin-left: 150px;">(memory_write_list_nse(pm')(waddr', bl') <div style="margin-left: 150px;">(state(memory_write_side_effect(pm') <div style="margin-left: 150px;">(waddr', bl', FALSE) <div style="margin-left: 150px;">(s'))))))))</div> </div> </div> </div> </div>
{-7}	unchanged_memory_invariant?(pm', states', <div style="margin-left: 150px;">(memory_read_transformers(pm', address_block(waddr', length(bl'))) ∪ (n <div style="margin-left: 150px;">address_block(waddr', length(bl')))</div> </div>
{-8}	transformers_ok?(states', <div style="margin-left: 150px;">(memory_read_transformers(pm', address_block(waddr', length(bl')))) ∪ memory_read_s</div>
{-9}	transformers_ok?(states', <div style="margin-left: 150px;">(memory_write_transformers(pm', address_block(waddr', length(bl')))) ∪ memory_write</div>
{-10}	states'(s')
{1}	transformer_invariant?(states', <div style="margin-left: 150px;">memory_read_transformers(pm', <div style="margin-left: 150px;">address_block(waddr', <div style="margin-left: 150px;">length[Byte](bl'))))</div> </div> </div>
{2}	states'(state(memory_read_list_nse(pm')(waddr', length[Byte](bl')) <div style="margin-left: 150px;">(state[State, Unit] <div style="margin-left: 150px;">(memory_write_list_nse(pm') <div style="margin-left: 150px;">(waddr', bl') <div style="margin-left: 150px;">(state <div style="margin-left: 150px;">(memory_write_side_effect <div style="margin-left: 150px;">(pm') <div style="margin-left: 150px;">(waddr', bl', FALSE) <div style="margin-left: 150px;">(s'))))))))))))</div> </div> </div> </div> </div> </div> </div></div>

Keeping (-4 1) and hiding *,

Using lemma transformer_invariant_mono_transformers,

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of unchanged_memory_read_list_write_list.1.3.

`unchanged_memory_read_list_write_list.2:`

```

{-1}  unchanged_memory_write_invariant?(pm', states', address_block(waddr', length(bl')))
{-2}  changed_memory_invariant?(pm', states', address_block(waddr', length(bl')))
{-3}  states'(state(memory_write_side_effect(pm')(waddr', bl', FALSE)(s')))
{-4}  data(memory_write_side_effect(pm')(waddr', bl', FALSE)(s')) = bl'
{-5}  transformer_invariant?(states',
      (memory_read_transformers(pm', address_block(waddr', length(bl')))
      (memory_write_transformers(pm', address_block(waddr', length(bl')))
      (memory_read_transformers(pm', address_block(waddr', length(bl')))
      (memory_write_transformers(pm', address_block(waddr', length(bl')))))
      (memory_write_side_effect(pm')(waddr', bl', FALSE)(s')))
      (memory_read_side_effect(pm')(waddr', length[Byte](bl')))
      (state[State, Unit]
      (memory_write_list_nse(pm')(waddr', bl')
      (state(memory_write_side_effect(pm')(waddr', bl', FALSE)(s')))))
      (state[State, list[Byte]]
      (memory_read_side_effect(pm')
      (waddr',
      data(memory_read_list_nse(pm')(waddr', length[Byte](bl'))
      (state[State, Unit]
      (memory_write_list_nse(pm')
      (waddr', bl')
      (state
      (memory_write_side_effect(pm')
      (pm')
      (waddr', bl')
      (s'))))))),
      FALSE)
      (state(memory_read_list_nse(pm')(waddr', length[Byte](bl')))
      (state[State, Unit]
      (memory_write_list_nse(pm')
      (waddr', bl')
      (state
      (memory_write_side_effect(pm')
      (pm')
      (waddr', bl')
      (s')))))))))
{-9}  unchanged_memory_invariant?(pm', states',
      (memory_read_transformers(pm', address_block(waddr', length(bl')))
      (memory_write_transformers(pm', address_block(waddr', length(bl')))
      (memory_read_transformers(pm', address_block(waddr', length(bl')))
      (memory_write_transformers(pm', address_block(waddr', length(bl')))))
      (memory_write_side_effect(pm')(waddr', bl', FALSE)(s')))
      (memory_read_side_effect(pm')(waddr', length[Byte](bl')))
      (state[State, Unit]
      (memory_write_list_nse(pm')
      (waddr', bl')
      (state
      (memory_write_side_effect(pm')
      (pm')
      (waddr', bl')
      (s')))))))))
{-10} transformers_ok?(states',
      (memory_read_transformers(pm', address_block(waddr', length(bl')))
      (memory_write_transformers(pm', address_block(waddr', length(bl')))))
      (memory_read_transformers(pm', address_block(waddr', length(bl')))
      (memory_write_transformers(pm', address_block(waddr', length(bl')))))
      (memory_read_side_effect(pm')(waddr', length[Byte](bl')))
      (memory_write_side_effect(pm')(waddr', length[Byte](bl')))
      (state[State, Unit]
      (memory_write_list_nse(pm')
      (waddr', bl')
      (state
      (memory_write_side_effect(pm')
      (pm')
      (waddr', bl')
      (s'))))))))
{-11} transformers_ok?(states',
      (memory_write_transformers(pm', address_block(waddr', length(bl')))
      (memory_read_transformers(pm', address_block(waddr', length(bl')))))
      (memory_write_transformers(pm', address_block(waddr', length(bl')))
      (memory_read_transformers(pm', address_block(waddr', length(bl')))))
      (memory_read_side_effect(pm')(waddr', length[Byte](bl')))
      (memory_write_side_effect(pm')(waddr', length[Byte](bl')))
      (state[State, Unit]
      (memory_write_list_nse(pm')
      (waddr', bl')
      (state
      (memory_write_side_effect(pm')
      (pm')
      (waddr', bl')
      (s'))))))))
{-12} side_effect_content_unchanged(address_block(waddr', length(bl')), states',
      memory_read_side_effect(pm'))
{-13} side_effect_content_unchanged(address_block(waddr', length(bl')), states',
      memory_write_side_effect(pm'))
{-14} states'(s')
-----
{1}   transformers_ok?(states',
      (memory_read_transformers(pm', address_block(waddr', length(bl')))
      (memory_write_transformers(pm', address_block(waddr', length(bl')))))
      (memory_read_transformers(pm', address_block(waddr', length(bl')))
      (memory_write_transformers(pm', address_block(waddr', length(bl')))))
      (memory_read_side_effect(pm')(waddr', length[Byte](bl')))
      (memory_write_side_effect(pm')(waddr', length[Byte](bl')))
      (state[State, Unit]
      (memory_write_list_nse(pm')
      (waddr', bl')
      (state
      (memory_write_side_effect(pm')
      (pm')
      (waddr', bl')
      (s'))))))))
{2}   data(memory_read_side_effect(pm')
      (waddr',
      data(memory_read_list_nse(pm')(waddr', length[Byte](bl'))
      (state[State, Unit]
      (memory_write_list_nse(pm')
      (waddr', bl')
      (state
      (memory_write_side_effect(pm')
      (pm')
      (waddr', bl')
      (s'))))))))
      FALSE)
      (state(memory_read_list_nse(pm')(waddr', length[Byte](bl')))
      (state[State, Unit]

```

Keeping (-11 1) and hiding *,

Using lemma transformers_ok_mono_transformers,

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of unchanged_memory_read_list_write_list.2.

unchanged_memory_read_list_write_list.3:

```

{-1}  unchanged_memory_write_invariant?(pm', states', address_block(waddr', length(bl')))
{-2}  changed_memory_invariant?(pm', states', address_block(waddr', length(bl')))
{-3}  states'(state(memory_write_side_effect(pm')(waddr', bl', FALSE)(s')))
{-4}  data(memory_write_side_effect(pm')(waddr', bl', FALSE)(s')) = bl'
{-5}  transformer_invariant?(states',
      (memory_read_transformers(pm', address_block(waddr', length(bl')))
{-6}  OK?(memory_write_side_effect(pm')(waddr', bl', FALSE)(s'))
{-7}  OK?(memory_read_list_nse(pm')(waddr', length[Byte](bl'))
      (state[State, Unit]
        (memory_write_list_nse(pm')(waddr', bl')
          (state(memory_write_side_effect(pm')(waddr', bl', FALSE)(s'))))))
{-8}  OK?[State, list[Byte]]
      (memory_read_side_effect(pm')
        (waddr',
          data(memory_read_list_nse(pm')(waddr', length[Byte](bl'))
            (state[State, Unit]
              (memory_write_list_nse(pm')
                (waddr', bl')
              (state
                (memory_write_side_effect(pm')
                  (waddr', bl')
                (s'))))))),
          FALSE)
        (state(memory_read_list_nse(pm')(waddr', length[Byte](bl'))
          (state[State, Unit]
            (memory_write_list_nse(pm')
              (waddr', bl')
            (state
              (memory_write_side_effect(pm')
                (waddr', bl')
              (s'))))))))
{-9}  unchanged_memory_invariant?(pm', states',
      (memory_read_transformers(pm', address_block(waddr', length(bl')))
        address_block(waddr', length(bl')))
{-10} transformers_ok?(states',
      (memory_read_transformers(pm', address_block(waddr', length(bl'))) ∪ mem
{-11} transformers_ok?(states',
      (memory_write_transformers(pm', address_block(waddr', length(bl'))) ∪ mem
{-12} side_effect_content_unchanged(address_block(waddr', length(bl')), states',
      memory_read_side_effect(pm'))
{-13} side_effect_content_unchanged(address_block(waddr', length(bl')), states',
      memory_write_side_effect(pm'))
{-14} states'(s')
-----
{1}  transformers_ok?(states',
      memory_read_transformers(pm', address_block(waddr', length(bl'))))
{2}  data(memory_read_side_effect(pm')
      (waddr',
        data(memory_read_list_nse(pm')(waddr', length[Byte](bl'))
          (state[State, Unit]
            (memory_write_list_nse(pm')
              (waddr', bl')
            (state
              (memory_write_side_effect(pm')
                (waddr', bl')
              (s'))))))),
          FALSE)
        (state(memory_read_list_nse(pm')(waddr', length[Byte](bl'))
          (state[State, Unit]
            (memory_write_list_nse(pm')
              (waddr', bl')
            (state
              (memory_write_side_effect(pm')
                (waddr', bl')
              (s'))))))))

```

Keeping (-10 1) and hiding *,

Using lemma transformers_ok_mono_transformers,

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of unchanged_memory_read_list_write_list.3.

unchanged_memory_read_list_write_list.4:

```

{-1}  unchanged_memory_write_invariant?(pm', states', address_block(waddr', length(bl')))
{-2}  changed_memory_invariant?(pm', states', address_block(waddr', length(bl')))
{-3}  states'(state(memory_write_side_effect(pm')(waddr', bl', FALSE)(s')))
{-4}  data(memory_write_side_effect(pm')(waddr', bl', FALSE)(s')) = bl'
{-5}  transformer_invariant?(states',
      (memory_read_transformers(pm', address_block(waddr', length(bl')))
{-6}  OK?(memory_write_side_effect(pm')(waddr', bl', FALSE)(s'))
{-7}  OK?(memory_read_list_nse(pm')(waddr', length[Byte](bl'))
      (state[State, Unit]
        (memory_write_list_nse(pm')(waddr', bl')
          (state(memory_write_side_effect(pm')(waddr', bl', FALSE)(s'))))))

{-8}  OK?[State, list[Byte]]
      (memory_read_side_effect(pm')
        (waddr',
          data(memory_read_list_nse(pm')(waddr', length[Byte](bl'))
              (state[State, Unit]
                (memory_write_list_nse(pm')
                  (waddr', bl')
                    (state
                      (memory_write_side_effect(pm')(waddr', bl')
                        (s'))))))),
          FALSE)
      (state(memory_read_list_nse(pm')(waddr', length[Byte](bl'))
          (state[State, Unit]
            (memory_write_list_nse(pm')
              (waddr', bl')
                (state
                  (memory_write_side_effect(pm')(waddr', bl')
                    (s'))))))))

{-9}  unchanged_memory_invariant?(pm', states',
      (memory_read_transformers(pm', address_block(waddr', length(bl')))
        address_block(waddr', length(bl')))
{-10} transformers_ok?(states',
      (memory_read_transformers(pm', address_block(waddr', length(bl'))) ∪ mem
{-11} transformers_ok?(states',
      (memory_write_transformers(pm', address_block(waddr', length(bl'))) ∪ mem
{-12} side_effect_content_unchanged(address_block(waddr', length(bl')), states',
      memory_read_side_effect(pm'))
{-13} side_effect_content_unchanged(address_block(waddr', length(bl')), states',
      memory_write_side_effect(pm'))
{-14} states'(s')

```

```

{1}  unchanged_memory_invariant?(pm', states',
      memory_read_transformers(pm',
        address_block(waddr', length(bl'))),
      address_block(waddr', length(bl')))
{2}  data(memory_read_side_effect(pm')
      (waddr',
        data(memory_read_list_nse(pm')(waddr', length[Byte](bl'))
          (state[State, Unit]
            (memory_write_list_nse(pm')
              (waddr', bl')
                (state
                  (memory_write_side_effect(pm')(waddr', bl')
                    (s'))))))))

```

Keeping (-9 1) and hiding *,
 Using lemma unchanged_memory_invariant_mono,
 Simplifying, rewriting, and recording with decision procedures,
 This completes the proof of unchanged_memory_read_list_write_list.4.
 Q.E.D.

C.124 Proofs for Memory_Change_3 (memory.pvs)

C.124.1 Mem- Memory_Change_3.expr_unchanged_memory_invariant_unchanged_TCC1

Terse proof for expr_unchanged_memory_invariant_unchanged_TCC1.

expr_unchanged_memory_invariant_unchanged_TCC1:

$$\{1\} \quad \forall (\text{addresses: PRED}[\text{Address}], \text{pm: Memory_struct}[\text{State}], \text{states: PRED}[\text{State}], \\ \text{transformers: PRED}[[\text{State} \rightarrow \text{SuperResult}[\text{State}]]], s: \text{State}, \\ q: [\text{State} \rightarrow \text{ExprResult}[\text{State}, \text{Data}]], a: \text{Address}): \\ \text{OK?}(\text{memory_read}(\text{pm})(a)(s)) \wedge \\ \text{OK?}(q(s)) \wedge \\ \text{addresses}(a) \wedge \\ \text{transformers}(\text{expr_2_super}(q)) \wedge \\ \text{states}(s) \wedge \text{unchanged_memory_invariant?}(\text{pm}, \text{states}, \text{transformers}, \text{addresses}) \\ \supset \text{OK?}[\text{State}, \text{Data}](q(s)) \vee \text{Exception?}[\text{State}, \text{Data}](q(s))$$

Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of expr_unchanged_memory_invariant_unchanged_TCC1.
 Q.E.D.

C.124.2 Memory_Change_3.expr_unchanged_memory_invariant_unchanged

Terse proof for expr_unchanged_memory_invariant_unchanged.

expr_unchanged_memory_invariant_unchanged:

$$\{1\} \quad \forall (\text{addresses: PRED}[\text{Address}], \text{pm: Memory_struct}[\text{State}], \text{states: PRED}[\text{State}], \\ \text{transformers: PRED}[[\text{State} \rightarrow \text{SuperResult}[\text{State}]]], s: \text{State}, \\ q: [\text{State} \rightarrow \text{ExprResult}[\text{State}, \text{Data}]], a: \text{Address}): \\ \text{unchanged_memory_invariant?}(\text{pm}, \text{states}, \text{transformers}, \text{addresses}) \wedge \\ \text{states}(s) \wedge \\ \text{transformers}(\text{expr_2_super}(q)) \wedge \\ \text{addresses}(a) \wedge \\ \text{OK?}(q(s)) \wedge \\ \text{OK?}(\text{memory_read}(\text{pm})(a)(s)) \wedge \text{OK?}(\text{memory_read}(\text{pm})(a)(\text{state}(q(s)))) \\ \supset \text{data}(\text{memory_read}(\text{pm})(a)(\text{state}(q(s)))) = \text{data}(\text{memory_read}(\text{pm})(a)(s))$$

Repeatedly Skolemizing and flattening,
 Using lemma unchanged_memory_invariant_unchanged,
 Installing automatic rewrites from: state_expr_2_super ok_expr_2_super has_next_state
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 This completes the proof of expr_unchanged_memory_invariant_unchanged.

Q.E.D.

C.124.3 Mem- ory_Change_3.expr_unchanged_memory_invariant_next_ok_TCC1

Terse proof for `expr_unchanged_memory_invariant_next_ok_TCC1`.

`expr_unchanged_memory_invariant_next_ok_TCC1`:

$\{1\} \quad \forall (\text{addresses: PRED}[\text{Address}], \text{pm: Memory_struct}[\text{State}], \text{states: PRED}[\text{State}], \\ \text{transformers: PRED}[[\text{State} \rightarrow \text{SuperResult}[\text{State}]]], s: \text{State}, \\ q: [\text{State} \rightarrow \text{ExprResult}[\text{State}, \text{Data}]]): \\ \text{unchanged_memory_invariant?}(\text{pm}, \text{states}, \text{transformers}, \text{addresses}) \wedge \\ \text{states}(s) \wedge \text{transformers}(\text{expr_2_super}(q)) \wedge \text{has_next_state}(q(s)) \\ \supset \text{OK?}[\text{State}, \text{Data}](q(s)) \vee \text{Exception?}[\text{State}, \text{Data}](q(s))$

Repeatedly Skolemizing and flattening,

Expanding the definition of `has_next_state`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `expr_unchanged_memory_invariant_next_ok_TCC1`.

Q.E.D.

C.124.4 Memory_Change_3.expr_unchanged_memory_invariant_next_ok

Terse proof for `expr_unchanged_memory_invariant_next_ok`.

`expr_unchanged_memory_invariant_next_ok`:

$\{1\} \quad \forall (\text{addresses: PRED}[\text{Address}], \text{pm: Memory_struct}[\text{State}], \text{states: PRED}[\text{State}], \\ \text{transformers: PRED}[[\text{State} \rightarrow \text{SuperResult}[\text{State}]]], s: \text{State}, \\ q: [\text{State} \rightarrow \text{ExprResult}[\text{State}, \text{Data}]]): \\ \text{unchanged_memory_invariant?}(\text{pm}, \text{states}, \text{transformers}, \text{addresses}) \wedge \\ \text{states}(s) \wedge \text{transformers}(\text{expr_2_super}(q)) \wedge \text{has_next_state}(q(s)) \\ \supset \text{states}(\text{state}(q(s)))$

Repeatedly Skolemizing and flattening,

Using lemma `unchanged_memory_invariant_next_ok[State]`,

Installing automatic rewrites from: `has_next_state_expr_2_super state_expr_2_super`

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `expr_unchanged_memory_invariant_next_ok`.

Q.E.D.

C.125 Proofs for Memory_Change_4 (memory.pvs)

C.125.1

Memory_Change_4.expr_unchanged_memory_invariant_composition

Terse proof for `expr_unchanged_memory_invariant_composition`.

expr_unchanged_memory_invariant_composition:

$\{1\} \quad \forall (\text{addresses: PRED}[\text{Address}], \text{pm: Memory_struct}[\text{State}], \text{states: PRED}[\text{State}], \\ \text{transformers_1, transformers_2: PRED}[[\text{State} \rightarrow \text{SuperResult}[\text{State}]]], \\ q_1: [\text{State} \rightarrow \text{ExprResult}[\text{State}, \text{Data1}]], q_2: [\text{State} \rightarrow \text{ExprRe-} \\ \text{sult}[\text{State}, \text{Data2}]]): \\ \text{transformers_1}(\text{expr_2_super}(q_1)) \wedge \\ \text{transformers_2}(\text{expr_2_super}(q_2)) \wedge \\ \text{transformers_ok}(\text{states}, \text{memory_read_transformers}(\text{pm}, \text{addresses})) \wedge \\ \text{unchanged_memory_invariant?}(\text{pm}, \text{states}, \text{transformers_1}, \text{addresses}) \wedge \\ \text{unchanged_memory_invariant?}(\text{pm}, \text{states}, \text{transformers_2}, \text{addresses}) \\ \supset \text{unchanged_memory_invariant?}(\text{pm}, \text{states}, \text{singleton}(\text{expr_2_super}(q_1 \#\# q_2)), \text{ad-} \\ \text{resses})$

Expanding the definition of `unchanged_memory_invariant?`,

Repeatedly Skolemizing and flattening,

Case splitting on `transformer_invariant?(states!1, singleton(expr_2_super(q1!1 ## q2!1))`,

we get 2 subgoals:

expr_unchanged_memory_invariant_composition.1:

<pre> {-1} transformer_invariant?(states', singleton(expr_2_super(q1' ## q2')) {-2} transformers'(expr_2_super(q1')) {-3} transformers''(expr_2_super(q2')) {-4} transformers_ok?(states', memory_read_transformers(pm', addresses')) {-5} transformer_invariant?(states', transformers') {-6} ∀ (s: State, q: [State → SuperResult[State]], a: Address): states'(s) ∧ transformers'(q) ∧ addresses'(a) ∧ OK?(q(s)) ∧ OK?(memory_read(pm')(a)(s)) ∧ OK?(memory_read(pm')(a)(state(q(s)))) ⊃ data(memory_read(pm')(a)(state(q(s)))) = data(memory_read(pm')(a)(s)) {-7} transformer_invariant?(states', transformers'') {-8} ∀ (s: State, q: [State → SuperResult[State]], a: Address): states'(s) ∧ transformers''(q) ∧ addresses'(a) ∧ OK?(q(s)) ∧ OK?(memory_read(pm')(a)(s)) ∧ OK?(memory_read(pm')(a)(state(q(s)))) ⊃ data(memory_read(pm')(a)(state(q(s)))) = data(memory_read(pm')(a)(s)) </pre>	<hr style="border: 0.5px solid black;"/> <pre> {1} transformer_invariant?(states', singleton(expr_2_super(q1' ## q2')) ∧ (∀ (s: State, q: [State → SuperResult[State]], a: Address): states'(s) ∧ singleton(expr_2_super(q1' ## q2'))(q) ∧ addresses'(a) ∧ OK?(q(s)) ∧ OK?(memory_read(pm')(a)(s)) ∧ OK?(memory_read(pm')(a)(state(q(s)))) ⊃ data(memory_read(pm')(a)(state(q(s)))) = data(memory_read(pm')(a)(s)) </pre>
--	---

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Repeatedly Skolemizing and flattening,

Expanding the definition of singleton,

Replacing using formula -10,

Instantiating the top quantifier in -6 with the terms: (s!1 expr_2_super(q1!1) a!1),

Installing automatic rewrites from: ok_expr_2_super state_expr_2_super has_next_state has_next_state_expr_2_super ##

Instantiating the top quantifier in -8 with the terms: (state(expr_2_super(q1!1)(s!1)) expr_2_super(q2!1) a!1),

we get 2 subgoals:

expr_unchanged_memory_invariant_composition.1.1:

{-1}	transformer_invariant?(states', singleton(expr_2_super(q'_1 ## q'_2)))
{-2}	transformers'(expr_2_super(q'_1))
{-3}	transformers''(expr_2_super(q'_2))
{-4}	transformers_ok?(states', memory_read_transformers(pm', addresses'))
{-5}	transformer_invariant?(states', transformers')
{-6}	states'(s') \wedge transformers'(expr_2_super(q'_1)) \wedge addresses'(a') \wedge OK?(expr_2_super(q'_1)(s')) \wedge OK?(memory_read(pm')(a')(s')) \wedge OK?(memory_read(pm')(a')(state(expr_2_super(q'_1)(s'))))
	\supset data(memory_read(pm')(a')(state(expr_2_super(q'_1)(s')))) = data(memory_read(pm')(a')(s'))
{-7}	transformer_invariant?(states', transformers'')
{-8}	states'(state(expr_2_super(q'_1)(s'))) \wedge transformers''(expr_2_super(q'_2)) \wedge addresses'(a') \wedge OK?(expr_2_super(q'_2)(state(expr_2_super(q'_1)(s')))) \wedge OK?(memory_read(pm')(a')(state(expr_2_super(q'_1)(s')))) \wedge OK?(memory_read(pm')(a')(state(expr_2_super(q'_2)(state(expr_2_super(q'_1)(s')))))
	\supset data(memory_read(pm')(a')(state(expr_2_super(q'_2)(state(expr_2_super(q'_1)(s'))))) = data(memory_read(pm')(a')(state(expr_2_super(q'_1)(s'))))
{-9}	states'(s')
{-10}	q' = expr_2_super(q'_1 ## q'_2)
{-11}	addresses'(a')
{-12}	OK?(expr_2_super(q'_1 ## q'_2)(s'))
{-13}	OK?(memory_read(pm')(a')(s'))
{-14}	OK?(memory_read(pm')(a')(state(expr_2_super(q'_1 ## q'_2)(s'))))
{1}	data(memory_read(pm')(a')(state(expr_2_super(q'_1 ## q'_2)(s')))) = data(memory_read(pm')(a')(s'))

Expanding the definition of ##,

Expanding the definition of ##,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 2 subgoals:

C Proof scripts

expr_unchanged_memory_invariant_composition.1.1.1:

```

{-1} transformer_invariant?(states',
                               singleton(expr_2_super(λ (s: State):
                                                    CASES q'_1(s) OF
                                                       OK(state, data): q'_2(state),
                                                       Excep-
tion(ex_type, state):
                                                    Excep-
tion(ex_type, state),
                                                    Fatal: Fatal,
                                                    Hang: Hang
                                                    ENDCASES)))
{-2} transformers'(expr_2_super(q'_1))
{-3} transformers''(expr_2_super(q'_2))
{-4} transformers_ok?(states', memory_read_transformers(pm', addresses'))
{-5} transformer_invariant?(states', transformers')
{-6} states'(s')
{-7} addresses'(a')
{-8} OK?(memory_read(pm')(a')(s'))
{-9} data(memory_read(pm')(a')(state(q'_1(s')))) =
      data(memory_read(pm')(a')(s'))
{-10} transformer_invariant?(states', transformers'')
{-11} q' =
      expr_2_super(λ (s: State):
                  CASES q'_1(s) OF
                     OK(state, data): q'_2(state),
                     Exception(ex_type, state): Exception(ex_type, state),
                     Fatal: Fatal,
                     Hang: Hang
                     ENDCASES)
{-12} OK?(q'_1(s'))
{-13} OK?(q'_2(state(q'_1(s'))))
{-14} OK?(memory_read(pm')(a')(state(q'_2(state(q'_1(s'))))))
-----
{1} states'(state(q'_1(s')))
{2} data(memory_read(pm')(a')(state(q'_2(state(q'_1(s')))))) =
      data(memory_read(pm')(a')(s'))

```

Rewriting using `expr_transformer_invariant_next_ok`, matching in * where `s` gets `s!1`, `q` gets `q!1!1`, `states` gets `states!1`, `transformers` gets `transformers!1`,

This completes the proof of `expr_unchanged_memory_invariant_composition.1.1.1`.

expr_unchanged_memory_invariant_composition.1.1.2:

```

{-1}  transformer_invariant?(states',
      singleton(expr_2_super(λ (s: State):
        CASES q'_1(s) OF
          OK(state, data): q'_2(state),
          Except-
            Except-
              Fatal: Fatal,
              Hang: Hang
            ENDCASES)))
      tion(ex_type, state):
      tion(ex_type, state),
{-2}  transformers'(expr_2_super(q'_1))
{-3}  transformers''(expr_2_super(q'_2))
{-4}  transformers_ok?(states', memory_read_transformers(pm', addresses'))
{-5}  transformer_invariant?(states', transformers')
{-6}  states'(s')
{-7}  addresses'(a')
{-8}  OK?(memory_read(pm')(a')(s'))
{-9}  transformer_invariant?(states', transformers'')
{-10} q' =
      expr_2_super(λ (s: State):
        CASES q'_1(s) OF
          OK(state, data): q'_2(state),
          Exception(ex_type, state): Exception(ex_type, state),
          Fatal: Fatal,
          Hang: Hang
        ENDCASES)
{-11} OK?(q'_1(s'))
{-12} OK?(q'_2(state(q'_1(s'))))
{-13} OK?(memory_read(pm')(a')(state(q'_2(state(q'_1(s'))))))
-----
{1}  OK?(memory_read(pm')(a')(state(q'_1(s'))))
{2}  data(memory_read(pm')(a')(state(q'_2(state(q'_1(s')))))) =
      data(memory_read(pm')(a')(s'))

```

Rewriting using `expr_transformers_ok_ok`, matching in `*` where `s` gets `state(q1!1(s!1))`, `q` gets `memory_read(pm!1)(a!1)`, `states` gets `states!1`, `transformers` gets `memory_read_transformers(pm!1, addresses!1)`,

we get 2 subgoals:

C Proof scripts

expr_unchanged_memory_invariant_composition.1.1.2.1:

```

{-1} transformer_invariant?(states',
                               singleton(expr_2_super(λ (s: State):
                                                    CASES q'_1(s) OF
                                                    OK(state, data): q'_2(state),
                                                    Except-
tion(ex_type, state):
                                                    Except-
tion(ex_type, state),
                                                    Fatal: Fatal,
                                                    Hang: Hang
                                                    ENDCASES)))
{-2} transformers'(expr_2_super(q'_1))
{-3} transformers''(expr_2_super(q'_2))
{-4} transformers_ok?(states', memory_read_transformers(pm', addresses'))
{-5} transformer_invariant?(states', transformers')
{-6} states'(s')
{-7} addresses'(a')
{-8} OK?(memory_read(pm')(a')(s'))
{-9} transformer_invariant?(states', transformers'')
{-10} q' =
      expr_2_super(λ (s: State):
                  CASES q'_1(s) OF
                  OK(state, data): q'_2(state),
                  Exception(ex_type, state): Exception(ex_type, state),
                  Fatal: Fatal,
                  Hang: Hang
                  ENDCASES)
{-11} OK?(q'_1(s'))
{-12} OK?(q'_2(state(q'_1(s'))))
{-13} OK?(memory_read(pm')(a')(state(q'_2(state(q'_1(s'))))))
-----
{1} states'(state(q'_1(s')))
{2} OK?(memory_read(pm')(a')(state(q'_1(s'))))
{3} data(memory_read(pm')(a')(state(q'_2(state(q'_1(s')))))) =
      data(memory_read(pm')(a')(s'))

```

Rewriting using `expr_transformer_invariant_next_ok`, matching in * where s gets s!1, q gets q1!1, states gets states!1, transformers gets transformers!1,

This completes the proof of `expr_unchanged_memory_invariant_composition.1.1.2.1`.

expr_unchanged_memory_invariant_composition.1.1.2.2:

<pre> {-1} transformer_invariant?(states', singleton(expr_2_super(λ (s: State): CASES q'_1(s) OF OK(state, data): q'_2(state), Excep- tion(ex_type, state): Excep- tion(ex_type, state), Fatal: Fatal, Hang: Hang ENDCASES))) {-2} transformers'(expr_2_super(q'_1)) {-3} transformers''(expr_2_super(q'_2)) {-4} transformers_ok?(states', memory_read_transformers(pm', addresses')) {-5} transformer_invariant?(states', transformers') {-6} states'(s') {-7} addresses'(a') {-8} OK?(memory_read(pm')(a')(s')) {-9} transformer_invariant?(states', transformers'') {-10} q' = expr_2_super(λ (s: State): CASES q'_1(s) OF OK(state, data): q'_2(state), Exception(ex_type, state): Exception(ex_type, state), Fatal: Fatal, Hang: Hang ENDCASES) {-11} OK?(q'_1(s')) {-12} OK?(q'_2(state(q'_1(s')))) {-13} OK?(memory_read(pm')(a')(state(q'_2(state(q'_1(s')))))) </pre>	<pre> memory_read_transformers(pm', addresses')(expr_2_super(memory_read(pm')(a'))) {2} OK?(memory_read(pm')(a')(state(q'_1(s')))) {3} data(memory_read(pm')(a')(state(q'_2(state(q'_1(s')))))) = data(memory_read(pm')(a')(s')) </pre>
---	---

Rewriting using memory_read_transformers_memory_read, matching in *,

This completes the proof of expr_unchanged_memory_invariant_composition.1.1.2.2.

C Proof scripts

expr_unchanged_memory_invariant_composition.1.2:

{-1}	transformer_invariant?(states', singleton(expr_2_super(q'_1 ## q'_2)))
{-2}	transformers'(expr_2_super(q'_1))
{-3}	transformers''(expr_2_super(q'_2))
{-4}	transformers_ok?(states', memory_read_transformers(pm', addresses'))
{-5}	transformer_invariant?(states', transformers')
{-6}	states'(s') \wedge transformers'(expr_2_super(q'_1)) \wedge addresses'(a') \wedge OK?(expr_2_super(q'_1)(s')) \wedge OK?(memory_read(pm')(a')(s')) \wedge OK?(memory_read(pm')(a')(state(expr_2_super(q'_1)(s'))))
	\supset data(memory_read(pm')(a')(state(expr_2_super(q'_1)(s')))) = data(memory_read(pm')(a')(s'))
{-7}	transformer_invariant?(states', transformers'')
{-8}	states'(s')
{-9}	q' = expr_2_super(q'_1 ## q'_2)
{-10}	addresses'(a')
{-11}	OK?(expr_2_super(q'_1 ## q'_2)(s'))
{-12}	OK?(memory_read(pm')(a')(s'))
{-13}	OK?(memory_read(pm')(a')(state(expr_2_super(q'_1 ## q'_2)(s'))))
{1}	OK?(q'_1(s')) \vee abnormal?[State](expr_2_super[State, Data1](q'_1)(s'))
{2}	data(memory_read(pm')(a')(state(expr_2_super(q'_1 ## q'_2)(s')))) = data(memory_read(pm')(a')(s'))

Keeping (-11 1) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of expr_unchanged_memory_invariant_composition.1.2.

expr_unchanged_memory_invariant_composition.2:

{-1}	transformers'(expr_2_super(q'_1))
{-2}	transformers''(expr_2_super(q'_2))
{-3}	transformers_ok?(states', memory_read_transformers(pm', addresses'))
{-4}	transformer_invariant?(states', transformers')
{-5}	$\forall (s: \text{State}, q: [\text{State} \rightarrow \text{SuperResult}[\text{State}]], a: \text{Address}):$ $\text{states}'(s) \wedge$ $\text{transformers}'(q) \wedge$ $\text{addresses}'(a) \wedge$ $\text{OK?}(q(s)) \wedge$ $\text{OK?}(\text{memory_read}(\text{pm}')(\text{a})(\text{s})) \wedge$ $\text{OK?}(\text{memory_read}(\text{pm}')(\text{a})(\text{state}(q(\text{s}))))$ \supset $\text{data}(\text{memory_read}(\text{pm}')(\text{a})(\text{state}(q(\text{s})))) = \text{data}(\text{memory_read}(\text{pm}')(\text{a})(\text{s}))$
{-6}	transformer_invariant?(states', transformers'')
{-7}	$\forall (s: \text{State}, q: [\text{State} \rightarrow \text{SuperResult}[\text{State}]], a: \text{Address}):$ $\text{states}'(s) \wedge$ $\text{transformers}''(q) \wedge$ $\text{addresses}'(a) \wedge$ $\text{OK?}(q(s)) \wedge$ $\text{OK?}(\text{memory_read}(\text{pm}')(\text{a})(\text{s})) \wedge$ $\text{OK?}(\text{memory_read}(\text{pm}')(\text{a})(\text{state}(q(\text{s}))))$ \supset $\text{data}(\text{memory_read}(\text{pm}')(\text{a})(\text{state}(q(\text{s})))) = \text{data}(\text{memory_read}(\text{pm}')(\text{a})(\text{s}))$
{1}	transformer_invariant?(states', singleton(expr_2_super(q'_1 ## q'_2)))
{2}	$\text{transformer_invariant?}(\text{states}', \text{singleton}(\text{expr_2_super}(q'_1 \## q'_2))) \wedge$ $(\forall (s: \text{State}, q: [\text{State} \rightarrow \text{SuperResult}[\text{State}]], a: \text{Address}):$ $\text{states}'(s) \wedge$ $\text{singleton}(\text{expr_2_super}(q'_1 \## q'_2))(q) \wedge$ $\text{addresses}'(a) \wedge$ $\text{OK?}(q(s)) \wedge$ $\text{OK?}(\text{memory_read}(\text{pm}')(\text{a})(\text{s})) \wedge$ $\text{OK?}(\text{memory_read}(\text{pm}')(\text{a})(\text{state}(q(\text{s}))))$ \supset $\text{data}(\text{memory_read}(\text{pm}')(\text{a})(\text{state}(q(\text{s})))) =$ $\text{data}(\text{memory_read}(\text{pm}')(\text{a})(\text{s}))$

Rewriting using expr_composition_transformer_invariant, matching in *,

we get 2 subgoals:

C Proof scripts

expr_unchanged_memory_invariant_composition.2.1:

<pre> {-1} transformers'(expr_2_super(q1)) {-2} transformers''(expr_2_super(q2)) {-3} transformers_ok?(states', memory_read_transformers(pm', addresses')) {-4} transformer_invariant?(states', transformers') {-5} $\forall (s: \text{State}, q: [\text{State} \rightarrow \text{SuperResult}[\text{State}]], a: \text{Address}):$ states'(s) \wedge transformers'(q) \wedge addresses'(a) \wedge OK?(q(s)) \wedge OK?(memory_read(pm')(a)(s)) \wedge OK?(memory_read(pm')(a)(state(q(s)))) \supset data(memory_read(pm')(a)(state(q(s)))) = data(memory_read(pm')(a)(s)) {-6} transformer_invariant?(states', transformers'') {-7} $\forall (s: \text{State}, q: [\text{State} \rightarrow \text{SuperResult}[\text{State}]], a: \text{Address}):$ states'(s) \wedge transformers''(q) \wedge addresses'(a) \wedge OK?(q(s)) \wedge OK?(memory_read(pm')(a)(s)) \wedge OK?(memory_read(pm')(a)(state(q(s)))) \supset data(memory_read(pm')(a)(state(q(s)))) = data(memory_read(pm')(a)(s)) </pre>	<pre> {1} transformer_invariant?(states', singleton(expr_2_super(q1))) {2} transformer_invariant?(states', singleton(expr_2_super(q1 ## q2))) {3} FALSE \wedge ($\forall (s: \text{State}, q: [\text{State} \rightarrow \text{SuperResult}[\text{State}]], a: \text{Address}):$ states'(s) \wedge singleton(expr_2_super(q1 ## q2))(q) \wedge addresses'(a) \wedge OK?(q(s)) \wedge OK?(memory_read(pm')(a)(s)) \wedge OK?(memory_read(pm')(a)(state(q(s)))) \supset data(memory_read(pm')(a)(state(q(s)))) = data(memory_read(pm')(a)(s)) </pre>
---	---

Keeping (-1 -3 1) and hiding *,

Rewriting using transformer_invariant_mono_transformers, matching in * where transformers_1 gets singleton(expr_2_super(q1!1)), transformers_2 gets transformers!1,

Hiding formulas: (-2 2),

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of expr_unchanged_memory_invariant_composition.2.1.

expr_unchanged_memory_invariant_composition.2.2:

<pre> {-1} transformers'(expr_2_super(q'_1)) {-2} transformers''(expr_2_super(q'_2)) {-3} transformers_ok?(states', memory_read_transformers(pm', addresses')) {-4} transformer_invariant?(states', transformers') {-5} ∀ (s: State, q: [State → SuperResult[State]], a: Address): states'(s) ∧ transformers'(q) ∧ addresses'(a) ∧ OK?(q(s)) ∧ OK?(memory_read(pm')(a)(s)) ∧ OK?(memory_read(pm')(a)(state(q(s)))) ⊃ data(memory_read(pm')(a)(state(q(s)))) = data(memory_read(pm')(a)(s)) {-6} transformer_invariant?(states', transformers'') {-7} ∀ (s: State, q: [State → SuperResult[State]], a: Address): states'(s) ∧ transformers''(q) ∧ addresses'(a) ∧ OK?(q(s)) ∧ OK?(memory_read(pm')(a)(s)) ∧ OK?(memory_read(pm')(a)(state(q(s)))) ⊃ data(memory_read(pm')(a)(state(q(s)))) = data(memory_read(pm')(a)(s)) </pre>	<hr style="border: 0.5px solid black;"/> <pre> {1} transformer_invariant?(states', singleton(expr_2_super(q'_2))) {2} transformer_invariant?(states', singleton(expr_2_super(q'_1 ## q'_2))) {3} FALSE ∧ (∀ (s: State, q: [State → SuperResult[State]], a: Address): states'(s) ∧ singleton(expr_2_super(q'_1 ## q'_2))(q) ∧ addresses'(a) ∧ OK?(q(s)) ∧ OK?(memory_read(pm')(a)(s)) ∧ OK?(memory_read(pm')(a)(state(q(s)))) ⊃ data(memory_read(pm')(a)(state(q(s)))) = data(memory_read(pm')(a)(s))) </pre>
---	--

Keeping (-2 -5 1) and hiding *,

Rewriting using transformer_invariant_mono_transformers, matching in * where transformers_1 gets singleton(expr_2_super(q?1)), transformers_2 gets transformers!2,

Hiding formulas: (-2 2),

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of expr_unchanged_memory_invariant_composition.2.2.

Q.E.D.

C.125.2

Memory_Change_4.fexpr_unchanged_memory_invariant_composition

Terse proof for fexpr_unchanged_memory_invariant_composition.

C Proof scripts

fexpr_unchanged_memory_invariant_composition:

$ \begin{aligned} &\{1\} \quad \forall (\text{addresses: PRED}[\text{Address}], \text{pm: Memory_struct}[\text{State}], \text{states: PRED}[\text{State}], \\ &\quad \text{transformers_1: PRED}[[\text{State} \rightarrow \text{SuperResult}[\text{State}]]], \\ &\quad q_1: [\text{State} \rightarrow \text{ExprResult}[\text{State}, \text{Data1}]], \\ &\quad q_2: [\text{Data1} \rightarrow [\text{State} \rightarrow \text{ExprResult}[\text{State}, \text{Data2}]]], P: \text{PRED}[\text{Data1}]): \\ &\quad (\text{transformers_1}(\text{expr_2_super}(q_1)) \wedge \\ &\quad \text{transformers_ok}?(states, \text{memory_read_transformers}(\text{pm}, \text{addresses})) \wedge \\ &\quad \text{unchanged_memory_invariant}?(pm, states, \text{transformers_1}, \text{addresses}) \wedge \\ &\quad (\forall (s: (states)): \text{OK}?(q_1(s)) \supset P(\text{data}(q_1(s)))) \wedge \\ &\quad (\forall (d: (P')): \\ &\quad \quad \text{unchanged_memory_invariant}?(pm, states, \text{singleton}(\text{expr_2_super}(q_2(d))), \\ &\quad \quad \quad \text{addresses})) \\ &\quad \supset \text{unchanged_memory_invariant}?(pm, states, \text{singleton}(\text{expr_2_super}(q_1 \#\# q_2)), \text{ad-} \\ &\quad \quad \text{resses})) \end{aligned} $

Expanding the definition of unchanged_memory_invariant?,

Repeatedly Skolemizing and flattening,

Applying propositional simplification,

we get 2 subgoals:

fexpr_unchanged_memory_invariant_composition.1:

$ \begin{aligned} &\{-1\} \quad \text{transformers}'(\text{expr_2_super}(q'_1)) \\ &\{-2\} \quad \text{transformers_ok}?(states', \text{memory_read_transformers}(\text{pm}', \text{addresses}')) \\ &\{-3\} \quad \text{unchanged_memory_invariant}?(pm', states', \text{transformers}', \text{addresses}') \\ &\{-4\} \quad \forall (s: (states')): \text{OK}?(q'_1(s)) \supset P'(\text{data}(q'_1(s))) \\ &\{-5\} \quad \forall (d: (P')): \\ &\quad \quad \text{unchanged_memory_invariant}?(pm', states', \text{singleton}(\text{expr_2_super}(q'_2(d))), \\ &\quad \quad \quad \text{addresses}') \\ &\{1\} \quad \text{transformer_invariant}?(states', \text{singleton}(\text{expr_2_super}(q'_1 \#\# q'_2))) \end{aligned} $
--

Rewriting using fexpr_composition_transformer_invariant[State, Data1, Data2], matching in * where states gets states!1, P gets P!1,

we get 2 subgoals:

fexpr_unchanged_memory_invariant_composition.1.1:

$ \begin{aligned} &\{-1\} \quad \text{transformers}'(\text{expr_2_super}(q'_1)) \\ &\{-2\} \quad \text{transformers_ok}?(states', \text{memory_read_transformers}(\text{pm}', \text{addresses}')) \\ &\{-3\} \quad \text{unchanged_memory_invariant}?(pm', states', \text{transformers}', \text{addresses}') \\ &\{-4\} \quad \forall (s: (states')): \text{OK}?(q'_1(s)) \supset P'(\text{data}(q'_1(s))) \\ &\{-5\} \quad \forall (d: (P')): \\ &\quad \quad \text{unchanged_memory_invariant}?(pm', states', \text{singleton}(\text{expr_2_super}(q'_2(d))), \\ &\quad \quad \quad \text{addresses}') \\ &\{1\} \quad \text{transformer_invariant}?(states', \text{singleton}(\text{expr_2_super}(q'_1))) \\ &\{2\} \quad \text{transformer_invariant}?(states', \text{singleton}(\text{expr_2_super}(q'_1 \#\# q'_2))) \end{aligned} $

Rewriting using transformer_invariant_mono_transformers[State], matching in * where transformers_1 gets singleton(expr_2_super(q1!1)), transformers_2 gets transformers!1,

we get 2 subgoals:

fexpr_unchanged_memory_invariant_composition.1.1.1:

{-1}	$\text{transformers}'(\text{expr_2_super}(q'_1))$
{-2}	$\text{transformers_ok}'(\text{states}', \text{memory_read_transformers}(\text{pm}', \text{addresses}'))$
{-3}	$\text{unchanged_memory_invariant}'(\text{pm}', \text{states}', \text{transformers}', \text{addresses}')$
{-4}	$\forall (s: (\text{states}')): \text{OK}'(q'_1(s)) \supset P'(\text{data}(q'_1(s)))$
{-5}	$\forall (d: (P')):$ $\text{unchanged_memory_invariant}'(\text{pm}', \text{states}', \text{singleton}(\text{expr_2_super}(q'_2(d))),$ $\text{addresses}')$
{1}	$(\text{singleton}(\text{expr_2_super}(q'_1)) \subseteq \text{transformers}')$
{2}	$\text{transformer_invariant}'(\text{states}', \text{singleton}(\text{expr_2_super}(q'_1)))$
{3}	$\text{transformer_invariant}'(\text{states}', \text{singleton}(\text{expr_2_super}(q'_1 \ \#\# \ q'_2)))$

Keeping (-1 1) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of fexpr_unchanged_memory_invariant_composition.1.1.1.

fexpr_unchanged_memory_invariant_composition.1.1.2:

{-1}	$\text{transformers}'(\text{expr_2_super}(q'_1))$
{-2}	$\text{transformers_ok}'(\text{states}', \text{memory_read_transformers}(\text{pm}', \text{addresses}'))$
{-3}	$\text{unchanged_memory_invariant}'(\text{pm}', \text{states}', \text{transformers}', \text{addresses}')$
{-4}	$\forall (s: (\text{states}')): \text{OK}'(q'_1(s)) \supset P'(\text{data}(q'_1(s)))$
{-5}	$\forall (d: (P')):$ $\text{unchanged_memory_invariant}'(\text{pm}', \text{states}', \text{singleton}(\text{expr_2_super}(q'_2(d))),$ $\text{addresses}')$
{1}	$\text{transformer_invariant}'(\text{states}', \text{transformers}')$
{2}	$\text{transformer_invariant}'(\text{states}', \text{singleton}(\text{expr_2_super}(q'_1)))$
{3}	$\text{transformer_invariant}'(\text{states}', \text{singleton}(\text{expr_2_super}(q'_1 \ \#\# \ q'_2)))$

Expanding the definition of unchanged_memory_invariant?,

which is trivially true.

This completes the proof of fexpr_unchanged_memory_invariant_composition.1.1.2.

fexpr_unchanged_memory_invariant_composition.1.2:

{-1}	$\text{transformers}'(\text{expr_2_super}(q'_1))$
{-2}	$\text{transformers_ok}'(\text{states}', \text{memory_read_transformers}(\text{pm}', \text{addresses}'))$
{-3}	$\text{unchanged_memory_invariant}'(\text{pm}', \text{states}', \text{transformers}', \text{addresses}')$
{-4}	$\forall (s: (\text{states}')): \text{OK}'(q'_1(s)) \supset P'(\text{data}(q'_1(s)))$
{-5}	$\forall (d: (P')):$ $\text{unchanged_memory_invariant}'(\text{pm}', \text{states}', \text{singleton}(\text{expr_2_super}(q'_2(d))),$ $\text{addresses}')$
{1}	$\forall (d: (P')): \text{transformer_invariant}'(\text{states}', \text{singleton}(\text{expr_2_super}(q'_2(d))))$
{2}	$\text{transformer_invariant}'(\text{states}', \text{singleton}(\text{expr_2_super}(q'_1 \ \#\# \ q'_2)))$

Repeatedly Skolemizing and flattening,

Instantiating quantified variables,

Expanding the definition of unchanged_memory_invariant?,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of fexpr_unchanged_memory_invariant_composition.1.2.

C Proof scripts

fexpr_unchanged_memory_invariant_composition.2:

<pre> {-1} transformers'(expr_2_super(q1)) {-2} transformers_ok?(states', memory_read_transformers(pm', addresses')) {-3} unchanged_memory_invariant?(pm', states', transformers', addresses') {-4} $\forall (s: (states')): \text{OK?}(q1'(s)) \supset P'(\text{data}(q1'(s)))$ {-5} $\forall (d: (P')):$ unchanged_memory_invariant?(pm', states', singleton(expr_2_super(q2'(d))), addresses') </pre>	<hr/> <pre> {1} $\forall (s: \text{State}, q: [\text{State} \rightarrow \text{SuperResult}[\text{State}]], a: \text{Address}):$ states'(s) \wedge singleton(expr_2_super(q1' ## q2'))(q) \wedge addresses'(a) \wedge OK?(q(s)) \wedge OK?(memory_read(pm')(a)(s)) \wedge OK?(memory_read(pm')(a)(state(q(s)))) \supset data(memory_read(pm')(a)(state(q(s)))) = data(memory_read(pm')(a)(s)) </pre>
---	--

Repeatedly Skolemizing and flattening,

Expanding the definition of singleton,

Replacing using formula -2,

Rewriting using state_expr_2_super, matching in *,

we get 2 subgoals:

fexpr_unchanged_memory_invariant_composition.2.1:

<pre> {-1} states'(s') {-2} q' = expr_2_super(q1' ## q2') {-3} addresses'(a') {-4} OK?(expr_2_super(q1' ## q2')(s')) {-5} OK?(memory_read(pm')(a')(s')) {-6} OK?(memory_read(pm')(a')(state((q1' ## q2')(s')))) {-7} transformers'(expr_2_super(q1)) {-8} transformers_ok?(states', memory_read_transformers(pm', addresses')) {-9} unchanged_memory_invariant?(pm', states', transformers', addresses') {-10} $\forall (s: (states')): \text{OK?}(q1'(s)) \supset P'(\text{data}(q1'(s)))$ {-11} $\forall (d: (P')):$ unchanged_memory_invariant?(pm', states', singleton(expr_2_super(q2'(d))), addresses') </pre>	<hr/> <pre> {1} data(memory_read(pm')(a')(state((q1' ## q2')(s')))) = data(memory_read(pm')(a')(s')) </pre>
---	---

Rewriting using comp_eval_if_ok_fexpr, matching in *,

we get 2 subgoals:

fexpr_unchanged_memory_invariant_composition.2.1.1.1:

{-1}	states'(s')
{-2}	$q' = \text{expr_2_super}(q'_1 \ \#\# \ q'_2)$
{-3}	addresses'(a')
{-4}	$\text{OK?}(\text{expr_2_super}(q'_1 \ \#\# \ q'_2)(s'))$
{-5}	$\text{OK?}(\text{memory_read}(\text{pm}')(\text{a}')(\text{s}'))$
{-6}	$\text{OK?}(\text{memory_read}(\text{pm}')(\text{a}')(\text{state}((q'_1 \ \#\# \ q'_2(\text{data}(q'_1(s'))))(s'))))$
{-7}	transformers'(expr_2_super(q'_1))
{-8}	transformers_ok?(states', memory_read_transformers(pm', addresses'))
{-9}	unchanged_memory_invariant?(pm', states', transformers', addresses')
{-10}	$\forall (s: (\text{states}')): \text{OK?}(q'_1(s)) \supset P'(\text{data}(q'_1(s)))$
{-11}	$\forall (d: (P')):$ unchanged_memory_invariant?(pm', states', singleton(expr_2_super(q'_2(d))), addresses')
{1} data(memory_read(pm')(a')(state((q'_1 #\# q'_2(data(q'_1(s'))))(s')))) = data(memory_read(pm')(a')(s'))	

Instantiating the top quantifier in -11 with the terms: (data(q1!1(s!1))),

we get 2 subgoals:

fexpr_unchanged_memory_invariant_composition.2.1.1.1.1:

{-1}	states'(s')
{-2}	$q' = \text{expr_2_super}(q'_1 \ \#\# \ q'_2)$
{-3}	addresses'(a')
{-4}	$\text{OK?}(\text{expr_2_super}(q'_1 \ \#\# \ q'_2)(s'))$
{-5}	$\text{OK?}(\text{memory_read}(\text{pm}')(\text{a}')(\text{s}'))$
{-6}	$\text{OK?}(\text{memory_read}(\text{pm}')(\text{a}')(\text{state}((q'_1 \ \#\# \ q'_2(\text{data}(q'_1(s'))))(s'))))$
{-7}	transformers'(expr_2_super(q'_1))
{-8}	transformers_ok?(states', memory_read_transformers(pm', addresses'))
{-9}	unchanged_memory_invariant?(pm', states', transformers', addresses')
{-10}	$\forall (s: (\text{states}')): \text{OK?}(q'_1(s)) \supset P'(\text{data}(q'_1(s)))$
{-11}	unchanged_memory_invariant?(pm', states', singleton(expr_2_super(q'_2(data(q'_1(s'))))), addresses')
{1} data(memory_read(pm')(a')(state((q'_1 #\# q'_2(data(q'_1(s'))))(s')))) = data(memory_read(pm')(a')(s'))	

Using lemma expr_unchanged_memory_invariant_composition,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 2 subgoals:

C Proof scripts

fexpr_unchanged_memory_invariant_composition.2.1.1.1.1:

{-1}	transformers'(expr_2_super(q ₁ '))
{-2}	transformers_ok?(states', memory_read_transformers(pm', addresses'))
{-3}	unchanged_memory_invariant?(pm', states', transformers', addresses')
{-4}	unchanged_memory_invariant?(pm', states', singleton(expr_2_super(q ₂ '(data(q ₁ '(s'))))), addresses')
{-5}	unchanged_memory_invariant?(pm', states', singleton(expr_2_super(q ₁ ' ## q ₂ '(data(q ₁ '(s'))))), addresses')
{-6}	states'(s')
{-7}	q' = expr_2_super(q ₁ ' ## q ₂ ')
{-8}	addresses'(a')
{-9}	OK?(expr_2_super(q ₁ ' ## q ₂ ')(s'))
{-10}	OK?(memory_read(pm')(a')(s'))
{-11}	OK?(memory_read(pm')(a')(state((q ₁ ' ## q ₂ '(data(q ₁ '(s'))))(s'))))
{-12}	∀ (s: (states')): OK?(q ₁ '(s)) ⊃ P'(data(q ₁ '(s)))
{1}	data(memory_read(pm')(a')(state((q ₁ ' ## q ₂ '(data(q ₁ '(s'))))(s')))) = data(memory_read(pm')(a')(s'))

Using lemma expr_unchanged_memory_invariant_unchanged,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 2 subgoals:

fexpr_unchanged_memory_invariant_composition.2.1.1.1.1.1:

{-1}	unchanged_memory_invariant?(pm', states', singleton(expr_2_super(q ₁ ' ## q ₂ '(data(q ₁ '(s'))))), addresses')
{-2}	states'(s')
{-3}	addresses'(a')
{-4}	OK?(memory_read(pm')(a')(s'))
{-5}	OK?(memory_read(pm')(a')(state((q ₁ ' ## q ₂ '(data(q ₁ '(s'))))(s'))))
{-6}	transformers'(expr_2_super(q ₁ '))
{-7}	transformers_ok?(states', memory_read_transformers(pm', addresses'))
{-8}	unchanged_memory_invariant?(pm', states', transformers', addresses')
{-9}	unchanged_memory_invariant?(pm', states', singleton(expr_2_super(q ₂ '(data(q ₁ '(s'))))), addresses')
{-10}	q' = expr_2_super(q ₁ ' ## q ₂ ')
{-11}	OK?(expr_2_super(q ₁ ' ## q ₂ ')(s'))
{-12}	∀ (s: (states')): OK?(q ₁ '(s)) ⊃ P'(data(q ₁ '(s)))
{1}	OK?((q ₁ ' ## q ₂ '(data(q ₁ '(s'))))(s'))
{2}	data(memory_read(pm')(a')(state((q ₁ ' ## q ₂ '(data(q ₁ '(s'))))(s')))) = data(memory_read(pm')(a')(s'))

Keeping (-11 1) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of fexpr_unchanged_memory_invariant_composition.2.1.1.1.1.1.

fexpr_unchanged_memory_invariant_composition.2.1.1.1.1.2:

{-1}	unchanged_memory_invariant?(pm', states', singleton(expr_2_super(q'_1 ## q'_2(data(q'_1(s')))), addresses')
{-2}	states'(s')
{-3}	addresses'(a')
{-4}	OK?(memory_read(pm')(a')(s'))
{-5}	OK?(memory_read(pm')(a')(state((q'_1 ## q'_2(data(q'_1(s'))))(s'))))
{-6}	transformers'(expr_2_super(q'_1))
{-7}	transformers_ok?(states', memory_read_transformers(pm', addresses')
{-8}	unchanged_memory_invariant?(pm', states', transformers', addresses')
{-9}	unchanged_memory_invariant?(pm', states', singleton(expr_2_super(q'_2(data(q'_1(s')))), addresses')
{-10}	q' = expr_2_super(q'_1 ## q'_2)
{-11}	OK?(expr_2_super(q'_1 ## q'_2)(s'))
{-12}	$\forall (s: (\text{states}')): \text{OK?}(q'_1(s)) \supset P'(\text{data}(q'_1(s)))$
{1}	singleton(expr_2_super(q'_1 ## q'_2(data(q'_1(s')))) (expr_2_super((q'_1 ## q'_2(data(q'_1(s')))))))
{2}	data(memory_read(pm')(a')(state((q'_1 ## q'_2(data(q'_1(s'))))(s')))) = data(memory_read(pm')(a')(s'))

Expanding the definition of singleton,

which is trivially true.

This completes the proof of fexpr_unchanged_memory_invariant_composition.2.1.1.1.1.2.

fexpr_unchanged_memory_invariant_composition.2.1.1.1.2:

{-1}	transformers'(expr_2_super(q'_1))
{-2}	transformers_ok?(states', memory_read_transformers(pm', addresses')
{-3}	unchanged_memory_invariant?(pm', states', transformers', addresses')
{-4}	unchanged_memory_invariant?(pm', states', singleton(expr_2_super(q'_2(data(q'_1(s')))), addresses')
{-5}	states'(s')
{-6}	q' = expr_2_super(q'_1 ## q'_2)
{-7}	addresses'(a')
{-8}	OK?(expr_2_super(q'_1 ## q'_2)(s'))
{-9}	OK?(memory_read(pm')(a')(s'))
{-10}	OK?(memory_read(pm')(a')(state((q'_1 ## q'_2(data(q'_1(s'))))(s'))))
{-11}	$\forall (s: (\text{states}')): \text{OK?}(q'_1(s)) \supset P'(\text{data}(q'_1(s)))$
{1}	singleton(expr_2_super(q'_2(data(q'_1(s')))) (expr_2_super(q'_2(data(q'_1(s')))))))
{2}	data(memory_read(pm')(a')(state((q'_1 ## q'_2(data(q'_1(s'))))(s')))) = data(memory_read(pm')(a')(s'))

Expanding the definition of singleton,

which is trivially true.

This completes the proof of fexpr_unchanged_memory_invariant_composition.2.1.1.1.2.

C Proof scripts

fexpr_unchanged_memory_invariant_composition.2.1.1.2:

{-1}	states'(s')
{-2}	q' = expr_2_super(q1 ## q2)
{-3}	addresses'(a')
{-4}	OK?(expr_2_super(q1 ## q2)(s'))
{-5}	OK?(memory_read(pm')(a')(s'))
{-6}	OK?(memory_read(pm')(a')(state((q1 ## q2(data(q1'(s'))))(s'))))
{-7}	transformers'(expr_2_super(q1))
{-8}	transformers_ok?(states', memory_read_transformers(pm', addresses'))
{-9}	unchanged_memory_invariant?(pm', states', transformers', addresses')
{-10}	$\forall (s: (\text{states}')): \text{OK}(q_1'(s)) \supset P'(\text{data}(q_1'(s)))$
{1}	$P'(\text{data}[\text{State}, \text{Data1}](q_1'(s)))$
{2}	$\text{data}(\text{memory_read}(\text{pm}')(\text{a}')(\text{state}((q_1' \## q_2'(\text{data}(q_1'(s'))))(s')))) = \text{data}(\text{memory_read}(\text{pm}')(\text{a}')(\text{s}'))$

Instantiating quantified variables,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Keeping (-4 3) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of fexpr_unchanged_memory_invariant_composition.2.1.1.2.

fexpr_unchanged_memory_invariant_composition.2.1.2:

{-1}	states'(s')
{-2}	q' = expr_2_super(q1 ## q2)
{-3}	addresses'(a')
{-4}	OK?(expr_2_super(q1 ## q2)(s'))
{-5}	OK?(memory_read(pm')(a')(s'))
{-6}	OK?(memory_read(pm')(a')(state((q1 ## q2)(s'))))
{-7}	transformers'(expr_2_super(q1))
{-8}	transformers_ok?(states', memory_read_transformers(pm', addresses'))
{-9}	unchanged_memory_invariant?(pm', states', transformers', addresses')
{-10}	$\forall (s: (\text{states}')): \text{OK}(q_1'(s)) \supset P'(\text{data}(q_1'(s)))$
{-11}	$\forall (d: (P')): \text{unchanged_memory_invariant?}(\text{pm}', \text{states}', \text{singleton}(\text{expr_2_super}(q_2'(d))), \text{addresses}')$
{1}	$\text{OK}(q_1'(s'))$
{2}	$\text{data}(\text{memory_read}(\text{pm}')(\text{a}')(\text{state}((q_1' \## q_2')(s')))) = \text{data}(\text{memory_read}(\text{pm}')(\text{a}')(\text{s}'))$

Keeping (-4 1) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of fexpr_unchanged_memory_invariant_composition.2.1.2.

fexpr_unchanged_memory_invariant_composition.2.2:

{-1}	$states'(s')$
{-2}	$q' = \text{expr_2_super}(q'_1 \ \#\# \ q'_2)$
{-3}	$addresses'(a')$
{-4}	$OK?(\text{expr_2_super}(q'_1 \ \#\# \ q'_2)(s'))$
{-5}	$OK?(\text{memory_read}(pm')(a')(s'))$
{-6}	$OK?(\text{memory_read}(pm')(a')(\text{state}(\text{expr_2_super}(q'_1 \ \#\# \ q'_2)(s'))))$
{-7}	$transformers'(\text{expr_2_super}(q'_1))$
{-8}	$transformers_ok?(states', \text{memory_read_transformers}(pm', addresses'))$
{-9}	$\text{unchanged_memory_invariant?}(pm', states', transformers', addresses')$
{-10}	$\forall (s: (states')): OK?(q'_1(s)) \supset P'(\text{data}(q'_1(s)))$
{-11}	$\forall (d: (P')):$ $\text{unchanged_memory_invariant?}(pm', states', \text{singleton}(\text{expr_2_super}(q'_2(d)),$ $\text{addresses}')$
<hr/>	
{1}	$\text{has_next_state}((q'_1 \ \#\# \ q'_2)(s'))$
{2}	$\text{data}(\text{memory_read}(pm')(a')(\text{state}(\text{expr_2_super}(q'_1 \ \#\# \ q'_2)(s')))) =$ $\text{data}(\text{memory_read}(pm')(a')(s'))$

Rewriting using ok_expr_2_super, matching in *,
 Expanding the definition of has_next_state,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 This completes the proof of fexpr_unchanged_memory_invariant_composition.2.2.
 Q.E.D.

C.126 Proofs for Memory_Physical_Memory (physical_memory.pvs)

This theory contains no provable formal statements.

C.127 Proofs for Memory_access (result.pvs)

This theory contains no provable formal statements.

C.128 Proofs for Memory_access_adt (Memory_access_adt.pvs)

This theory contains no provable formal statements.

C.129 Proofs for Memory_access_adt_reduce (Memory_access_adt.pvs)

This theory contains no provable formal statements.

C.130 Proofs for Memory_access_util (result.pvs)

This theory contains no provable formal statements.

C.131 Proofs for Memory_privilege (result.pvs)

This theory contains no provable formal statements.

C.132 Proofs for Memory_privilege_adt (Memory_privilege_adt.pvs)

This theory contains no provable formal statements.

C.133 Proofs for Memory_privilege_adt_reduce (Memory_privilege_adt.pvs)

This theory contains no provable formal statements.

C.134 Proofs for Memory_privilege_util (result.pvs)

C.134.1 Memory_privilege_util.bool_to_memory_privilege_iso

Terse proof for bool_to_memory_privilege_iso.

bool_to_memory_privilege_iso:

$$\frac{}{\{1\} \quad \forall (ar: \text{Memory_privilege}): \text{bool_to_memory_privilege}(\text{memory_privilege_to_bool}(ar)) = ar}$$

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of bool_to_memory_privilege_iso.
Q.E.D.

C.135 Proofs for Min_Max_Extension (vfiasco-prelude.pvs)

C.135.1 Min_Max_Extension.max_stable_under_extension_TCC1

Terse proof for max_stable_under_extension_TCC1.

max_stable_under_extension_TCC1:

$$\frac{}{\{1\} \quad \forall (P: \text{non_empty_finite_set}[S]): \neg \text{empty?}[T](\text{extend}[T, S, \text{bool}, \text{FALSE}](P))}$$

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of max_stable_under_extension_TCC1.
Q.E.D.

C.135.2 Min_Max_Extension.max_stable_under_extension

Terse proof for max_stable_under_extension.

max_stable_under_extension:

$$\frac{\{1\} \quad \forall (P: \text{non_empty_finite_set}[S]):}{\text{max}[T, \leq](\text{extend}[T, S, \text{bool}, \text{FALSE}](P)) = \text{max}[S, \text{restrict}[[T, T], [S, S], \text{boolean}](\leq)](P)}$$

Repeatedly Skolemizing and flattening,
 Rewriting using max_lem, matching in *,
 Applying propositional simplification,
 we get 2 subgoals:

max_stable_under_extension.1:

$$\frac{\{-1\} \quad \text{is_finite}[S](P')}{\begin{array}{l} \{1\} \quad \text{extend}[T, S, \text{bool}, \text{FALSE}](P')(\text{max}[S, \text{restrict}[[T, T], [S, S], \text{boolean}](\leq)](P')) \\ \{2\} \quad \text{empty?}[S](P') \end{array}}$$

Expanding the definition of extend,
 which is trivially true.
 This completes the proof of max_stable_under_extension.1.
 max_stable_under_extension.2:

$$\frac{\{-1\} \quad \text{is_finite}[S](P')}{\begin{array}{l} \{1\} \quad \forall (x: (\text{extend}[T, S, \text{bool}, \text{FALSE}](P'))): \\ \quad x \leq \text{max}[S, \text{restrict}[[T, T], [S, S], \text{boolean}](\leq)](P') \\ \{2\} \quad \text{empty?}[S](P') \end{array}}$$

Repeatedly Skolemizing and flattening,
 Expanding the definition of extend,
 Adding type constraints for max[S, restrict[[T, T], [S, S], boolean](<=)](P!1),
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 Instantiating the top quantifier in -3 with the terms: (x!1),
 Expanding the definition of restrict,
 which is trivially true.
 This completes the proof of max_stable_under_extension.2.
 Q.E.D.

C.135.3 Min_Max_Extension.min_stable_under_extension

Terse proof for min_stable_under_extension.

min_stable_under_extension:

$$\frac{\{1\} \quad \forall (P: \text{non_empty_finite_set}[S]):}{\text{min}[T, \leq](\text{extend}[T, S, \text{bool}, \text{FALSE}](P)) = \text{min}[S, \text{restrict}[[T, T], [S, S], \text{boolean}](\leq)](P)}$$

Repeatedly Skolemizing and flattening,
 Rewriting using min_lem, matching in *,
 Expanding the definition of extend,
 Repeatedly Skolemizing and flattening,
 Adding type constraints for min[S, restrict[[T, T], [S, S], boolean](<=)](P!1),
 Expanding the definition of extend,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Instantiating the top quantifier in -3 with the terms: (x!1),
 Expanding the definition of restrict,
 which is trivially true.
 This completes the proof of `min_stable_under_extension`.
 Q.E.D.

C.136 Proofs for More_Divides (vfiasco-prelude.pvs)

C.136.1 More_Divides.expt_divides

Terse proof for `expt_divides`.

`expt_divides`:

$$\frac{}{\{1\} \quad \forall (m, n: \text{nat}, a: \text{posnat}): a > 1 \wedge n \leq m \supset \text{divides}(\text{expt}(a, n), \text{expt}(a, m))}$$

Repeatedly Skolemizing and flattening,
 Expanding the definition of divides,
 Instantiating the top quantifier in 1 with the terms: `expt(a', m' - n')`,
 we get 2 subgoals:

`expt_divides.1`:

$$\frac{\begin{array}{l} \{-1\} \quad m' \geq 0 \\ \{-2\} \quad n' \geq 0 \\ \{-3\} \quad a' > 0 \\ \{-4\} \quad a' > 1 \\ \{-5\} \quad n' \leq m' \end{array}}{\{1\} \quad \text{expt}(a', m') = \text{expt}(a', n') \times \text{expt}(a', m' - n')}$$

Rewriting using `expt_plus_aux`, matching in *,
 This completes the proof of `expt_divides.1`.

`expt_divides.2`:

$$\frac{\begin{array}{l} \{-1\} \quad m' \geq 0 \\ \{-2\} \quad n' \geq 0 \\ \{-3\} \quad a' > 0 \\ \{-4\} \quad a' > 1 \\ \{-5\} \quad n' \leq m' \end{array}}{\{1\} \quad m' - n' \geq 0}$$

Simplifying, rewriting, and recording with decision procedures,
 This completes the proof of `expt_divides.2`.
 Q.E.D.

C.136.2 More_Divides.divides_expt_gt

Terse proof for `divides_expt_gt`.

`divides_expt_gt`:

$$\frac{}{\{1\} \quad \forall (n, i, j: \text{nat}): j \leq i \wedge \text{divides}(\text{expt}(2, i), n) \supset \text{divides}(\text{expt}(2, j), n)}$$

Repeatedly Skolemizing and flattening,
 Applying `divides_transitive`
 Instantiating the top quantifier in -1 with the terms: `n', expt(2, i'), expt(2, j')`,

Simplifying, rewriting, and recording with decision procedures,
 Rewriting using `expt_divides`, matching in `*`,
 This completes the proof of `divides_expt_gt`.
 Q.E.D.

C.137 Proofs for More_Function (vfiasco-prelude.pvs)

C.137.1 More_Function.restrict_to_image_TCC1

Terse proof for `restrict_to_image_TCC1`.

`restrict_to_image_TCC1`:

$$\frac{\{1\} \quad \forall (f: [\text{Domain} \rightarrow \text{Range}]): \quad \forall (x_1: \text{Domain}): \text{image}[\text{Domain}, \text{Range}](f, \text{fullset}[\text{Domain}])(f(x_1))}{}$$

Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of `restrict_to_image_TCC1`.
 Q.E.D.

C.137.2 More_Function.surjective_restrict_to_image

Terse proof for `surjective_restrict_to_image`.

`surjective_restrict_to_image`:

$$\frac{\{1\} \quad \forall (f: [\text{Domain} \rightarrow \text{Range}]): \text{surjective?}(\text{restrict_to_image}(f))}{}$$

Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of `surjective_restrict_to_image`.
 Q.E.D.

C.137.3 More_Function.bijective_restrict_to_image

Terse proof for `bijective_restrict_to_image`.

`bijective_restrict_to_image`:

$$\frac{\{1\} \quad \forall (f: [\text{Domain} \rightarrow \text{Range}]): \text{injective?}(f) \supset \text{bijective?}(\text{restrict_to_image}(f))}{}$$

Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of `bijective_restrict_to_image`.
 Q.E.D.

C.138 Proofs for More_List_Props (vfiasco-prelude.pvs)

C.138.1 More_List_Props.length_is_cons

Terse proof for `length_is_cons`.

`length_is_cons`:

$$\frac{\{1\} \quad \forall (l: \text{list}[T]): \text{length}(l) > 0 \supset \text{cons?}(l)}{}$$

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `length_is_cons`.
Q.E.D.

C.138.2 More_List_Props.cons_length

Terse proof for `cons_length`.

`cons_length`:

$$\{1\} \quad \forall (l: (\text{cons?}[T])): \text{length}(l) > 0$$

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `cons_length`.
Q.E.D.

C.138.3 More_List_Props.length_cdr_TCC1

Terse proof for `length_cdr_TCC1`.

`length_cdr_TCC1`:

$$\{1\} \quad \forall (l: \text{list}[T]): \text{length}(l) > 0 \supset \text{cons?}[T](l)$$

Repeatedly Skolemizing and flattening,
Rewriting using `length_is_cons`, matching in *,
This completes the proof of `length_cdr_TCC1`.
Q.E.D.

C.138.4 More_List_Props.length_cdr

Terse proof for `length_cdr`.

`length_cdr`:

$$\{1\} \quad \forall (l: \text{list}[T]): \text{length}(l) > 0 \supset \text{length}(\text{cdr}(l)) = \text{length}(l) - 1$$

Repeatedly Skolemizing and flattening,
Expanding the definition of `length`,
Expanding the definition of `length`,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `length_cdr`.
Q.E.D.

C.138.5 More_List_Props.conslist_induction

Terse proof for `conslist_induction`.

`conslist_induction`:

$$\{1\} \quad \forall (p: [(\text{cons?}[T]) \rightarrow \text{boolean}]):$$
$$(\forall (t: T): p(\text{cons}(t, \text{null}))) \wedge$$
$$(\forall (t: T, \text{tail}: (\text{cons?}[T])): p(\text{tail}) \supset p(\text{cons}(t, \text{tail})))$$
$$\supset (\forall (l: (\text{cons?}[T])): p(l))$$

Skolemizing and flattening,
 Inducting on l on formula 1,
 we get 3 subgoals:

conslist_induction.1:

{-1}	$\forall (t: T): p'(\text{cons}(t, \text{null}))$
{-2}	$\forall (t: T, \text{tail}: (\text{cons}?[T])): p'(\text{tail}) \supset p'(\text{cons}(t, \text{tail}))$
{1}	$\text{cons}?[T](l')$
{2}	$p'(l')$

Simplifying, rewriting, and recording with decision procedures,
 This completes the proof of **conslist_induction.1**.

conslist_induction.2:

{-1}	$\forall (t: T): p'(\text{cons}(t, \text{null}))$
{-2}	$\forall (t: T, \text{tail}: (\text{cons}?[T])): p'(\text{tail}) \supset p'(\text{cons}(t, \text{tail}))$
{1}	$\text{FALSE} \supset p'(\text{null})$

Simplifying, rewriting, and recording with decision procedures,
 This completes the proof of **conslist_induction.2**.

conslist_induction.3:

{-1}	$\forall (t: T): p'(\text{cons}(t, \text{null}))$
{-2}	$\forall (t: T, \text{tail}: (\text{cons}?[T])): p'(\text{tail}) \supset p'(\text{cons}(t, \text{tail}))$
{1}	$\forall (\text{cons1_var}: T, \text{cons2_var}: \text{list}[T]):$ $(\text{cons}?[T](\text{cons2_var}) \supset p'(\text{cons2_var})) \supset \text{TRUE} \supset p'(\text{cons}(\text{cons1_var}, \text{cons2_var}))$

Repeatedly Skolemizing and flattening,
 Case splitting on $\text{cons2_var}!1 = \text{null}$,
 we get 2 subgoals:

conslist_induction.3.1:

{-1}	$\text{cons2_var}' = \text{null}$
{-2}	$\text{cons}?[T](\text{cons2_var}') \supset p'(\text{cons2_var}')$
{-3}	$\forall (t: T): p'(\text{cons}(t, \text{null}))$
{-4}	$\forall (t: T, \text{tail}: (\text{cons}?[T])): p'(\text{tail}) \supset p'(\text{cons}(t, \text{tail}))$
{1}	$p'(\text{cons}(\text{cons1_var}', \text{cons2_var}'))$

Instantiating quantified variables,
 Simplifying, rewriting, and recording with decision procedures,
 This completes the proof of **conslist_induction.3.1**.

conslist_induction.3.2:

{-1}	$\text{cons}?[T](\text{cons2_var}') \supset p'(\text{cons2_var}')$
{-2}	$\forall (t: T): p'(\text{cons}(t, \text{null}))$
{-3}	$\forall (t: T, \text{tail}: (\text{cons}?[T])): p'(\text{tail}) \supset p'(\text{cons}(t, \text{tail}))$
{1}	$\text{cons2_var}' = \text{null}$
{2}	$p'(\text{cons}(\text{cons1_var}', \text{cons2_var}'))$

Simplifying, rewriting, and recording with decision procedures,
 Instantiating quantified variables,
 Simplifying, rewriting, and recording with decision procedures,
 This completes the proof of **conslist_induction.3.2**.
 Q.E.D.

C.138.6 More_List_Props.head_TCC1

Terse proof for head_TCC1.

head_TCC1:

$$\frac{}{\{1\} \quad \forall (l: \text{list}[T], n: \text{upto}(\text{length}[T](l))): \neg n = 0 \supset \text{cons?}[T](l)}$$

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of head_TCC1.

Q.E.D.

C.138.7 More_List_Props.head_TCC2

Terse proof for head_TCC2.

head_TCC2:

$$\frac{}{\{1\} \quad \forall (l: \text{list}[T], n: \text{upto}(\text{length}[T](l))): \neg n = 0 \supset n - 1 \geq 0 \wedge n - 1 \leq \text{length}[T](\text{cdr}[T](l))}$$

Repeatedly Skolemizing and flattening,

Expanding the definition of length,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of head_TCC2.

Q.E.D.

C.138.8 More_List_Props.head_TCC3

Terse proof for head_TCC3.

head_TCC3:

$$\frac{}{\{1\} \quad \forall (l: \text{list}[T], n: \text{upto}(\text{length}[T](l))): \neg n = 0 \supset n - 1 < n}$$

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of head_TCC3.

Q.E.D.

C.138.9 More_List_Props.head_tail

Terse proof for head_tail.

head_tail:

$$\frac{}{\{1\} \quad \forall (l: \text{list}[T], n: \text{nat}): n \leq \text{length}(l) \supset \text{append}(\text{head}(l, n), \text{tail}(l, n)) = l}$$

Inducting on l on formula 1,

we get 2 subgoals:

head_tail.1:

$$\frac{}{\{1\} \quad \forall (n: \text{nat}): n \leq \text{length}(\text{null}) \supset \text{append}(\text{head}(\text{null}, n), \text{tail}(\text{null}, n)) = \text{null}}$$

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of head_tail.1.

head_tail.2:

$\begin{array}{l} \{1\} \quad \forall (\text{cons1_var}: T, \text{cons2_var}: \text{list}[T]): \\ \quad (\forall (n: \text{nat}): \\ \quad \quad n \leq \text{length}(\text{cons2_var}) \supset \\ \quad \quad \quad \text{append}(\text{head}(\text{cons2_var}, n), \text{tail}(\text{cons2_var}, n)) = \text{cons2_var}) \\ \quad \supset \\ \quad (\forall (n: \text{nat}): \\ \quad \quad n \leq \text{length}(\text{cons}(\text{cons1_var}, \text{cons2_var})) \supset \\ \quad \quad \quad \text{append}(\text{head}(\text{cons}(\text{cons1_var}, \text{cons2_var}), n), \\ \quad \quad \quad \quad \text{tail}(\text{cons}(\text{cons1_var}, \text{cons2_var}), n)) \\ \quad \quad \quad = \text{cons}(\text{cons1_var}, \text{cons2_var})) \end{array}$
--

Repeatedly Skolemizing and flattening,

Expanding the definition of length,

Expanding the definition of head,

Expanding the definition of tail,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 2 subgoals:

head_tail.2.1:

$\begin{array}{l} \{-1\} \quad n' \geq 0 \\ \{-2\} \quad \forall (n: \text{nat}): \\ \quad \quad n \leq \text{length}(\text{cons2_var}') \supset \\ \quad \quad \quad \text{append}(\text{head}(\text{cons2_var}', n), \text{tail}(\text{cons2_var}', n)) = \text{cons2_var}' \\ \{-3\} \quad n' \leq 1 + \text{length}(\text{cons2_var}') \\ \{-4\} \quad n' = 0 \end{array}$
$\{1\} \quad \text{append}(\text{null}, \text{cons}(\text{cons1_var}', \text{cons2_var}')) = \text{cons}(\text{cons1_var}', \text{cons2_var}')$

Expanding the definition of append,

which is trivially true.

This completes the proof of head_tail.2.1.

head_tail.2.2:

$\begin{array}{l} \{-1\} \quad n' \geq 0 \\ \{-2\} \quad \forall (n: \text{nat}): \\ \quad \quad n \leq \text{length}(\text{cons2_var}') \supset \\ \quad \quad \quad \text{append}(\text{head}(\text{cons2_var}', n), \text{tail}(\text{cons2_var}', n)) = \text{cons2_var}' \\ \{-3\} \quad n' \leq 1 + \text{length}(\text{cons2_var}') \end{array}$
$\begin{array}{l} \{1\} \quad n' = 0 \\ \{2\} \quad \text{append}(\text{cons}(\text{cons1_var}', \text{head}(\text{cons2_var}', n' - 1)), \\ \quad \quad \quad \text{tail}(\text{cons2_var}', n' - 1)) \\ \quad \quad = \text{cons}(\text{cons1_var}', \text{cons2_var}') \end{array}$

Expanding the definition of append,

Rewriting using -2, matching in *,

This completes the proof of head_tail.2.2.

Q.E.D.

C.138.10 More_List_Props.length_head

Terse proof for length_head.

C Proof scripts

`length_head:`

$$\{1\} \quad \forall (l: \text{list}[T], n: \text{nat}): n \leq \text{length}(l) \supset \text{length}(\text{head}(l, n)) = n$$

Inducting on l on formula 1,

we get 2 subgoals:

`length_head.1:`

$$\{1\} \quad \forall (n: \text{nat}): n \leq \text{length}(\text{null}) \supset \text{length}(\text{head}(\text{null}, n)) = n$$

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `length_head.1`.

`length_head.2:`

$$\{1\} \quad \forall (\text{cons1_var}: T, \text{cons2_var}: \text{list}[T]): \\ (\forall (n: \text{nat}): n \leq \text{length}(\text{cons2_var}) \supset \text{length}(\text{head}(\text{cons2_var}, n)) = n) \supset \\ (\forall (n: \text{nat}): \\ n \leq \text{length}(\text{cons}(\text{cons1_var}, \text{cons2_var})) \supset \\ \text{length}(\text{head}(\text{cons}(\text{cons1_var}, \text{cons2_var}), n)) = n)$$

Repeatedly Skolemizing and flattening,

Expanding the definition of `head`,

Expanding the definition of `length`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Instantiating the top quantifier in -2 with the terms: $(n!1 - 1)$,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `length_head.2`.

Q.E.D.

C.138.11 More_List_Props.length_tail

Terse proof for `length_tail`.

`length_tail:`

$$\{1\} \quad \forall (l: \text{list}[T], n: \text{nat}): \\ n \leq \text{length}(l) \supset \text{length}(\text{tail}(l, n)) = \text{length}(l) - n$$

Inducting on l on formula 1,

we get 2 subgoals:

`length_tail.1:`

$$\{1\} \quad \forall (n: \text{nat}): n \leq \text{length}(\text{null}) \supset \text{length}(\text{tail}(\text{null}, n)) = \text{length}(\text{null}) - n$$

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `length_tail.1`.

length_tail.2:

$\{1\} \quad \forall (\text{cons1_var}: T, \text{cons2_var}: \text{list}[T]):$ $\quad (\forall (n: \text{nat}):$ $\quad \quad n \leq \text{length}(\text{cons2_var}) \supset$ $\quad \quad \text{length}(\text{tail}(\text{cons2_var}, n)) = \text{length}(\text{cons2_var}) - n$ $\quad \quad \supset$ $\quad \quad (\forall (n: \text{nat}):$ $\quad \quad \quad n \leq \text{length}(\text{cons}(\text{cons1_var}, \text{cons2_var})) \supset$ $\quad \quad \quad \text{length}(\text{tail}(\text{cons}(\text{cons1_var}, \text{cons2_var}), n)) =$ $\quad \quad \quad \text{length}(\text{cons}(\text{cons1_var}, \text{cons2_var})) - n$
--

Repeatedly Skolemizing and flattening,

Expanding the definition of tail,

Expanding the definition of length,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 2 subgoals:

length_tail.2.1:

$\{-1\} \quad n' \geq 0$ $\{-2\} \quad \forall (n: \text{nat}):$ $\quad n \leq \text{length}(\text{cons2_var}') \supset$ $\quad \quad \text{length}(\text{tail}(\text{cons2_var}', n)) = \text{length}(\text{cons2_var}') - n$ $\{-3\} \quad n' \leq \text{length}(\text{cons}(\text{cons1_var}', \text{cons2_var}'))$ $\{-4\} \quad \text{null}?(\text{tail}(\text{cons2_var}', n' - 1))$
$\{1\} \quad n' = 0$ $\{2\} \quad 0 = 1 + \text{length}(\text{cons2_var}') - n'$

Instantiating the top quantifier in -2 with the terms: (n!1 - 1),

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of length_tail.2.1.

length_tail.2.2:

$\{-1\} \quad n' \geq 0$ $\{-2\} \quad \forall (n: \text{nat}):$ $\quad n \leq \text{length}(\text{cons2_var}') \supset$ $\quad \quad \text{length}(\text{tail}(\text{cons2_var}', n)) = \text{length}(\text{cons2_var}') - n$ $\{-3\} \quad n' \leq \text{length}(\text{cons}(\text{cons1_var}', \text{cons2_var}'))$
$\{1\} \quad n' = 0$ $\{2\} \quad \text{null}?(\text{tail}(\text{cons2_var}', n' - 1))$ $\{3\} \quad \text{length}(\text{cdr}(\text{tail}(\text{cons2_var}', n' - 1))) = \text{length}(\text{cons2_var}') - n'$

Instantiating the top quantifier in -2 with the terms: (n!1 - 1),

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of length_tail.2.2.

Q.E.D.

C.138.12 More_List_Props.nth_head_TCC1

Terse proof for nth_head_TCC1.

nth_head_TCC1:

$\{1\} \quad \forall (l: \text{list}[T], n, i: \text{nat}):$ $\quad n \leq \text{length}(l) \wedge i < n \supset i < \text{length}[T](\text{head}(l, n))$

Repeatedly Skolemizing and flattening,
 Rewriting using `length_head`, matching in `*`,
 This completes the proof of `nth_head_TCC1`.
 Q.E.D.

C.138.13 More_List_Props.nth_head_TCC2

Terse proof for `nth_head_TCC2`.

`nth_head_TCC2`:

$$\frac{\{1\} \quad \forall (l: \text{list}[T], n, i: \text{nat}): n \leq \text{length}(l) \wedge i < n \supset i < \text{length}[T](l)}{\quad}$$

Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of `nth_head_TCC2`.
 Q.E.D.

C.138.14 More_List_Props.nth_head

Terse proof for `nth_head`.

`nth_head`:

$$\frac{\{1\} \quad \forall (l: \text{list}[T], n, i: \text{nat}): n \leq \text{length}(l) \wedge i < n \supset \text{nth}(\text{head}(l, n), i) = \text{nth}(l, i)}{\quad}$$

Inducting on n on formula 1,
 we get 4 subgoals:

`nth_head.1`:

$$\frac{\{1\} \quad \forall (l: \text{list}[T], i: \text{nat}): 0 \leq \text{length}(l) \wedge i < 0 \supset \text{nth}(\text{head}(l, 0), i) = \text{nth}(l, i)}{\quad}$$

Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of `nth_head.1`.

`nth_head.2`:

$$\frac{\{1\} \quad \forall j: (\forall (l: \text{list}[T], i: \text{nat}): j \leq \text{length}(l) \wedge i < j \supset \text{nth}(\text{head}(l, j), i) = \text{nth}(l, i)) \supset (\forall (l: \text{list}[T], i: \text{nat}): j + 1 \leq \text{length}(l) \wedge i < j + 1 \supset \text{nth}(\text{head}(l, j + 1), i) = \text{nth}(l, i))}{\quad}$$

Repeatedly Skolemizing and flattening,
 Expanding the definition of `head`,
 Expanding the definition of `nth`,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 Rewriting using `-3`, matching in `*`,
 Rewriting using `length_cdr`, matching in `*`,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 This completes the proof of `nth_head.2`.

`nth_head.3:`

{1}	$\forall (n: \text{nat}, l: \text{list}[T], i: \text{nat}): n \leq \text{length}(l) \wedge i < n \supset i < \text{length}[T](l)$
{2}	$\forall (l: \text{list}[T], i: \text{nat}):$ $n' \leq \text{length}(l) \wedge i < n' \supset \text{nth}(\text{head}(l, n'), i) = \text{nth}(l, i)$

Hiding formulas: 2,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `nth_head.3`.

`nth_head.4:`

{1}	$\forall (n: \text{nat}, l: \text{list}[T], i: \text{nat}):$ $n \leq \text{length}(l) \wedge i < n \supset i < \text{length}[T](\text{head}(l, n))$
{2}	$\forall (l: \text{list}[T], i: \text{nat}):$ $n' \leq \text{length}(l) \wedge i < n' \supset \text{nth}(\text{head}(l, n'), i) = \text{nth}(l, i)$

Hiding formulas: 2,

Repeatedly Skolemizing and flattening,

Rewriting using `length_head`, matching in *,

This completes the proof of `nth_head.4`.

Q.E.D.

C.138.15 More_List_Props.nth_tail_TCC1

Terse proof for `nth_tail_TCC1`.

`nth_tail_TCC1:`

{1}	$\forall (l: \text{list}[T], n, i: \text{nat}):$ $n \leq \text{length}(l) \wedge i + n < \text{length}(l) \supset i < \text{length}[T](\text{tail}(l, n))$
-----	---

Repeatedly Skolemizing and flattening,

Rewriting using `length_tail`, matching in *,

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `nth_tail_TCC1`.

Q.E.D.

C.138.16 More_List_Props.nth_tail_TCC2

Terse proof for `nth_tail_TCC2`.

`nth_tail_TCC2:`

{1}	$\forall (l: \text{list}[T], n, i: \text{nat}):$ $n \leq \text{length}(l) \wedge i + n < \text{length}(l) \supset n + i < \text{length}[T](l)$
-----	---

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `nth_tail_TCC2`.

Q.E.D.

C.138.17 More_List_Props.nth_tail

Terse proof for `nth_tail`.

C Proof scripts

`nth_tail:`

{1} $\forall (l: \text{list}[T], n, i: \text{nat}):$
 $n \leq \text{length}(l) \wedge i + n < \text{length}(l) \supset$
 $\text{nth}(\text{tail}(l, n), i) = \text{nth}(l, n + i)$

Inducting on n on formula 1,

we get 4 subgoals:

`nth_tail.1:`

{1} $\forall (l: \text{list}[T], i: \text{nat}):$
 $0 \leq \text{length}(l) \wedge i + 0 < \text{length}(l) \supset$
 $\text{nth}(\text{tail}(l, 0), i) = \text{nth}(l, 0 + i)$

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `nth_tail.1`.

`nth_tail.2:`

{1} $\forall j:$
 $(\forall (l: \text{list}[T], i: \text{nat}):$
 $j \leq \text{length}(l) \wedge i + j < \text{length}(l) \supset$
 $\text{nth}(\text{tail}(l, j), i) = \text{nth}(l, j + i))$
 \supset
 $(\forall (l: \text{list}[T], i: \text{nat}):$
 $j + 1 \leq \text{length}(l) \wedge i + j + 1 < \text{length}(l) \supset$
 $\text{nth}(\text{tail}(l, j + 1), i) = \text{nth}(l, j + 1 + i))$

Repeatedly Skolemizing and flattening,

Expanding the definition of tail,

Expanding the definition of nth,

Installing automatic rewrites from: -3 length_cdr

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `nth_tail.2`.

`nth_tail.3:`

{1} $\forall (n: \text{nat}, l: \text{list}[T], i: \text{nat}):$
 $n \leq \text{length}(l) \wedge i + n < \text{length}(l) \supset i + n < \text{length}[T](l)$
 {2} $\forall (l: \text{list}[T], i: \text{nat}):$
 $n' \leq \text{length}(l) \wedge i + n' < \text{length}(l) \supset$
 $\text{nth}(\text{tail}(l, n'), i) = \text{nth}(l, n' + i)$

Hiding formulas: 2,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `nth_tail.3`.

`nth_tail.4:`

{1} $\forall (n: \text{nat}, l: \text{list}[T], i: \text{nat}):$
 $n \leq \text{length}(l) \wedge i + n < \text{length}(l) \supset i < \text{length}[T](\text{tail}(l, n))$
 {2} $\forall (l: \text{list}[T], i: \text{nat}):$
 $n' \leq \text{length}(l) \wedge i + n' < \text{length}(l) \supset$
 $\text{nth}(\text{tail}(l, n'), i) = \text{nth}(l, n' + i)$

Hiding formulas: 2,
 Repeatedly Skolemizing and flattening,
 Rewriting using length_tail, matching in *,
 Simplifying, rewriting, and recording with decision procedures,
 This completes the proof of nth_tail.4.
 Q.E.D.

C.138.18 More_List_Props.car_head_TCC1

Terse proof for car_head_TCC1.

car_head_TCC1:

$$\frac{}{\{1\} \quad \forall (l: \text{list}[T], n: \text{upto}(\text{length}[T](l))): n > 0 \supset \text{cons?}[T](\text{head}(l, n))}$$

Repeatedly Skolemizing and flattening,
 Expanding the definition of head,
 Simplifying, rewriting, and recording with decision procedures,
 This completes the proof of car_head_TCC1.
 Q.E.D.

C.138.19 More_List_Props.car_head_TCC2

Terse proof for car_head_TCC2.

car_head_TCC2:

$$\frac{}{\{1\} \quad \forall (l: \text{list}[T], n: \text{upto}(\text{length}[T](l))): n > 0 \supset \text{cons?}[T](l)}$$

Repeatedly Skolemizing and flattening,
 Expanding the definition of length,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 This completes the proof of car_head_TCC2.
 Q.E.D.

C.138.20 More_List_Props.car_head

Terse proof for car_head.

car_head:

$$\frac{}{\{1\} \quad \forall (l: \text{list}[T], n: \text{upto}(\text{length}(l))): n > 0 \supset \text{car}(\text{head}(l, n)) = \text{car}(l)}$$

Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of car_head.
 Q.E.D.

C.138.21 More_List_Props.cons_tail

Terse proof for cons_tail.

cons_tail:

$$\frac{}{\{1\} \quad \forall (l: \text{list}[T], n: \text{upto}(\text{length}(l))): n < \text{length}(l) \supset \text{cons?}(\text{tail}(l, n))}$$

C Proof scripts

Repeatedly Skolemizing and flattening,
Using lemma length_tail,
Simplifying, rewriting, and recording with decision procedures,
Expanding the definition of length,
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of cons_tail.
Q.E.D.

C.138.22 More_List_Props.car_tail_TCC1

Terse proof for car_tail_TCC1.

car_tail_TCC1:

$$\{1\} \quad \forall (l: \text{list}[T], n: \text{upto}(\text{length}[T](l))): \\ n < \text{length}(l) \supset \text{cons?}[T](\text{tail}(l, n))$$

Repeatedly Skolemizing and flattening,
Using lemma length_tail,
Simplifying, rewriting, and recording with decision procedures,
Expanding the definition of length,
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of car_tail_TCC1.
Q.E.D.

C.138.23 More_List_Props.car_tail

Terse proof for car_tail.

car_tail:

$$\{1\} \quad \forall (l: \text{list}[T], n: \text{upto}(\text{length}(l))): \\ n < \text{length}(l) \supset \text{car}(\text{tail}(l, n)) = \text{nth}(l, n)$$

Inducting on l on formula 1,
we get 3 subgoals:

car_tail.1:

$$\{1\} \quad \forall (n: \text{upto}(\text{length}(\text{null}))) : \\ n < \text{length}(\text{null}) \supset \text{car}(\text{tail}(\text{null}, n)) = \text{nth}(\text{null}, n)$$

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of car_tail.1.

car_tail.2:

$$\{1\} \quad \forall (\text{cons1_var}: T, \text{cons2_var}: \text{list}[T]): \\ (\forall (n: \text{upto}(\text{length}(\text{cons2_var}))) : \\ n < \text{length}(\text{cons2_var}) \supset \text{car}(\text{tail}(\text{cons2_var}, n)) = \text{nth}(\text{cons2_var}, n)) \\ \supset \\ (\forall (n: \text{upto}(\text{length}(\text{cons}(\text{cons1_var}, \text{cons2_var})))) : \\ n < \text{length}(\text{cons}(\text{cons1_var}, \text{cons2_var})) \supset \\ \text{car}(\text{tail}(\text{cons}(\text{cons1_var}, \text{cons2_var}), n)) = \text{nth}(\text{cons}(\text{cons1_var}, \text{cons2_var}), n))$$

Repeatedly Skolemizing and flattening,

Expanding the definition of tail,
 Expanding the definition of nth,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 Rewriting using -2, matching in *,
 Expanding the definition of length,
 Simplifying, rewriting, and recording with decision procedures,
 This completes the proof of `car_tail.2`.

`car_tail.3`:

$$\begin{array}{|l} \hline \{1\} \quad \forall (l: \text{list}[T], n: \text{upto}(\text{length}[T](l))): \\ \quad \quad n < \text{length}(l) \supset \text{cons?}[T](\text{tail}(l, n)) \\ \{2\} \quad \forall (n: \text{upto}(\text{length}(l'))): n < \text{length}(l') \supset \text{car}(\text{tail}(l', n)) = \text{nth}(l', n) \end{array}$$

Hiding formulas: 2,
 Repeatedly Skolemizing and flattening,
 Using lemma `length_tail`,
 Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of `car_tail.3`.
 Q.E.D.

C.138.24 More_List_Props.cdr_tail_TCC1

Terse proof for `cdr_tail_TCC1`.

`cdr_tail_TCC1`:

$$\begin{array}{|l} \hline \{1\} \quad \forall (l: \text{list}[T], n: \text{upto}(\text{length}[T](l))): \\ \quad \quad n < \text{length}(l) \supset n + 1 \leq \text{length}[T](l) \end{array}$$

Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of `cdr_tail_TCC1`.
 Q.E.D.

C.138.25 More_List_Props.cdr_tail

Terse proof for `cdr_tail`.

`cdr_tail`:

$$\begin{array}{|l} \hline \{1\} \quad \forall (l: \text{list}[T], n: \text{upto}(\text{length}(l))): \\ \quad \quad n < \text{length}(l) \supset \text{cdr}(\text{tail}(l, n)) = \text{tail}(l, n + 1) \end{array}$$

Inducting on l on formula 1,
 we get 4 subgoals:

`cdr_tail.1`:

$$\begin{array}{|l} \hline \{1\} \quad \forall (n: \text{upto}(\text{length}(\text{null}))) : \\ \quad \quad n < \text{length}(\text{null}) \supset \text{cdr}(\text{tail}(\text{null}, n)) = \text{tail}(\text{null}, n + 1) \end{array}$$

Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of `cdr_tail.1`.

cdr_tail.2:

$\{1\} \quad \forall (\text{cons1_var}: T, \text{cons2_var}: \text{list}[T]):$ $(\forall (n: \text{upto}(\text{length}(\text{cons2_var}))):$ $n < \text{length}(\text{cons2_var}) \supset$ $\text{cdr}(\text{tail}(\text{cons2_var}, n)) = \text{tail}(\text{cons2_var}, n + 1))$ \supset $(\forall (n: \text{upto}(\text{length}(\text{cons}(\text{cons1_var}, \text{cons2_var})))):$ $n < \text{length}(\text{cons}(\text{cons1_var}, \text{cons2_var})) \supset$ $\text{cdr}(\text{tail}(\text{cons}(\text{cons1_var}, \text{cons2_var}), n)) =$ $\text{tail}(\text{cons}(\text{cons1_var}, \text{cons2_var}), n + 1))$
--

Repeatedly Skolemizing and flattening,

Expanding the definition of tail,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 2 subgoals:

cdr_tail.2.1:

$\{-1\} \quad n' \leq \text{length}(\text{cons}(\text{cons1_var}', \text{cons2_var}'))$ $\{-2\} \quad \forall (n: \text{upto}(\text{length}(\text{cons2_var}'))):$ $n < \text{length}(\text{cons2_var}') \supset$ $\text{cdr}(\text{tail}(\text{cons2_var}', n)) = \text{tail}(\text{cons2_var}', 1 + n)$ $\{-3\} \quad n' < \text{length}(\text{cons}(\text{cons1_var}', \text{cons2_var}'))$ $\{-4\} \quad n' = 0$
$\{1\} \quad \text{cons2_var}' = \text{tail}(\text{cons2_var}', n')$

Expanding the definition of tail,

which is trivially true.

This completes the proof of cdr_tail.2.1.

cdr_tail.2.2:

$\{-1\} \quad n' \leq \text{length}(\text{cons}(\text{cons1_var}', \text{cons2_var}'))$ $\{-2\} \quad \forall (n: \text{upto}(\text{length}(\text{cons2_var}'))):$ $n < \text{length}(\text{cons2_var}') \supset$ $\text{cdr}(\text{tail}(\text{cons2_var}', n)) = \text{tail}(\text{cons2_var}', 1 + n)$ $\{-3\} \quad n' < \text{length}(\text{cons}(\text{cons1_var}', \text{cons2_var}'))$
$\{1\} \quad n' = 0$ $\{2\} \quad \text{cdr}(\text{tail}(\text{cons2_var}', n' - 1)) = \text{tail}(\text{cons2_var}', n')$

Rewriting using -2, matching in *,

Expanding the definition of length,

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of cdr_tail.2.2.

cdr_tail.3:

$\{1\} \quad \forall (l: \text{list}[T], n: \text{upto}(\text{length}[T](l))):$ $n < \text{length}(l) \supset 1 + n \leq \text{length}[T](l)$ $\{2\} \quad \forall (n: \text{upto}(\text{length}(l'))):$ $n < \text{length}(l') \supset \text{cdr}(\text{tail}(l', n)) = \text{tail}(l', n + 1)$
--

Hiding formulas: 2,

Repeatedly Skolemizing and flattening,

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of cdr_tail.3.

cdr_tail.4:

$\{1\} \quad \forall (l: \text{list}[T], n: \text{upto}(\text{length}[T](l))):$ $n < \text{length}(l) \supset \text{cons?}[T](\text{tail}(l, n))$ $\{2\} \quad \forall (n: \text{upto}(\text{length}(l'))):$ $n < \text{length}(l') \supset \text{cdr}(\text{tail}(l', n)) = \text{tail}(l', n + 1)$
--

Hiding formulas: 2,

Repeatedly Skolemizing and flattening,

Rewriting using cons_tail, matching in *,

This completes the proof of cdr_tail.4.

Q.E.D.

C.138.26 More_List_Props.every_implied

Terse proof for every_implied.

every_implied:

$\{1\} \quad \forall (l: \text{list}[T], P, Q: \text{PRED}[T]):$ $(\forall (t: T): P(t) \supset Q(t)) \wedge \text{every}(P)(l) \supset \text{every}(Q)(l)$

Inducting on l on formula 1,

we get 2 subgoals:

every_implied.1:

$\{1\} \quad \forall (P, Q: \text{PRED}[T]):$ $(\forall (t: T): P(t) \supset Q(t)) \wedge \text{every}(P)(\text{null}) \supset \text{every}(Q)(\text{null})$
--

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of every_implied.1.

every_implied.2:

$\{1\} \quad \forall (\text{cons1_var}: T, \text{cons2_var}: \text{list}[T]):$ $(\forall (P, Q: \text{PRED}[T]):$ $(\forall (t: T): P(t) \supset Q(t)) \wedge \text{every}(P)(\text{cons2_var}) \supset$ $\text{every}(Q)(\text{cons2_var}))$ \supset $(\forall (P, Q: \text{PRED}[T]):$ $(\forall (t: T): P(t) \supset Q(t)) \wedge \text{every}(P)(\text{cons}(\text{cons1_var}, \text{cons2_var})) \supset$ $\text{every}(Q)(\text{cons}(\text{cons1_var}, \text{cons2_var})))$
--

Repeatedly Skolemizing and flattening,

Expanding the definition of every,

Instantiating the top quantifier in -1 with the terms: (P!1 Q!1),

Replacing using formula -2,

Instantiating quantified variables,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of every_implied.2.

Q.E.D.

C.138.27 More_List_Props.every_conjunct_left

Terse proof for every_conjunct_left.

every_conjunct_left:

$$\frac{\{1\} \quad \forall (l: \text{list}[T], P, Q: \text{PRED}[T]):}{\text{every}(P)(l) \wedge \text{every}(Q)(l) \supset \text{every}(\lambda (t: T): P(t) \wedge Q(t))(l)}$$

Inducting on l on formula 1,

we get 2 subgoals:

every_conjunct_left.1:

$$\frac{\{1\} \quad \forall (P, Q: \text{PRED}[T]):}{\text{every}(P)(\text{null}) \wedge \text{every}(Q)(\text{null}) \supset \text{every}(\lambda (t: T): P(t) \wedge Q(t))(\text{null})}$$

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of every_conjunct_left.1.

every_conjunct_left.2:

$$\frac{\{1\} \quad \forall (\text{cons1_var}: T, \text{cons2_var}: \text{list}[T]):}{\begin{array}{l} (\forall (P, Q: \text{PRED}[T]): \\ \text{every}(P)(\text{cons2_var}) \wedge \text{every}(Q)(\text{cons2_var}) \supset \\ \text{every}(\lambda (t: T): P(t) \wedge Q(t))(\text{cons2_var})) \\ \supset \\ (\forall (P, Q: \text{PRED}[T]): \\ \text{every}(P)(\text{cons}(\text{cons1_var}, \text{cons2_var})) \wedge \text{every}(Q)(\text{cons}(\text{cons1_var}, \text{cons2_var})) \\ \supset \text{every}(\lambda (t: T): P(t) \wedge Q(t))(\text{cons}(\text{cons1_var}, \text{cons2_var}))) \end{array}}$$

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of every_conjunct_left.2.

Q.E.D.

C.138.28 More_List_Props.every_iff_forall

Terse proof for every_iff_forall.

every_iff_forall:

$$\frac{\{1\} \quad \forall (l: \text{list}[T], P: \text{PRED}[T]):}{\text{every}(P)(l) \equiv (\forall (t: T): \text{member}(t, l) \supset P(t))}$$

Inducting on l on formula 1,

we get 2 subgoals:

every_iff_forall.1:

$$\frac{\{1\} \quad \forall (P: \text{PRED}[T]):}{\text{every}(P)(\text{null}) \equiv (\forall (t: T): \text{member}(t, \text{null}) \supset P(t))}$$

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of every_iff_forall.1.

`every_iff_forall.2:`

$$\begin{array}{|l}
 \hline
 \{1\} \quad \forall (\text{cons1_var}: T, \text{cons2_var}: \text{list}[T]): \\
 \quad (\forall (P: \text{PRED}[T]): \\
 \quad \quad \text{every}(P)(\text{cons2_var}) \equiv (\forall (t: T): \text{member}(t, \text{cons2_var}) \supset P(t))) \\
 \quad \supset \\
 \quad (\forall (P: \text{PRED}[T]): \\
 \quad \quad \text{every}(P)(\text{cons}(\text{cons1_var}, \text{cons2_var})) \equiv \\
 \quad \quad (\forall (t: T): \text{member}(t, \text{cons}(\text{cons1_var}, \text{cons2_var})) \supset P(t))) \\
 \hline
 \end{array}$$

Repeatedly Skolemizing and flattening,

Instantiating quantified variables,

Expanding the definition of every,

Expanding the definition of member,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 3 subgoals:

`every_iff_forall.2.1:`

$$\begin{array}{|l}
 \{ -1 \} \quad \text{every}(P')(\text{cons2_var}') \\
 \{ -2 \} \quad \forall (t: T): \text{member}(t, \text{cons2_var}') \supset P'(t) \\
 \{ -3 \} \quad P'(\text{cons1_var}') \\
 \hline
 \{1\} \quad \forall (t: T): t = \text{cons1_var}' \vee \text{member}(t, \text{cons2_var}') \supset P'(t) \\
 \hline
 \end{array}$$

Repeatedly Skolemizing and flattening,

Instantiating quantified variables,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `every_iff_forall.2.1`.

`every_iff_forall.2.2:`

$$\begin{array}{|l}
 \{ -1 \} \quad \text{every}(P')(\text{cons2_var}') \\
 \{ -2 \} \quad \forall (t: T): \text{member}(t, \text{cons2_var}') \supset P'(t) \\
 \{ -3 \} \quad \forall (t: T): t = \text{cons1_var}' \vee \text{member}(t, \text{cons2_var}') \supset P'(t) \\
 \hline
 \{1\} \quad P'(\text{cons1_var}') \\
 \hline
 \end{array}$$

Instantiating quantified variables,

This completes the proof of `every_iff_forall.2.2`.

`every_iff_forall.2.3:`

$$\begin{array}{|l}
 \{ -1 \} \quad \forall (t: T): t = \text{cons1_var}' \vee \text{member}(t, \text{cons2_var}') \supset P'(t) \\
 \hline
 \{1\} \quad \text{every}(P')(\text{cons2_var}') \\
 \{2\} \quad \forall (t: T): \text{member}(t, \text{cons2_var}') \supset P'(t) \\
 \hline
 \end{array}$$

Repeatedly Skolemizing and flattening,

Instantiating quantified variables,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `every_iff_forall.2.3`.

Q.E.D.

C.138.29 More_List_Props.some_iff_exists

Terse proof for `some_iff_exists`.

C Proof scripts

`some_iff_exists:`

$$\frac{\{1\} \quad \forall (l: \text{list}[T], P: \text{PRED}[T]): \text{some}(P)(l) \equiv (\exists (t: T): \text{member}(t, l) \wedge P(t))}{}$$

Inducting on l on formula 1,

we get 2 subgoals:

`some_iff_exists.1:`

$$\frac{\{1\} \quad \forall (P: \text{PRED}[T]): \text{some}(P)(\text{null}) \equiv (\exists (t: T): \text{member}(t, \text{null}) \wedge P(t))}{}$$

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `some_iff_exists.1`.

`some_iff_exists.2:`

$$\frac{\{1\} \quad \forall (\text{cons1_var}: T, \text{cons2_var}: \text{list}[T]): \begin{array}{l} (\forall (P: \text{PRED}[T]): \\ \text{some}(P)(\text{cons2_var}) \equiv (\exists (t: T): \text{member}(t, \text{cons2_var}) \wedge P(t))) \\ \supset \\ (\forall (P: \text{PRED}[T]): \\ \text{some}(P)(\text{cons}(\text{cons1_var}, \text{cons2_var})) \equiv \\ (\exists (t: T): \text{member}(t, \text{cons}(\text{cons1_var}, \text{cons2_var})) \wedge P(t))) \end{array}}{}$$

Repeatedly Skolemizing and flattening,

Instantiating quantified variables,

Expanding the definition of `some`,

Expanding the definition of `member`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 3 subgoals:

`some_iff_exists.2.1:`

$$\frac{\begin{array}{l} \{-1\} \quad \text{some}(P')(\text{cons2_var}') \\ \{-2\} \quad \exists (t: T): \text{member}(t, \text{cons2_var}') \wedge P'(t) \end{array}}{\{1\} \quad \exists (t: T): (t = \text{cons1_var}' \vee \text{member}(t, \text{cons2_var}')) \wedge P'(t)}$$

Repeatedly Skolemizing and flattening,

Instantiating the top quantifier in 1 with the terms: (t!1),

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `some_iff_exists.2.1`.

`some_iff_exists.2.2:`

$$\frac{\begin{array}{l} \{-1\} \quad P'(\text{cons1_var}') \\ \{1\} \quad \text{some}(P')(\text{cons2_var}') \\ \{2\} \quad \exists (t: T): \text{member}(t, \text{cons2_var}') \wedge P'(t) \\ \{3\} \quad \exists (t: T): (t = \text{cons1_var}' \vee \text{member}(t, \text{cons2_var}')) \wedge P'(t) \end{array}}{}$$

Instantiating the top quantifier in 3 with the terms: (cons1_var!1),

which is trivially true.

This completes the proof of `some_iff_exists.2.2`.

`some_iff_exists.2.3:`

$$\frac{\begin{array}{l} \{-1\} \quad \exists (t: T): (t = \text{cons1_var}' \vee \text{member}(t, \text{cons2_var}')) \wedge P'(t) \\ \{1\} \quad \text{some}(P')(\text{cons2_var}') \\ \{2\} \quad \exists (t: T): \text{member}(t, \text{cons2_var}') \wedge P'(t) \\ \{3\} \quad P'(\text{cons1_var}') \end{array}}{}$$

Repeatedly Skolemizing and flattening,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 Instantiating quantified variables,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 This completes the proof of `some_iff_exists.2.3`.
 Q.E.D.

C.138.30 More_List_Props.list_extensionality_TCC1

Terse proof for `list_extensionality_TCC1`.

`list_extensionality_TCC1`:

$$\frac{\{1\} \quad \forall (l_1, l_2: \text{list}[T]): \quad \text{length}(l_1) = \text{length}(l_2) \supset (\forall (i: \text{below}(\text{length}[T](l_1))): i < \text{length}[T](l_2))}{}$$

Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of `list_extensionality_TCC1`.
 Q.E.D.

C.138.31 More_List_Props.list_extensionality

Terse proof for `list_extensionality`.

`list_extensionality`:

$$\frac{\{1\} \quad \forall (l_1, l_2: \text{list}[T]): \quad l_1 = l_2 \equiv (\text{length}(l_1) = \text{length}(l_2) \wedge (\forall (i: \text{below}(\text{length}(l_1))): \text{nth}(l_1, i) = \text{nth}(l_2, i)))}{}$$

Inducting on l_1 on formula 1,

we get 3 subgoals:

`list_extensionality.1`:

$$\frac{\{1\} \quad \forall (l_2: \text{list}[T]): \quad \text{null} = l_2 \equiv (\text{length}(\text{null}) = \text{length}(l_2) \wedge (\forall (i: \text{below}(\text{length}(\text{null}))) : \text{nth}(\text{null}, i) = \text{nth}(l_2, i)))}{}$$

Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of `list_extensionality.1`.

list_extensionality.2:

$\{1\} \quad \forall (\text{cons1_var}: T, \text{cons2_var}: \text{list}[T]):$ $\quad (\forall (l_2: \text{list}[T]):$ $\quad \quad \text{cons2_var} = l_2 \equiv$ $\quad \quad (\text{length}(\text{cons2_var}) = \text{length}(l_2) \wedge$ $\quad \quad (\forall (i: \text{below}(\text{length}(\text{cons2_var}))) : \text{nth}(\text{cons2_var}, i) = \text{nth}(l_2, i)))$ $\quad \supset$ $\quad (\forall (l_2: \text{list}[T]):$ $\quad \quad \text{cons}(\text{cons1_var}, \text{cons2_var}) = l_2 \equiv$ $\quad \quad (\text{length}(\text{cons}(\text{cons1_var}, \text{cons2_var})) = \text{length}(l_2) \wedge$ $\quad \quad (\forall (i: \text{below}(\text{length}(\text{cons}(\text{cons1_var}, \text{cons2_var})))) :$ $\quad \quad \quad \text{nth}(\text{cons}(\text{cons1_var}, \text{cons2_var}), i) = \text{nth}(l_2, i)))$
--

Repeatedly Skolemizing and flattening,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Expanding the definition of length,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Instantiating the top quantifier in -1 with the terms: $\text{cdr}(l'_2)$,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 2 subgoals:

list_extensionality.2.1:

$\{-1\} \quad \text{cons2_var}' = \text{cdr}(l'_2)$ $\{-2\} \quad \text{length}(\text{cons2_var}') = \text{length}(\text{cdr}(l'_2))$ $\{-3\} \quad \forall (i: \text{below}(\text{length}(\text{cons2_var}')) : \text{nth}(\text{cons2_var}', i) = \text{nth}(\text{cdr}(l'_2), i)$ $\{-4\} \quad \forall (i: \text{below}(\text{length}(\text{cons}(\text{cons1_var}', \text{cons2_var}')))) :$ $\quad \quad \text{nth}(\text{cons}(\text{cons1_var}', \text{cons2_var}'), i) = \text{nth}(l'_2, i)$
$\{1\} \quad \text{cons}(\text{cons1_var}', \text{cons2_var}') = l'_2$ $\{2\} \quad \text{null?}(l'_2)$

Applying extensionality,

Instantiating the top quantifier in -4 with the terms: 0,

we get 2 subgoals:

list_extensionality.2.1.1:

$\{-1\} \quad \text{cons2_var}' = \text{cdr}(l'_2)$ $\{-2\} \quad \text{length}(\text{cons2_var}') = \text{length}(\text{cdr}(l'_2))$ $\{-3\} \quad \forall (i: \text{below}(\text{length}(\text{cons2_var}')) : \text{nth}(\text{cons2_var}', i) = \text{nth}(\text{cdr}(l'_2), i)$ $\{-4\} \quad \text{nth}(\text{cons}(\text{cons1_var}', \text{cons2_var}'), 0) = \text{nth}(l'_2, 0)$
$\{1\} \quad \text{cons1_var}' = \text{car}(l'_2)$ $\{2\} \quad \text{null?}(l'_2)$

Expanding the definition of nth,

which is trivially true.

This completes the proof of list_extensionality.2.1.1.

list_extensionality.2.1.2:

$\{-1\} \quad \text{cons2_var}' = \text{cdr}(l'_2)$ $\{-2\} \quad \text{length}(\text{cons2_var}') = \text{length}(\text{cdr}(l'_2))$ $\{-3\} \quad \forall (i: \text{below}(\text{length}(\text{cons2_var}')) : \text{nth}(\text{cons2_var}', i) = \text{nth}(\text{cdr}(l'_2), i)$
$\{1\} \quad 0 < \text{length}[T](\text{cons}[T](\text{cons1_var}', \text{cons2_var}'))$ $\{2\} \quad \text{cons1_var}' = \text{car}(l'_2)$ $\{3\} \quad \text{null?}(l'_2)$

Expanding the definition of length,
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of list_extensionality.2.1.2.

list_extensionality.2.2:

{-1}	$\text{length}(\text{cons2_var}') = \text{length}(\text{cdr}(l'_2))$
{-2}	$\forall (i: \text{below}(\text{length}(\text{cons}(\text{cons1_var}', \text{cons2_var}')))):$ $\text{nth}(\text{cons}(\text{cons1_var}', \text{cons2_var}'), i) = \text{nth}(l'_2, i)$
{1}	$\text{cons2_var}' = \text{cdr}(l'_2)$
{2}	$\forall (i: \text{below}(\text{length}(\text{cons2_var}'))): \text{nth}(\text{cons2_var}', i) = \text{nth}(\text{cdr}(l'_2), i)$
{3}	$\text{cons}(\text{cons1_var}', \text{cons2_var}') = l'_2$
{4}	$\text{null?}(l'_2)$

Repeatedly Skolemizing and flattening,
Instantiating the top quantifier in -3 with the terms: $i' + 1$,
we get 2 subgoals:

list_extensionality.2.2.1:

{-1}	$i' < \text{length}(\text{cons2_var}')$
{-2}	$\text{length}(\text{cons2_var}') = \text{length}(\text{cdr}(l'_2))$
{-3}	$\text{nth}(\text{cons}(\text{cons1_var}', \text{cons2_var}'), i' + 1) = \text{nth}(l'_2, i' + 1)$
{1}	$\text{cons2_var}' = \text{cdr}(l'_2)$
{2}	$\text{nth}(\text{cons2_var}', i') = \text{nth}(\text{cdr}(l'_2), i')$
{3}	$\text{cons}(\text{cons1_var}', \text{cons2_var}') = l'_2$
{4}	$\text{null?}(l'_2)$

Expanding the definition of nth,
which is trivially true.
This completes the proof of list_extensionality.2.2.1.

list_extensionality.2.2.2:

{-1}	$i' < \text{length}(\text{cons2_var}')$
{-2}	$\text{length}(\text{cons2_var}') = \text{length}(\text{cdr}(l'_2))$
{1}	$1 + i' < \text{length}[T](\text{cons}[T](\text{cons1_var}', \text{cons2_var}'))$
{2}	$\text{cons2_var}' = \text{cdr}(l'_2)$
{3}	$\text{nth}(\text{cons2_var}', i') = \text{nth}(\text{cdr}(l'_2), i')$
{4}	$\text{cons}(\text{cons1_var}', \text{cons2_var}') = l'_2$
{5}	$\text{null?}(l'_2)$

Expanding the definition of length,
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of list_extensionality.2.2.2.

list_extensionality.3:

{1}	$\forall (l_1, l_2: \text{list}[T]):$ $\text{length}(l_1) = \text{length}(l_2) \supset$ $(\forall (i: \text{below}(\text{length}[T](l_1))): i < \text{length}[T](l_2))$
{2}	$\forall (l_2: \text{list}[T]):$ $l'_1 = l_2 \equiv$ $(\text{length}(l'_1) = \text{length}(l_2) \wedge$ $(\forall (i: \text{below}(\text{length}(l'_1))): \text{nth}(l'_1, i) = \text{nth}(l_2, i)))$

Hiding formulas: 2,
Repeatedly Skolemizing and flattening,
Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `list_extensionality.3`.
Q.E.D.

C.138.32 More_List_Props.list_remove_TCC1

Terse proof for `list_remove_TCC1`.

`list_remove_TCC1`:

$$\{1\} \quad \forall (x: T, l: \text{list}[T], \text{car}: T, \text{cdr}: \text{list}[T]):$$

$$x = \text{car} \wedge l = \text{cons}(\text{car}, \text{cdr}) \supset \text{length}[T](\text{cdr}) < \text{length}[T](l)$$

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `list_remove_TCC1`.
Q.E.D.

C.138.33 More_List_Props.list_remove_TCC2

Terse proof for `list_remove_TCC2`.

`list_remove_TCC2`:

$$\{1\} \quad \forall (x: T, l: \text{list}[T], \text{car}: T, \text{cdr}: \text{list}[T]):$$

$$\neg x = \text{car} \wedge l = \text{cons}(\text{car}, \text{cdr}) \supset \text{length}[T](\text{cdr}) < \text{length}[T](l)$$

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `list_remove_TCC2`.
Q.E.D.

C.138.34 More_List_Props.list_remove_commutative

Terse proof for `list_remove_commutative`.

`list_remove_commutative`:

$$\{1\} \quad \forall (x, y: T, l: \text{list}[T]):$$

$$\text{list_remove}(x, \text{list_remove}(y, l)) = \text{list_remove}(y, \text{list_remove}(x, l))$$

By induction on l , and by repeatedly rewriting and simplifying,
This completes the proof of `list_remove_commutative`.
Q.E.D.

C.138.35 More_List_Props.member_list_remove

Terse proof for `member_list_remove`.

`member_list_remove`:

$$\{1\} \quad \forall (x, y: T, l: \text{list}[T]):$$

$$\text{member}(x, \text{list_remove}(y, l)) = \text{IF } x = y \text{ THEN FALSE ELSE } \text{member}(x, l) \text{ ENDIF}$$

By induction on l , and by repeatedly rewriting and simplifying,
This completes the proof of `member_list_remove`.
Q.E.D.

C.138.36 More_List_Props.nth_member

Terse proof for nth_member.

nth_member:

$$\frac{}{\{1\} \quad \forall (x: T, l: \text{list}[T]): \quad \text{member}(x, l) \equiv (\exists (i: \text{below}(\text{length}(l))): \text{nth}(l, i) = x)}$$

Inducting on l on formula 1,

we get 2 subgoals:

nth_member.1:

$$\frac{}{\{1\} \quad \forall (x: T): \text{member}(x, \text{null}) \equiv (\exists (i: \text{below}(\text{length}(\text{null})): \text{nth}(\text{null}, i) = x)}$$

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of nth_member.1.

nth_member.2:

$$\frac{}{\{1\} \quad \forall (\text{cons1_var}: T, \text{cons2_var}: \text{list}[T]): \quad \begin{aligned} &(\forall (x: T): \\ &\quad \text{member}(x, \text{cons2_var}) \equiv \\ &\quad (\exists (i: \text{below}(\text{length}(\text{cons2_var})): \text{nth}(\text{cons2_var}, i) = x)) \\ &\quad \supset \\ &(\forall (x: T): \\ &\quad \text{member}(x, \text{cons}(\text{cons1_var}, \text{cons2_var})) \equiv \\ &\quad (\exists (i: \text{below}(\text{length}(\text{cons}(\text{cons1_var}, \text{cons2_var})))): \\ &\quad \quad \text{nth}(\text{cons}(\text{cons1_var}, \text{cons2_var}), i) = x)) \end{aligned}}$$

Repeatedly Skolemizing and flattening,

Installing automatic rewrites from: member nth length

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 3 subgoals:

nth_member.2.1:

$$\frac{\begin{array}{l} \{-1\} \quad \forall (x: T): \\ \quad \text{member}(x, \text{cons2_var}') \equiv \\ \quad (\exists (i: \text{below}(\text{length}(\text{cons2_var}')): \text{nth}(\text{cons2_var}', i) = x) \\ \{-2\} \quad x' = \text{cons1_var}' \end{array}}{\{1\} \quad \exists (i: \text{below}(\text{length}(\text{cons}(\text{cons1_var}', \text{cons2_var}')))): \quad \text{nth}(\text{cons}(\text{cons1_var}', \text{cons2_var}'), i) = x'}$$

Instantiating the top quantifier in 1 with the terms: 0,

we get 2 subgoals:

nth_member.2.1.1:

$$\frac{\begin{array}{l} \{-1\} \quad \forall (x: T): \\ \quad \text{member}(x, \text{cons2_var}') \equiv \\ \quad (\exists (i: \text{below}(\text{length}(\text{cons2_var}')): \text{nth}(\text{cons2_var}', i) = x) \\ \{-2\} \quad x' = \text{cons1_var}' \end{array}}{\{1\} \quad \text{nth}(\text{cons}(\text{cons1_var}', \text{cons2_var}'), 0) = x'}$$

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of nth_member.2.1.1.

C Proof scripts

`nth_member.2.1.2:`

$$\begin{array}{l|l}
 \{-1\} & \forall (x: T): \\
 & \text{member}(x, \text{cons2_var}') \equiv \\
 & (\exists (i: \text{below}(\text{length}(\text{cons2_var}'))): \text{nth}(\text{cons2_var}', i) = x) \\
 \{-2\} & x' = \text{cons1_var}' \\
 \hline
 \{1\} & 0 < 1 + \text{length}(\text{cons2_var}')
 \end{array}$$

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `nth_member.2.1.2`.

`nth_member.2.2:`

$$\begin{array}{l|l}
 \{-1\} & \forall (x: T): \\
 & \text{member}(x, \text{cons2_var}') \equiv \\
 & (\exists (i: \text{below}(\text{length}(\text{cons2_var}'))): \text{nth}(\text{cons2_var}', i) = x) \\
 \{-2\} & \text{member}(x', \text{cons2_var}') \\
 \hline
 \{1\} & \exists (i: \text{below}(\text{length}(\text{cons}(\text{cons1_var}', \text{cons2_var}')))): \\
 & \text{nth}(\text{cons}(\text{cons1_var}', \text{cons2_var}'), i) = x'
 \end{array}$$

Instantiating quantified variables,

Simplifying, rewriting, and recording with decision procedures,

Repeatedly Skolemizing and flattening,

Instantiating the top quantifier in 1 with the terms: $i' + 1$,

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `nth_member.2.2`.

`nth_member.2.3:`

$$\begin{array}{l|l}
 \{-1\} & \forall (x: T): \\
 & \text{member}(x, \text{cons2_var}') \equiv \\
 & (\exists (i: \text{below}(\text{length}(\text{cons2_var}'))): \text{nth}(\text{cons2_var}', i) = x) \\
 \{-2\} & \exists (i: \text{below}(\text{length}(\text{cons}(\text{cons1_var}', \text{cons2_var}')))): \\
 & \text{nth}(\text{cons}(\text{cons1_var}', \text{cons2_var}'), i) = x' \\
 \hline
 \{1\} & x' = \text{cons1_var}' \\
 \{2\} & \text{member}(x', \text{cons2_var}')
 \end{array}$$

Repeatedly Skolemizing and flattening,

Case splitting on $i!1 = 0$,

we get 2 subgoals:

`nth_member.2.3.1:`

$$\begin{array}{l|l}
 \{-1\} & i' = 0 \\
 \{-2\} & i' < \text{length}(\text{cons}(\text{cons1_var}', \text{cons2_var}')) \\
 \{-3\} & \forall (x: T): \\
 & \text{member}(x, \text{cons2_var}') \equiv \\
 & (\exists (i: \text{below}(\text{length}(\text{cons2_var}'))): \text{nth}(\text{cons2_var}', i) = x) \\
 \{-4\} & \text{nth}(\text{cons}(\text{cons1_var}', \text{cons2_var}'), i') = x' \\
 \hline
 \{1\} & x' = \text{cons1_var}' \\
 \{2\} & \text{member}(x', \text{cons2_var}')
 \end{array}$$

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `nth_member.2.3.1`.

`nth_member.2.3.2:`

$\{-1\}$	$i' < \text{length}(\text{cons}(\text{cons1_var}', \text{cons2_var}'))$
$\{-2\}$	$\forall (x: T):$ $\text{member}(x, \text{cons2_var}') \equiv$ $(\exists (i: \text{below}(\text{length}(\text{cons2_var}'))): \text{nth}(\text{cons2_var}', i) = x)$
$\{-3\}$	$\text{nth}(\text{cons}(\text{cons1_var}', \text{cons2_var}'), i') = x'$
$\{1\}$	
$\{1\}$	$i' = 0$
$\{2\}$	$x' = \text{cons1_var}'$
$\{3\}$	$\text{member}(x', \text{cons2_var}')$

Simplifying, rewriting, and recording with decision procedures,
Instantiating quantified variables,
Simplifying, rewriting, and recording with decision procedures,
Instantiating the top quantifier in 2 with the terms: $i' - 1$,
This completes the proof of `nth_member.2.3.2`.
Q.E.D.

C.138.37 More_List_Props.flatten_TCC1

Terse proof for `flatten_TCC1`.

`flatten_TCC1:`

$\{1\}$	$\forall (\text{ll}: \text{list}[\text{list}[T]], \text{car}: \text{list}[T], \text{cdr}: \text{list}[\text{list}[T]]):$ $\text{ll} = \text{cons}(\text{car}, \text{cdr}) \supset \text{length}[\text{list}[T]](\text{cdr}) < \text{length}[\text{list}[T]](\text{ll})$
---------	--

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `flatten_TCC1`.
Q.E.D.

C.138.38 More_List_Props.member_append

Terse proof for `member_append`.

`member_append:`

$\{1\}$	$\forall (x: T, l_1, l_2: \text{list}[T]):$ $\text{member}(x, \text{append}(l_1, l_2)) \equiv \text{member}(x, l_1) \vee \text{member}(x, l_2)$
---------	--

By induction on l_1 , and by repeatedly rewriting and simplifying,
This completes the proof of `member_append`.
Q.E.D.

C.138.39 More_List_Props.member_flatten

Terse proof for `member_flatten`.

`member_flatten:`

$\{1\}$	$\forall (x: T, \text{ll}: \text{list}[\text{list}[T]]):$ $\text{member}(x, \text{flatten}(\text{ll})) \equiv (\exists (l: \text{list}[T]): \text{member}(l, \text{ll}) \wedge \text{member}(x, l))$
---------	---

Inducting on ll on formula 1,
we get 2 subgoals:

C Proof scripts

`member_flatten.1:`

$$\frac{\{1\} \quad \forall (x: T): \quad \text{member}(x, \text{flatten}(\text{null})) \equiv (\exists (l: \text{list}[T]): \text{member}(l, \text{null}) \wedge \text{member}(x, l))}{}$$

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `member_flatten.1`.

`member_flatten.2:`

$$\frac{\{1\} \quad \forall (\text{cons1_var}: \text{list}[T], \text{cons2_var}: \text{list}[\text{list}[T]]): \quad \begin{array}{l} (\forall (x: T): \\ \quad \text{member}(x, \text{flatten}(\text{cons2_var})) \equiv \\ \quad (\exists (l: \text{list}[T]): \text{member}(l, \text{cons2_var}) \wedge \text{member}(x, l)) \\ \supset \\ (\forall (x: T): \\ \quad \text{member}(x, \text{flatten}(\text{cons}(\text{cons1_var}, \text{cons2_var}))) \equiv \\ \quad (\exists (l: \text{list}[T]): \text{member}(l, \text{cons}(\text{cons1_var}, \text{cons2_var})) \wedge \text{member}(x, l)) \end{array}}{}$$

Repeatedly Skolemizing and flattening,
Expanding the definition of `flatten`,
Rewriting using `member_append`, matching in `*`,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
we get 3 subgoals:

`member_flatten.2.1:`

$$\frac{\begin{array}{l} \{-1\} \quad \forall (x: T): \\ \quad \text{member}(x, \text{flatten}(\text{cons2_var}')) \equiv \\ \quad (\exists (l: \text{list}[T]): \text{member}(l, \text{cons2_var}') \wedge \text{member}(x, l)) \\ \{-2\} \quad \text{member}(x', \text{cons1_var}') \end{array}}{\{1\} \quad \exists (l: \text{list}[T]): \text{member}(l, \text{cons}(\text{cons1_var}', \text{cons2_var}')) \wedge \text{member}(x', l)}$$

Instantiating the top quantifier in 1 with the terms: `cons1_var'`,
Simplifying, rewriting, and recording with decision procedures,
Expanding the definition of `member`,
which is trivially true.

This completes the proof of `member_flatten.2.1`.

`member_flatten.2.2:`

$$\frac{\begin{array}{l} \{-1\} \quad \forall (x: T): \\ \quad \text{member}(x, \text{flatten}(\text{cons2_var}')) \equiv \\ \quad (\exists (l: \text{list}[T]): \text{member}(l, \text{cons2_var}') \wedge \text{member}(x, l)) \\ \{-2\} \quad \text{member}(x', \text{flatten}(\text{cons2_var}')) \end{array}}{\{1\} \quad \exists (l: \text{list}[T]): \text{member}(l, \text{cons}(\text{cons1_var}', \text{cons2_var}')) \wedge \text{member}(x', l)}$$

Instantiating quantified variables,
Simplifying, rewriting, and recording with decision procedures,
Repeatedly Skolemizing and flattening,
Instantiating quantified variables,
Simplifying, rewriting, and recording with decision procedures,
Expanding the definition of `member`,
which is trivially true.

This completes the proof of `member_flatten.2.2`.

member_flatten.2.3:

{-1}	$\forall (x: T):$ $\text{member}(x, \text{flatten}(\text{cons2_var}')) \equiv$ $(\exists (l: \text{list}[T]): \text{member}(l, \text{cons2_var}') \wedge \text{member}(x, l))$
{-2}	$\exists (l: \text{list}[T]): \text{member}(l, \text{cons}(\text{cons1_var}', \text{cons2_var}')) \wedge \text{member}(x', l)$
{1}	$\text{member}(x', \text{cons1_var}')$
{2}	$\text{member}(x', \text{flatten}(\text{cons2_var}'))$

Repeatedly Skolemizing and flattening,
 Expanding the definition of member,
 Simplifying, rewriting, and recording with decision procedures,
 Instantiating quantified variables,
 Simplifying, rewriting, and recording with decision procedures,
 Instantiating quantified variables,
 Simplifying, rewriting, and recording with decision procedures,
 This completes the proof of member_flatten.2.3.
 Q.E.D.

C.138.40 More_List_Props.map2_TCC1

Terse proof for map2_TCC1.

map2_TCC1:

{1}	$\forall (f: [T, T \rightarrow T], l_1, l_2: \text{list}[T], h_1: T, t_1: \text{list}[T]):$ $l_1 = \text{cons}(h_1, t_1) \supset$ $(\forall (h_2: T, t_2: \text{list}[T]):$ $l_2 = \text{cons}(h_2, t_2) \supset \text{length}[T](t_1) < \text{length}[T](l_1))$
-----	---

Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of map2_TCC1.
 Q.E.D.

C.138.41 More_List_Props.map2_property

Terse proof for map2_property.

map2_property:

{1}	$\forall (P: \text{PRED}[T], f: [T, T \rightarrow T], l_1, l_2: \text{list}[T]):$ $(\forall (t_1, t_2: T): P(f(t_1, t_2))) \supset \text{every}(P)(\text{map2}(f)(l_1, l_2))$
-----	--

Inducting on l_1 on formula 1,

we get 2 subgoals:

map2_property.1:

{1}	$\forall (P: \text{PRED}[T], f: [T, T \rightarrow T], l_2: \text{list}[T]):$ $(\forall (t_1, t_2: T): P(f(t_1, t_2))) \supset \text{every}(P)(\text{map2}(f)(\text{null}, l_2))$
-----	---

Repeatedly Skolemizing and flattening,
 Expanding the definition of every,
 Expanding the definition of map2,
 which is trivially true.
 This completes the proof of map2_property.1.

map2_property.2:

$$\begin{array}{|l}
 \{1\} \quad \forall (\text{cons1_var}: T, \text{cons2_var}: \text{list}[T]): \\
 \quad (\forall (P: \text{PRED}[T], f: [T, T \rightarrow T], l_2: \text{list}[T]): \\
 \quad \quad (\forall (t_1, t_2: T): P(f(t_1, t_2))) \supset \text{every}(P)(\text{map2}(f)(\text{cons2_var}, l_2))) \\
 \quad \supset \\
 \quad (\forall (P: \text{PRED}[T], f: [T, T \rightarrow T], l_2: \text{list}[T]): \\
 \quad \quad (\forall (t_1, t_2: T): P(f(t_1, t_2))) \supset \\
 \quad \quad \text{every}(P)(\text{map2}(f)(\text{cons}(\text{cons1_var}, \text{cons2_var}), l_2)))
 \end{array}$$

Repeatedly Skolemizing and flattening,
 Expanding the definition of map2,
 Expanding the definition of every,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 we get 2 subgoals:

map2_property.2.1:

$$\begin{array}{|l}
 \{-1\} \quad \forall (P: \text{PRED}[T], f: [T, T \rightarrow T], l_2: \text{list}[T]): \\
 \quad (\forall (t_1, t_2: T): P(f(t_1, t_2))) \supset \text{every}(P)(\text{map2}(f)(\text{cons2_var}', l_2)) \\
 \{-2\} \quad \forall (t_1, t_2: T): P'(f'(t_1, t_2)) \\
 \hline
 \{1\} \quad \text{null?}(l_2') \\
 \{2\} \quad \text{every}(P')(\text{map2}(f')(\text{cons2_var}', \text{cdr}(l_2')))
 \end{array}$$

Instantiating quantified variables,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 This completes the proof of map2_property.2.1.

map2_property.2.2:

$$\begin{array}{|l}
 \{-1\} \quad \forall (P: \text{PRED}[T], f: [T, T \rightarrow T], l_2: \text{list}[T]): \\
 \quad (\forall (t_1, t_2: T): P(f(t_1, t_2))) \supset \text{every}(P)(\text{map2}(f)(\text{cons2_var}', l_2)) \\
 \{-2\} \quad \forall (t_1, t_2: T): P'(f'(t_1, t_2)) \\
 \hline
 \{1\} \quad \text{null?}(l_2') \\
 \{2\} \quad P'(f'(\text{cons1_var}', \text{car}(l_2')))
 \end{array}$$

Instantiating quantified variables,
 This completes the proof of map2_property.2.2.
 Q.E.D.

C.139 Proofs for More_List_Props_2 (vfiasco-prelude.pvs)

C.139.1 More_List_Props_2.member_map

Terse proof for member_map.

member_map:

$$\begin{array}{|l}
 \{1\} \quad \forall (b: B, f: [A \rightarrow B], l: \text{list}[A]): \\
 \quad \text{member}(b, \text{map}(f)(l)) \equiv (\exists (a: A): \text{member}(a, l) \wedge f(a) = b)
 \end{array}$$

By induction on l , and by repeatedly rewriting and simplifying,
 This completes the proof of member_map.
 Q.E.D.

C.140 Proofs for More_More_List_Props (vfiasco-prelude.pvs)

C.140.1 More_More_List_Props.list_all_length

Terse proof for `list_all_length`.

`list_all_length`:

$$\frac{}{\{1\} \quad \forall (n: \text{nat}): \exists (l: \text{list}[T]): \text{length}(l) = n}$$

Inducting on n on formula 1,

we get 2 subgoals:

`list_all_length.1`:

$$\frac{}{\{1\} \quad \exists (l: \text{list}[T]): \text{length}(l) = 0}$$

Instantiating the top quantifier in 1 with the terms: `null`,

Expanding the definition of `length`,

which is trivially true.

This completes the proof of `list_all_length.1`.

`list_all_length.2`:

$$\frac{}{\{1\} \quad \forall j: \\ \quad (\exists (l: \text{list}[T]): \text{length}(l) = j) \supset \\ \quad (\exists (l: \text{list}[T]): \text{length}(l) = j + 1)}$$

Repeatedly Skolemizing and flattening,

Instantiating the top quantifier in 1 with the terms: `(cons(choose(LAMBDA (t: T): TRUE), !1))`,

we get 2 subgoals:

`list_all_length.2.1`:

$$\frac{\begin{array}{l} \{-1\} \quad j' \geq 0 \\ \{-2\} \quad \text{length}(l') = j' \end{array}}{\{1\} \quad \text{length}(\text{cons}(\text{choose}(\lambda (t: T): \text{TRUE}), l')) = j' + 1}$$

Expanding the definition of `length`,

which is trivially true.

This completes the proof of `list_all_length.2.1`.

`list_all_length.2.2`:

$$\frac{\begin{array}{l} \{-1\} \quad j' \geq 0 \\ \{-2\} \quad \text{length}(l') = j' \end{array}}{\{1\} \quad \text{nonempty?}[T](\lambda (t: T): \text{TRUE})}$$

Expanding the definition of `nonempty?`,

Expanding the definition of `empty?`,

Expanding the definition of `member`,

which is trivially true.

This completes the proof of `list_all_length.2.2`.

Q.E.D.

C.141 Proofs for More_Relations (vfiasco-prelude.pvs)

C.141.1 More_Relations.zorn_lr

Terse proof for zorn_lr.

zorn_lr:

$$\frac{\{1\} \quad \forall (R: \text{PRED}[[T, T]]): \text{well_founded?}(R) \supset \neg (\exists (f: [\text{nat} \rightarrow T]): \forall (n: \text{nat}): R(f(n+1), f(n)))}{}$$

Expanding the definition of well_founded?,

Repeatedly Skolemizing and flattening,

Instantiating the top quantifier in -1 with the terms: (LAMBDA (t: T): EXISTS (n: nat): f!1(n) = t),

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting, we get 2 subgoals:

zorn_lr.1:

$$\frac{\begin{array}{l} \{-1\} \quad \exists (y: (\lambda (t: T): \exists (n: \text{nat}): f'(n) = t)): \\ \quad \forall (x: (\lambda (t: T): \exists (n: \text{nat}): f'(n) = t)): (\neg R'(x, y)) \\ \{-2\} \quad \forall (n: \text{nat}): R'(f'(1+n), f'(n)) \end{array}}{}$$

Repeatedly Skolemizing and flattening,

Instantiating quantified variables,

Instantiating quantified variables,

we get 3 subgoals:

zorn_lr.1.1:

$$\frac{\begin{array}{l} \{-1\} \quad n' \geq 0 \\ \{-2\} \quad f'(n') = y' \\ \{-3\} \quad R'(f'(1+n'), f'(n')) \end{array}}{\{1\} \quad R'(f'(1+n'), y')}$$

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of zorn_lr.1.1.

zorn_lr.1.2:

$$\frac{\begin{array}{l} \{-1\} \quad n' \geq 0 \\ \{-2\} \quad f'(n') = y' \\ \{-3\} \quad R'(f'(1+n'), f'(n')) \end{array}}{\{1\} \quad \exists (n: \text{nat}): f'(n) = f'(1+n')}$$

Instantiating quantified variables,

This completes the proof of zorn_lr.1.2.

zorn_lr.1.3:

$$\frac{\begin{array}{l} \{-1\} \quad n' \geq 0 \\ \{-2\} \quad f'(n') = y' \\ \{-3\} \quad \forall (x: (\lambda (t: T): \exists (n: \text{nat}): f'(n) = t)): (\neg R'(x, y')) \\ \{-4\} \quad R'(f'(1+n'), f'(n')) \end{array}}{\{1\} \quad \exists (n: \text{nat}): f'(n) = f'(1+n')}$$

Instantiating quantified variables,

This completes the proof of zorn_lr.1.3.

zorn_lr.2:

$$\frac{\{-1\} \quad \forall (n: \text{nat}): R'(f'(1+n), f'(n))}{\{1\} \quad \exists (y: T): \exists (n: \text{nat}): f'(n) = y}$$

Instantiating the top quantifier in 1 with the terms: (f!1(0)),
 Instantiating quantified variables,
 This completes the proof of zorn_lr.2.
 Q.E.D.

C.142 Proofs for More_Sets_Lemmas (vfiasco-prelude.pvs)

C.142.1 More_Sets_Lemmas.subset_equal

Terse proof for subset_equal.

subset_equal:

$$\frac{}{\{1\} \quad \forall (a, b: \text{set}[T]): a = b \supset (a \subseteq b)}$$

Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of subset_equal.
 Q.E.D.

C.142.2 More_Sets_Lemmas.union_subset3

Terse proof for union_subset3.

union_subset3:

$$\frac{}{\{1\} \quad \forall (a, b: \text{set}[T]): (a \subseteq (b \cup a))}$$

Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of union_subset3.
 Q.E.D.

C.142.3 More_Sets_Lemmas.subset_bigger_union_left

Terse proof for subset_bigger_union_left.

subset_bigger_union_left:

$$\frac{}{\{1\} \quad \forall (a, b, c: \text{set}[T]): (a \subseteq b) \supset (a \subseteq (b \cup c))}$$

Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of subset_bigger_union_left.
 Q.E.D.

C.142.4 More_Sets_Lemmas.subset_bigger_union_right

Terse proof for subset_bigger_union_right.

subset_bigger_union_right:

$$\{1\} \quad \forall (a, b, c: \text{set}[T]): (a \subseteq c) \supset (a \subseteq (b \cup c))$$

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `subset_bigger_union_right`.
Q.E.D.

C.142.5 More_Sets_Lemmas.union_empty_left

Terse proof for `union_empty_left`.

union_empty_left:

$$\{1\} \quad \forall (a: \text{set}[T]): (\text{emptyset} \cup a) = a$$

Repeatedly Skolemizing and flattening,
Rewriting using `union_commutative`, matching in *,
Rewriting using `union_empty`, matching in *,
This completes the proof of `union_empty_left`.
Q.E.D.

C.142.6 More_Sets_Lemmas.union_left

Terse proof for `union_left`.

union_left:

$$\{1\} \quad \forall (a, b: \text{set}[T], e: T): a(e) \supset \text{union}(a, b)(e)$$

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `union_left`.
Q.E.D.

C.142.7 More_Sets_Lemmas.union_right

Terse proof for `union_right`.

union_right:

$$\{1\} \quad \forall (a, b: \text{set}[T], e: T): b(e) \supset \text{union}(a, b)(e)$$

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `union_right`.
Q.E.D.

C.142.8 More_Sets_Lemmas.subset_of_differences

Terse proof for `subset_of_differences`.

subset_of_differences:

$$\{1\} \quad \forall (a, b, c: \text{set}[T]): (a \subseteq b) \supset ((c \setminus b) \subseteq (c \setminus a))$$

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `subset_of_differences`.
 Q.E.D.

C.142.9 More_Sets_Lemmas.difference_disjoint_3

Terse proof for `difference_disjoint_3`.
`difference_disjoint_3`:

$$\{1\} \quad \forall (a, b: \text{set}[T]): \text{disjoint?}((b \setminus a), a)$$

Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of `difference_disjoint_3`.
 Q.E.D.

C.142.10 More_Sets_Lemmas.subset_singleton

Terse proof for `subset_singleton`.
`subset_singleton`:

$$\{1\} \quad \forall (a: \text{set}[T], e: T): (\text{singleton}(e) \subseteq a) = a(e)$$

Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of `subset_singleton`.
 Q.E.D.

C.142.11 More_Sets_Lemmas.subset_member

Terse proof for `subset_member`.
`subset_member`:

$$\{1\} \quad \forall (a, b: \text{set}[T], e: T): a(e) \wedge (a \subseteq b) \supset b(e)$$

Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of `subset_member`.
 Q.E.D.

C.142.12 More_Sets_Lemmas.disjoint_symmetric

Terse proof for `disjoint_symmetric`.
`disjoint_symmetric`:

$$\{1\} \quad \forall (a, b: \text{set}[T]): \text{disjoint?}(a, b) = \text{disjoint?}(b, a)$$

Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of `disjoint_symmetric`.
 Q.E.D.

C.142.13 More_Sets_Lemmas.disjoint_mono

Terse proof for disjoint_mono.

disjoint_mono:

$$\{1\} \quad \forall (a_1, a_2, b_1, b_2: \text{set}[T]): \\ (a_1 \subseteq a_2) \wedge (b_1 \subseteq b_2) \wedge \text{disjoint?}(a_2, b_2) \supset \text{disjoint?}(a_1, b_1)$$

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of disjoint_mono.

Q.E.D.

C.142.14 More_Sets_Lemmas.disjoint_subset_difference

Terse proof for disjoint_subset_difference.

disjoint_subset_difference:

$$\{1\} \quad \forall (a, b, c: \text{set}[T]): (a \subseteq b) \wedge \text{disjoint?}(a, c) \supset (a \subseteq (b \setminus c))$$

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of disjoint_subset_difference.

Q.E.D.

C.143 Proofs for Number_Props (vfiasco-prelude.pvs)

C.143.1 Number_Props.number_split

Terse proof for number_split.

number_split:

$$\{1\} \quad \forall (x: \text{int}, a: \text{posnat}): x = a \times \text{ndiv}(x, a) + \text{rem}(a)(x)$$

Repeatedly Skolemizing and flattening,

Adding type constraints for ndiv(x!1, a!1),

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of number_split.

Q.E.D.

C.143.2 Number_Props.number_split_less

Terse proof for number_split_less.

number_split_less:

$$\{1\} \quad \forall (a, b, p, q: \text{nat}): a < p \supset (a + b \times p < p \times q \equiv b < q)$$

Repeatedly Skolemizing and flattening,

Applying both_sides_times_pos_le1

Instantiating the top quantifier in -1 with the terms: p' , $b' + 1$, q' ,

we get 2 subgoals:

number_split_less.1:

{-1}	$b' + 1 \times p' \leq q' \times p' \equiv b' + 1 \leq q'$
{-2}	$a' \geq 0$
{-3}	$b' \geq 0$
{-4}	$p' \geq 0$
{-5}	$q' \geq 0$
{-6}	$a' < p'$
{1}	
	$(a' + b' \times p' < p' \times q' \equiv b' < q')$

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Applying both_sides_times_pos_lt1

Instantiating the top quantifier in -1 with the terms: p', b', q' ,

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of number_split_less.1.

number_split_less.2:

{-1}	$a' \geq 0$
{-2}	$b' \geq 0$
{-3}	$p' \geq 0$
{-4}	$q' \geq 0$
{-5}	$a' < p'$
{1}	
	$p' > 0$
{2}	$(a' + b' \times p' < p' \times q' \equiv b' < q')$

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of number_split_less.2.

Q.E.D.

C.143.3 Number_Props.bit_split_less

Terse proof for bit_split_less.

bit_split_less:

{1}	
	$\forall (q, r, j, k: \text{nat}):$
	$r < \text{expt}(2, k) \supset$
	$(q \times \text{expt}(2, k) + r < \text{expt}(2, k + j) \equiv q < \text{expt}(2, j))$

Repeatedly Skolemizing and flattening,

Applying number_split_less

Instantiating the top quantifier in -1 with the terms: $r', q', \text{expt}(2, k'), \text{expt}(2, j')$,

Rewriting using expt_plus_aux, matching in *,

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of bit_split_less.

Q.E.D.

C.143.4 Number_Props.rem_def_pos

Terse proof for rem_def_pos.

rem_def_pos:

{1}	
	$\forall (b: \text{posnat}, x: \text{nat}):$
	$\forall (r: \text{mod}(b)): \text{rem}(b)(x) = r \equiv (\exists (q: \text{nat}): x = b \times q + r)$

C Proof scripts

Repeatedly Skolemizing and flattening,
Using lemma `rem_def`,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
we get 2 subgoals:

`rem_def_pos.1`:

$$\begin{array}{l|l} \{-1\} & \text{rem}(b')(x') = r' \\ \{-2\} & \exists q: x' = b' \times q + r' \\ \{-3\} & r' < b' \\ \{-4\} & b' > 0 \\ \{-5\} & x' \geq 0 \\ \hline \{1\} & \exists (q: \text{nat}): x' = b' \times q + r' \end{array}$$

Repeatedly Skolemizing and flattening,
Instantiating quantified variables,
Using lemma `both_sides_times_pos_le2`,
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `rem_def_pos.1`.

`rem_def_pos.2`:

$$\begin{array}{l|l} \{-1\} & r' < b' \\ \{-2\} & b' > 0 \\ \{-3\} & x' \geq 0 \\ \{-4\} & \exists (q: \text{nat}): x' = b' \times q + r' \\ \hline \{1\} & \text{rem}(b')(x') = r' \\ \{2\} & \exists q: x' = b' \times q + r' \end{array}$$

Repeatedly Skolemizing and flattening,
Instantiating quantified variables,
This completes the proof of `rem_def_pos.2`.
Q.E.D.

C.143.5 Number_Props.rem_rem

Terse proof for `rem_rem`.

`rem_rem`:

$$\begin{array}{l|l} \{1\} & \forall (a, b: \text{posnat}, x: \text{int}): \text{divides}(a, b) \supset \text{rem}(a)(\text{rem}(b)(x)) = \text{rem}(a)(x) \end{array}$$

Repeatedly Skolemizing and flattening,
Applying `rem_def`
Instantiating the top quantifier in -1 with the terms: a' , x' , $\text{rem}(a')(\text{rem}(b')(x'))$,
Simplifying, rewriting, and recording with decision procedures,
Hiding formulas: 2,
Adding type constraints for $\text{rem}(a!1)(\text{rem}(b!1)(x!1))$,
Adding type constraints for $\text{rem}(b!1)(x!1)$,
Expanding the definition of `divides`,
Repeatedly Skolemizing and flattening,
Replacing using formula -7,
Instantiating the top quantifier in 1 with the terms: $q'' + x'' \times q'$,
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `rem_rem`.
Q.E.D.

C.143.6 Number_Props.divides_times

Terse proof for divides_times.

divides_times:

$$\frac{\{1\} \quad \forall (i: \text{posnat}, j, k: \text{int}):}{\text{divides}(i \times j, k) \equiv \text{divides}(i, k) \wedge \text{divides}(j, \text{ndiv}(k, i))}$$

Repeatedly Skolemizing and flattening,

Splitting conjunctions,

we get 2 subgoals:

divides_times.1:

$$\frac{\begin{array}{l} \{-1\} \quad i' > 0 \\ \{-2\} \quad \text{integer_pred}(j') \\ \{-3\} \quad \text{integer_pred}(k') \end{array}}{\{1\} \quad \text{divides}(i' \times j', k') \supset \text{divides}(i', k') \wedge \text{divides}(j', \text{ndiv}(k', i'))}$$

Case splitting on divides(i!1, k!1),

we get 2 subgoals:

divides_times.1.1:

$$\frac{\begin{array}{l} \{-1\} \quad \text{divides}(i', k') \\ \{-2\} \quad i' > 0 \\ \{-3\} \quad \text{integer_pred}(j') \\ \{-4\} \quad \text{integer_pred}(k') \end{array}}{\{1\} \quad \text{divides}(i' \times j', k') \supset \text{divides}(i', k') \wedge \text{divides}(j', \text{ndiv}(k', i'))}$$

Applying propositional simplification,

Adding type constraints for ndiv(k!1, i!1),

Case splitting on rem(i!1)(k!1) = 0,

we get 2 subgoals:

divides_times.1.1.1:

$$\frac{\begin{array}{l} \{-1\} \quad \text{rem}(i')(k') = 0 \\ \{-2\} \quad k' = \text{rem}(i')(k') + i' \times \text{ndiv}(k', i') \\ \{-3\} \quad \text{divides}(i', k') \\ \{-4\} \quad \text{divides}(i' \times j', k') \\ \{-5\} \quad i' > 0 \\ \{-6\} \quad \text{integer_pred}(j') \\ \{-7\} \quad \text{integer_pred}(k') \end{array}}{\{1\} \quad \text{divides}(j', \text{ndiv}(k', i'))}$$

Replacing using formula -1,

Hiding formulas: -1, -3,

Simplifying, rewriting, and recording with decision procedures,

Expanding the definition of divides,

Repeatedly Skolemizing and flattening,

Instantiating the top quantifier in 1 with the terms: x' ,

Applying both_sides_times1

Instantiating the top quantifier in -1 with the terms: i' , $\text{ndiv}(k', i')$, $j' \times x'$,

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of divides_times.1.1.1.

C Proof scripts

`divides_times.1.1.2:`

{-1}	$k' = \text{rem}(i')(k') + i' \times \text{ndiv}(k', i')$
{-2}	$\text{divides}(i', k')$
{-3}	$\text{divides}(i' \times j', k')$
{-4}	$i' > 0$
{-5}	$\text{integer_pred}(j')$
{-6}	$\text{integer_pred}(k')$
{1}	$\text{rem}(i')(k') = 0$
{2}	$\text{divides}(j', \text{ndiv}(k', i'))$

Rewriting using `rem_def2`, matching in `*`,

This completes the proof of `divides_times.1.1.2`.

`divides_times.1.2:`

{-1}	$i' > 0$
{-2}	$\text{integer_pred}(j')$
{-3}	$\text{integer_pred}(k')$
{1}	$\text{divides}(i', k')$
{2}	$\text{divides}(i' \times j', k') \supset \text{divides}(i', k') \wedge \text{divides}(j', \text{ndiv}(k', i'))$

Simplifying, rewriting, and recording with decision procedures,

Expanding the definition of `divides`,

Repeatedly Skolemizing and flattening,

Instantiating the top quantifier in 1 with the terms: $x' \times j'$,

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `divides_times.1.2`.

`divides_times.2:`

{-1}	$i' > 0$
{-2}	$\text{integer_pred}(j')$
{-3}	$\text{integer_pred}(k')$
{1}	$\text{divides}(i', k') \wedge \text{divides}(j', \text{ndiv}(k', i')) \supset \text{divides}(i' \times j', k')$

Applying disjunctive simplification to flatten sequent,

Adding type constraints for `ndiv(k!1, i!1)`,

Case splitting on `rem(i!1)(k!1) = 0`,

we get 2 subgoals:

`divides_times.2.1:`

{-1}	$\text{rem}(i')(k') = 0$
{-2}	$k' = \text{rem}(i')(k') + i' \times \text{ndiv}(k', i')$
{-3}	$\text{divides}(i', k')$
{-4}	$\text{divides}(j', \text{ndiv}(k', i'))$
{-5}	$i' > 0$
{-6}	$\text{integer_pred}(j')$
{-7}	$\text{integer_pred}(k')$
{1}	$\text{divides}(i' \times j', k')$

Replacing using formula -1,

Hiding formulas: -1, -3,

Simplifying, rewriting, and recording with decision procedures,

Expanding the definition of `divides`,

Repeatedly Skolemizing and flattening,

Instantiating the top quantifier in 1 with the terms: x' ,

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `divides_times.2.1`.

`divides_times.2.2`:

{-1}	$k' = \text{rem}(i')(k') + i' \times \text{ndiv}(k', i')$
{-2}	$\text{divides}(i', k')$
{-3}	$\text{divides}(j', \text{ndiv}(k', i'))$
{-4}	$i' > 0$
{-5}	$\text{integer_pred}(j')$
{-6}	$\text{integer_pred}(k')$
{1}	$\text{rem}(i')(k') = 0$
{2}	$\text{divides}(i' \times j', k')$

Rewriting using `rem_def2`, matching in `*`,

This completes the proof of `divides_times.2.2`.

Q.E.D.

C.143.7 Number_Props.ndiv_floor

Terse proof for `ndiv_floor`.

`ndiv_floor`:

{1}	$\forall (a: \text{int}, b: \text{posnat}): \text{ndiv}(a, b) = \text{floor}(a/b)$
-----	--

Repeatedly Skolemizing and flattening,

Adding type constraints for `ndiv(a!1, b!1)`,

Using lemma `rem_floor`,

Applying `unique_quotient`

Instantiating the top quantifier in -1 with the terms: b' , $\text{ndiv}(a', b')$, $\text{floor}(a'/b')$, $\text{rem}(b')(a')$, $\text{rem}(b')(a')$,

Simplifying, rewriting, and recording with decision procedures,

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `ndiv_floor`.

Q.E.D.

C.143.8 Number_Props.ndiv_self

Terse proof for `ndiv_self`.

`ndiv_self`:

{1}	$\forall (a: \text{posnat}): \text{ndiv}(a, a) = 1$
-----	---

Repeatedly Skolemizing and flattening,

Adding type constraints for `ndiv(a!1, a!1)`,

Rewriting using `rem_self`, matching in `*`,

Simplifying, rewriting, and recording with decision procedures,

Applying `both_sides_times2`

Instantiating the top quantifier in -1 with the terms: a' , 1, $\text{ndiv}(a', a')$,

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `ndiv_self`.

Q.E.D.

C.143.9 Number_Props.ndiv_plus_1

Terse proof for `ndiv_plus_1`.

`ndiv_plus_1`:

<pre>{1} $\forall (a, b: \text{int}, c: \text{posnat}):$ $\text{divides}(c, a) \supset \text{ndiv}(a + b, c) = \text{ndiv}(a, c) + \text{ndiv}(b, c)$</pre>
--

Repeatedly Skolemizing and flattening,

Applying `rem_def2`

Instantiating the top quantifier in -1 with the terms: c' , a' , 0,

Simplifying, rewriting, and recording with decision procedures,

Adding type constraints for `ndiv(a!1 + b!1, c!1)`,

Applying `rem_sum1`

Instantiating the top quantifier in -1 with the terms: c' , a' , b' ,

Replacing using formula -3,

Simplifying, rewriting, and recording with decision procedures,

Replacing using formula -1,

Hiding formulas: -1,

Adding type constraints for `ndiv(a!1, c!1)`,

Adding type constraints for `ndiv(b!1, c!1)`,

Case splitting on $c!1 * (\text{ndiv}(a!1, c!1) + \text{ndiv}(b!1, c!1)) = c!1 * \text{ndiv}(a!1 + b!1, c!1)$,

we get 2 subgoals:

`ndiv_plus_1.1`:

<pre>{-1} $c' \times (\text{ndiv}(a', c') + \text{ndiv}(b', c')) = c' \times \text{ndiv}(a' + b', c')$ {-2} $b' = \text{rem}(c')(b') + c' \times \text{ndiv}(b', c')$ {-3} $a' = \text{rem}(c')(a') + c' \times \text{ndiv}(a', c')$ {-4} $a' + b' = \text{rem}(c')(b') + c' \times \text{ndiv}(a' + b', c')$ {-5} $\text{rem}(c')(a') = 0$ {-6} $\text{integer_pred}(a')$ {-7} $\text{integer_pred}(b')$ {-8} $c' > 0$ {-9} $\text{divides}(c', a')$</pre>	<pre>{1} $\text{ndiv}(a' + b', c') = \text{ndiv}(a', c') + \text{ndiv}(b', c')$</pre>
--	---

Applying `both_sides_times2`

Instantiating the top quantifier in -1 with the terms: c' , $(\text{ndiv}(a', c') + \text{ndiv}(b', c'))$, $\text{ndiv}(a' + b', c')$,

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `ndiv_plus_1.1`.

`ndiv_plus_1.2`:

<pre>{-1} $b' = \text{rem}(c')(b') + c' \times \text{ndiv}(b', c')$ {-2} $a' = \text{rem}(c')(a') + c' \times \text{ndiv}(a', c')$ {-3} $a' + b' = \text{rem}(c')(b') + c' \times \text{ndiv}(a' + b', c')$ {-4} $\text{rem}(c')(a') = 0$ {-5} $\text{integer_pred}(a')$ {-6} $\text{integer_pred}(b')$ {-7} $c' > 0$ {-8} $\text{divides}(c', a')$</pre>	<pre>{1} $c' \times (\text{ndiv}(a', c') + \text{ndiv}(b', c')) = c' \times \text{ndiv}(a' + b', c')$ {2} $\text{ndiv}(a' + b', c') = \text{ndiv}(a', c') + \text{ndiv}(b', c')$</pre>
--	--

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `ndiv_plus_1.2`.

Q.E.D.

C.143.10 Number_Props.ndiv_plus_2

Terse proof for `ndiv_plus_2`.

`ndiv_plus_2`:

$$\frac{\{1\} \quad \forall (a, b: \text{int}, c: \text{posnat}): \quad \text{divides}(c, b) \supset \text{ndiv}(a + b, c) = \text{ndiv}(a, c) + \text{ndiv}(b, c)}{\quad}$$

Repeatedly Skolemizing and flattening,

Using lemma `ndiv_plus_1`,

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `ndiv_plus_2`.

Q.E.D.

C.143.11 Number_Props.ndiv_plus_mod

Terse proof for `ndiv_plus_mod`.

`ndiv_plus_mod`:

$$\frac{\{1\} \quad \forall (a: \text{int}, c: \text{posnat}, b: \text{mod}(c)): \quad \text{divides}(c, a) \supset \text{ndiv}(a + b, c) = \text{ndiv}(a, c)}{\quad}$$

Repeatedly Skolemizing and flattening,

Adding type constraints for `ndiv(a!1, c!1)`,

Case splitting on `rem(c!1)(a!1) = 0`,

we get 2 subgoals:

`ndiv_plus_mod.1`:

$$\frac{\begin{array}{l} \{-1\} \quad \text{rem}(c')(a') = 0 \\ \{-2\} \quad a' = \text{rem}(c')(a') + c' \times \text{ndiv}(a', c') \\ \{-3\} \quad \text{integer_pred}(a') \\ \{-4\} \quad c' > 0 \\ \{-5\} \quad b' < c' \\ \{-6\} \quad \text{divides}(c', a') \end{array}}{\{1\} \quad \text{ndiv}(a' + b', c') = \text{ndiv}(a', c')}$$

Replacing using formula -1,

Hiding formulas: -1,

Simplifying, rewriting, and recording with decision procedures,

Adding type constraints for `ndiv(a!1 + b!1, c!1)`,

Case splitting on `rem(c!1)(a!1 + b!1) = b!1`,

we get 2 subgoals:

C Proof scripts

`ndiv_plus_mod.1.1:`

{-1}	$\text{rem}(c')(a' + b') = b'$
{-2}	$a' + b' = \text{rem}(c')(a' + b') + c' \times \text{ndiv}(a' + b', c')$
{-3}	$a' = c' \times \text{ndiv}(a', c')$
{-4}	$\text{integer_pred}(a')$
{-5}	$c' > 0$
{-6}	$b' < c'$
{-7}	$\text{divides}(c', a')$
{1}	$\text{ndiv}(a' + b', c') = \text{ndiv}(a', c')$

Replacing using formula -1,

Hiding formulas: -1,

Simplifying, rewriting, and recording with decision procedures,

Applying `both_sides_times2`

Instantiating the top quantifier in -1 with the terms: c' , $\text{ndiv}(a' + b', c')$, $\text{ndiv}(a', c')$,

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `ndiv_plus_mod.1.1`.

`ndiv_plus_mod.1.2:`

{-1}	$a' + b' = \text{rem}(c')(a' + b') + c' \times \text{ndiv}(a' + b', c')$
{-2}	$a' = c' \times \text{ndiv}(a', c')$
{-3}	$\text{integer_pred}(a')$
{-4}	$c' > 0$
{-5}	$b' < c'$
{-6}	$\text{divides}(c', a')$
{1}	$\text{rem}(c')(a' + b') = b'$
{2}	$\text{ndiv}(a' + b', c') = \text{ndiv}(a', c')$

Hiding formulas: -1, -2, 2,

Rewriting using `rem_def`, matching in *,

Expanding the definition of `divides`,

which is trivially true.

This completes the proof of `ndiv_plus_mod.1.2`.

`ndiv_plus_mod.2:`

{-1}	$a' = \text{rem}(c')(a') + c' \times \text{ndiv}(a', c')$
{-2}	$\text{integer_pred}(a')$
{-3}	$c' > 0$
{-4}	$b' < c'$
{-5}	$\text{divides}(c', a')$
{1}	$\text{rem}(c')(a') = 0$
{2}	$\text{ndiv}(a' + b', c') = \text{ndiv}(a', c')$

Rewriting using `rem_def2`, matching in *,

This completes the proof of `ndiv_plus_mod.2`.

Q.E.D.

C.143.12 `Number_Props.ndiv_plus_mod_2`

Terse proof for `ndiv_plus_mod_2`.

`ndiv_plus_mod_2:`

$$\frac{\{1\} \quad \forall (a: \text{int}, c: \text{posnat}, b_1, b_2: \text{mod}(c)):\quad \text{divides}(c, a) \supset \text{ndiv}(a + b_1, c) = \text{ndiv}(a + b_2, c)}{\quad}$$

Repeatedly Skolemizing and flattening,

Applying `ndiv_plus_mod`

Instantiating (with copying) the top quantifier in -1 with the terms: a', c', b'_1 ,

Instantiating the top quantifier in -1 with the terms: a', c', b'_2 ,

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `ndiv_plus_mod_2`.

Q.E.D.

C.143.13 Number_Props.ndiv_minus_2

Terse proof for `ndiv_minus_2`.

`ndiv_minus_2:`

$$\frac{\{1\} \quad \forall (a, b: \text{int}, c: \text{posnat}): \quad \text{divides}(c, b) \supset \text{ndiv}(a - b, c) = \text{ndiv}(a, c) - \text{ndiv}(b, c)}{\quad}$$

Repeatedly Skolemizing and flattening,

Applying `ndiv_plus_2`

Instantiating the top quantifier in -1 with the terms: $a', -b', c'$,

Case splitting on `divides(c!1, -b!1)`,

we get 2 subgoals:

`ndiv_minus_2.1:`

$$\frac{\begin{array}{l} \{-1\} \quad \text{divides}(c', -b') \\ \{-2\} \quad \text{divides}(c', -b') \supset \\ \quad \text{ndiv}(a' + -b', c') = \text{ndiv}(a', c') + \text{ndiv}(-b', c') \\ \{-3\} \quad \text{integer_pred}(a') \\ \{-4\} \quad \text{integer_pred}(b') \\ \{-5\} \quad c' > 0 \\ \{-6\} \quad \text{divides}(c', b') \end{array}}{\{1\} \quad \text{ndiv}(a' - b', c') = \text{ndiv}(a', c') - \text{ndiv}(b', c')}$$

Simplifying, rewriting, and recording with decision procedures,

Case splitting on `-ndiv(b!1, c!1) = ndiv(-b!1, c!1)`,

we get 2 subgoals:

`ndiv_minus_2.1.1:`

$$\frac{\begin{array}{l} \{-1\} \quad -\text{ndiv}(b', c') = \text{ndiv}(-b', c') \\ \{-2\} \quad \text{divides}(c', -b') \\ \{-3\} \quad \text{ndiv}(-b' + a', c') = \text{ndiv}(-b', c') + \text{ndiv}(a', c') \\ \{-4\} \quad \text{integer_pred}(a') \\ \{-5\} \quad \text{integer_pred}(b') \\ \{-6\} \quad c' > 0 \\ \{-7\} \quad \text{divides}(c', b') \end{array}}{\{1\} \quad \text{ndiv}(a' - b', c') = \text{ndiv}(a', c') - \text{ndiv}(b', c')}$$

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `ndiv_minus_2.1.1`.

C Proof scripts

ndiv_minus_2.1.2:

{-1}	$\text{divides}(c', -b')$
{-2}	$\text{ndiv}(-b' + a', c') = \text{ndiv}(-b', c') + \text{ndiv}(a', c')$
{-3}	$\text{integer_pred}(a')$
{-4}	$\text{integer_pred}(b')$
{-5}	$c' > 0$
{-6}	$\text{divides}(c', b')$
{1}	$-\text{ndiv}(b', c') = \text{ndiv}(-b', c')$
{2}	$\text{ndiv}(a' - b', c') = \text{ndiv}(a', c') - \text{ndiv}(b', c')$

Hiding formulas: -2, 2,

Adding type constraints for $\text{ndiv}(b!1, c!1)$,

Adding type constraints for $\text{ndiv}(-b!1, c!1)$,

Case splitting on $\text{rem}(c!1)(-b!1) = 0$,

we get 2 subgoals:

ndiv_minus_2.1.2.1:

{-1}	$\text{rem}(c')(-b') = 0$
{-2}	$-b' = \text{rem}(c')(-b') + c' \times \text{ndiv}(-b', c')$
{-3}	$b' = \text{rem}(c')(b') + c' \times \text{ndiv}(b', c')$
{-4}	$\text{divides}(c', -b')$
{-5}	$\text{integer_pred}(a')$
{-6}	$\text{integer_pred}(b')$
{-7}	$c' > 0$
{-8}	$\text{divides}(c', b')$
{1}	$-\text{ndiv}(b', c') = \text{ndiv}(-b', c')$

Case splitting on $\text{rem}(c!1)(b!1) = 0$,

we get 2 subgoals:

ndiv_minus_2.1.2.1.1:

{-1}	$\text{rem}(c')(b') = 0$
{-2}	$\text{rem}(c')(-b') = 0$
{-3}	$-b' = \text{rem}(c')(-b') + c' \times \text{ndiv}(-b', c')$
{-4}	$b' = \text{rem}(c')(b') + c' \times \text{ndiv}(b', c')$
{-5}	$\text{divides}(c', -b')$
{-6}	$\text{integer_pred}(a')$
{-7}	$\text{integer_pred}(b')$
{-8}	$c' > 0$
{-9}	$\text{divides}(c', b')$
{1}	$-\text{ndiv}(b', c') = \text{ndiv}(-b', c')$

Replacing using formula -1,

Replacing using formula -2,

Hiding formulas: -1, -2,

Simplifying, rewriting, and recording with decision procedures,

Applying `both_sides_times2`

Instantiating the top quantifier in -1 with the terms: c' , $-\text{ndiv}(b', c')$, $\text{ndiv}(-b', c')$,

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `ndiv_minus_2.1.2.1.1`.

ndiv_minus_2.1.2.1.2:

{-1}	$\text{rem}(c')(-b') = 0$
{-2}	$-b' = \text{rem}(c')(-b') + c' \times \text{ndiv}(-b', c')$
{-3}	$b' = \text{rem}(c')(b') + c' \times \text{ndiv}(b', c')$
{-4}	$\text{divides}(c', -b')$
{-5}	$\text{integer_pred}(a')$
{-6}	$\text{integer_pred}(b')$
{-7}	$c' > 0$
{-8}	$\text{divides}(c', b')$
{1}	$\text{rem}(c')(b') = 0$
{2}	$-\text{ndiv}(b', c') = \text{ndiv}(-b', c')$

Rewriting using rem_def2, matching in *,

This completes the proof of ndiv_minus_2.1.2.1.2.

ndiv_minus_2.1.2.2:

{-1}	$-b' = \text{rem}(c')(-b') + c' \times \text{ndiv}(-b', c')$
{-2}	$b' = \text{rem}(c')(b') + c' \times \text{ndiv}(b', c')$
{-3}	$\text{divides}(c', -b')$
{-4}	$\text{integer_pred}(a')$
{-5}	$\text{integer_pred}(b')$
{-6}	$c' > 0$
{-7}	$\text{divides}(c', b')$
{1}	$\text{rem}(c')(-b') = 0$
{2}	$-\text{ndiv}(b', c') = \text{ndiv}(-b', c')$

Rewriting using rem_def2, matching in *,

This completes the proof of ndiv_minus_2.1.2.2.

ndiv_minus_2.2:

{-1}	$\text{divides}(c', -b') \supset$ $\text{ndiv}(a' - b', c') = \text{ndiv}(a', c') + \text{ndiv}(-b', c')$
{-2}	$\text{integer_pred}(a')$
{-3}	$\text{integer_pred}(b')$
{-4}	$c' > 0$
{-5}	$\text{divides}(c', b')$
{1}	$\text{divides}(c', -b')$
{2}	$\text{ndiv}(a' - b', c') = \text{ndiv}(a', c') - \text{ndiv}(b', c')$

Hiding formulas: -1, 2,

Expanding the definition of divides,

Repeatedly Skolemizing and flattening,

Instantiating the top quantifier in 1 with the terms: $-x'$,

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of ndiv_minus_2.2.

Q.E.D.

C.143.14 Number_Props.ndiv_times_divident_1

Terse proof for ndiv_times_divident_1.

ndiv_times_divident_1:

{1}	$\forall (a, b: \text{int}, c: \text{posnat}):$ $\text{divides}(c, a) \supset \text{ndiv}(a \times b, c) = \text{ndiv}(a, c) \times b$
-----	---

C Proof scripts

Repeatedly Skolemizing and flattening,
 Adding type constraints for $\text{ndiv}(a!1 * b!1, c!1)$,
 Adding type constraints for $\text{ndiv}(a!1, c!1)$,
 Case splitting on $\text{rem}(c!1)(a!1) = 0$,
 we get 2 subgoals:

ndiv_times_divident_1.1:

{-1}	$\text{rem}(c')(a') = 0$
{-2}	$a' = \text{rem}(c')(a') + c' \times \text{ndiv}(a', c')$
{-3}	$a' \times b' = \text{rem}(c')(a' \times b') + c' \times \text{ndiv}(a' \times b', c')$
{-4}	$\text{integer_pred}(a')$
{-5}	$\text{integer_pred}(b')$
{-6}	$c' > 0$
{-7}	$\text{divides}(c', a')$
{1}	
$\text{ndiv}(a' \times b', c') = \text{ndiv}(a', c') \times b'$	

Replacing using formula -1,
 Hiding formulas: -1,
 Case splitting on $\text{rem}(c!1)(a!1 * b!1) = 0$,
 we get 2 subgoals:

ndiv_times_divident_1.1.1:

{-1}	$\text{rem}(c')(a' \times b') = 0$
{-2}	$a' = 0 + c' \times \text{ndiv}(a', c')$
{-3}	$a' \times b' = \text{rem}(c')(a' \times b') + c' \times \text{ndiv}(a' \times b', c')$
{-4}	$\text{integer_pred}(a')$
{-5}	$\text{integer_pred}(b')$
{-6}	$c' > 0$
{-7}	$\text{divides}(c', a')$
{1}	
$\text{ndiv}(a' \times b', c') = \text{ndiv}(a', c') \times b'$	

Replacing using formula -1,
 Hiding formulas: -1,
 Simplifying, rewriting, and recording with decision procedures,
 Applying `both_sides_times2`
 Instantiating the top quantifier in -1 with the terms: c' , $\text{ndiv}(a', c') \times b'$, $\text{ndiv}(a' \times b', c')$,
 Simplifying, rewriting, and recording with decision procedures,
 This completes the proof of **ndiv_times_divident_1.1.1**.

ndiv_times_divident_1.1.2:

{-1}	$a' = 0 + c' \times \text{ndiv}(a', c')$
{-2}	$a' \times b' = \text{rem}(c')(a' \times b') + c' \times \text{ndiv}(a' \times b', c')$
{-3}	$\text{integer_pred}(a')$
{-4}	$\text{integer_pred}(b')$
{-5}	$c' > 0$
{-6}	$\text{divides}(c', a')$
{1}	
$\text{rem}(c')(a' \times b') = 0$	
{2}	
$\text{ndiv}(a' \times b', c') = \text{ndiv}(a', c') \times b'$	

Rewriting using `rem_def2`, matching in *,
 Rewriting using `divides_prod1`, matching in *,
 This completes the proof of **ndiv_times_divident_1.1.2**.

`ndiv_times_divident_1.2:`

{-1}	$a' = \text{rem}(c')(a') + c' \times \text{ndiv}(a', c')$
{-2}	$a' \times b' = \text{rem}(c')(a' \times b') + c' \times \text{ndiv}(a' \times b', c')$
{-3}	$\text{integer_pred}(a')$
{-4}	$\text{integer_pred}(b')$
{-5}	$c' > 0$
{-6}	$\text{divides}(c', a')$
{1}	$\text{rem}(c')(a') = 0$
{2}	$\text{ndiv}(a' \times b', c') = \text{ndiv}(a', c') \times b'$

Rewriting using `rem_def2`, matching in `*`,
 This completes the proof of `ndiv_times_divident_1.2`.
 Q.E.D.

C.143.15 Number_Props.ndiv_times_divident_2

Terse proof for `ndiv_times_divident_2`.

`ndiv_times_divident_2:`

{1}	$\forall (a, b: \text{int}, c: \text{posnat}):$ $\text{divides}(c, b) \supset \text{ndiv}(a \times b, c) = a \times \text{ndiv}(b, c)$
-----	---

Repeatedly Skolemizing and flattening,
 Applying `ndiv_times_divident_1`
 Instantiating the top quantifier in -1 with the terms: b', a', c' ,
 Simplifying, rewriting, and recording with decision procedures,
 This completes the proof of `ndiv_times_divident_2`.
 Q.E.D.

C.143.16 Number_Props.rem_prod_3

Terse proof for `rem_prod_3`.

`rem_prod_3:`

{1}	$\forall (b_1, b_2: \text{posnat}, x: \text{int}):$ $\text{rem}(b_1 \times b_2)(x) = \text{rem}(b_1)(x) + \text{rem}(b_2)(\text{ndiv}(x, b_1)) \times b_1$
-----	---

Installing automatic rewrites from: `ndiv_self divides_reflexive rem_def2`
 Repeatedly Skolemizing and flattening,
 Rewriting using `rem_def2`, matching in `*`,
 we get 2 subgoals:

`rem_prod_3.1:`

{-1}	$b'_1 > 0$
{-2}	$b'_2 > 0$
{-3}	$\text{integer_pred}(x')$
{1}	$\text{divides}(b'_1 \times b'_2,$ $\quad -1 \times \text{rem}(b'_1)(x') - \text{rem}(b'_2)(\text{ndiv}(x', b'_1)) \times b'_1 + x')$

Rewriting using `divides_times`, matching in `*`,
 Applying propositional simplification,
 we get 2 subgoals:

rem_prod_3.1.1:

<div style="display: flex; flex-direction: column; gap: 5px;"> <div style="border-bottom: 1px solid black; padding-bottom: 5px;"> <div style="display: flex; gap: 10px;"> {-1} $b'_1 > 0$ </div> <div style="display: flex; gap: 10px;"> {-2} $b'_2 > 0$ </div> <div style="display: flex; gap: 10px;"> {-3} $\text{integer_pred}(x')$ </div> </div> </div>	<div style="display: flex; gap: 10px;"> {1} $\text{divides}(b'_1,$ $\quad -1 \times \text{rem}(b'_1)(x') - \text{rem}(b'_2)(\text{ndiv}(x', b'_1)) \times b'_1 + x')$ </div>
--	--

Applying divides_diff

Instantiating the top quantifier in -1 with the terms: $\text{rem}(b'_2)(\text{ndiv}(x', b'_1)) \times b'_1$, $x' - \text{rem}(b'_1)(x')$, b'_1 ,
Simplifying, rewriting, and recording with decision procedures,

Hiding formulas: 2,

Applying propositional simplification,

we get 2 subgoals:

rem_prod_3.1.1.1:

<div style="display: flex; flex-direction: column; gap: 5px;"> <div style="border-bottom: 1px solid black; padding-bottom: 5px;"> <div style="display: flex; gap: 10px;"> {-1} $b'_1 > 0$ </div> <div style="display: flex; gap: 10px;"> {-2} $b'_2 > 0$ </div> <div style="display: flex; gap: 10px;"> {-3} $\text{integer_pred}(x')$ </div> </div> </div>	<div style="display: flex; gap: 10px;"> {1} $\text{divides}(b'_1, x' - \text{rem}(b'_1)(x'))$ </div>
--	--

Using lemma rem_def2,

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of rem_prod_3.1.1.1.

rem_prod_3.1.1.2:

<div style="display: flex; flex-direction: column; gap: 5px;"> <div style="border-bottom: 1px solid black; padding-bottom: 5px;"> <div style="display: flex; gap: 10px;"> {-1} $b'_1 > 0$ </div> <div style="display: flex; gap: 10px;"> {-2} $b'_2 > 0$ </div> <div style="display: flex; gap: 10px;"> {-3} $\text{integer_pred}(x')$ </div> </div> </div>	<div style="display: flex; gap: 10px;"> {1} $\text{divides}(b'_1, \text{rem}(b'_2)(\text{ndiv}(x', b'_1)) \times b'_1)$ </div>
--	--

Using lemma divides_prod2,

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of rem_prod_3.1.1.2.

rem_prod_3.1.2:

<div style="display: flex; flex-direction: column; gap: 5px;"> <div style="border-bottom: 1px solid black; padding-bottom: 5px;"> <div style="display: flex; gap: 10px;"> {-1} $b'_1 > 0$ </div> <div style="display: flex; gap: 10px;"> {-2} $b'_2 > 0$ </div> <div style="display: flex; gap: 10px;"> {-3} $\text{integer_pred}(x')$ </div> </div> </div>	<div style="display: flex; gap: 10px;"> {1} $\text{divides}(b'_2,$ $\quad \text{ndiv}(-1 \times \text{rem}(b'_1)(x') - \text{rem}(b'_2)(\text{ndiv}(x', b'_1)) \times b'_1 + x',$ $\quad \quad b'_1)$ </div>
--	--

Applying ndiv_minus_2

Instantiating the top quantifier in -1 with the terms: $x' - \text{rem}(b'_1)(x')$, $\text{rem}(b'_2)(\text{ndiv}(x', b'_1)) \times b'_1$, b'_1 ,
Splitting conjunctions,

we get 2 subgoals:

rem_prod_3.1.2.1:

<div style="display: flex; flex-direction: column; gap: 5px;"> <div style="border-bottom: 1px solid black; padding-bottom: 5px;"> <div style="display: flex; gap: 10px;"> {-1} $\text{ndiv}(x' - \text{rem}(b'_1)(x') - \text{rem}(b'_2)(\text{ndiv}(x', b'_1)) \times b'_1, b'_1) =$ $\quad \text{ndiv}(x' - \text{rem}(b'_1)(x'), b'_1) - \text{ndiv}(\text{rem}(b'_2)(\text{ndiv}(x', b'_1)) \times b'_1, b'_1)$ </div> <div style="display: flex; gap: 10px;"> {-2} $b'_1 > 0$ </div> <div style="display: flex; gap: 10px;"> {-3} $b'_2 > 0$ </div> <div style="display: flex; gap: 10px;"> {-4} $\text{integer_pred}(x')$ </div> </div> </div>	<div style="display: flex; gap: 10px;"> {1} $\text{divides}(b'_2,$ $\quad \text{ndiv}(-1 \times \text{rem}(b'_1)(x') - \text{rem}(b'_2)(\text{ndiv}(x', b'_1)) \times b'_1 + x',$ $\quad \quad b'_1)$ </div>
---	--

Simplifying, rewriting, and recording with decision procedures,
 Replacing using formula -1,
 Hiding formulas: -1,
 Applying `ndiv_times_divident_2`
 Instantiating the top quantifier in -1 with the terms: $\text{rem}(b'_2)(\text{ndiv}(x', b'_1))$, b'_1 , b'_1 ,
 Simplifying, rewriting, and recording with decision procedures,
 Replacing using formula -1,
 Hiding formulas: -1,
 Applying `ndiv_plus_mod`
 Instantiating the top quantifier in -1 with the terms: $x' - \text{rem}(b'_1)(x')$, b'_1 , $\text{rem}(b'_1)(x')$,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 we get 2 subgoals:

`rem_prod_3.1.2.1.1:`

$$\left| \begin{array}{l} \{-1\} \quad \text{ndiv}(x', b'_1) = \text{ndiv}(x' - \text{rem}(b'_1)(x'), b'_1) \\ \{-2\} \quad b'_1 > 0 \\ \{-3\} \quad b'_2 > 0 \\ \{-4\} \quad \text{integer_pred}(x') \end{array} \right. \hline \{1\} \quad \text{divides}(b'_2, \text{ndiv}(x' - \text{rem}(b'_1)(x'), b'_1) - \text{rem}(b'_2)(\text{ndiv}(x', b'_1)))$$

Replacing using formula -1,
 Hiding formulas: -1,
 Adding type constraints for $\text{rem}(b'_2)(\text{ndiv}(x', b'_1))$,
 Repeatedly Skolemizing and flattening,
 Expanding the definition of divides,
 Instantiating quantified variables,
 Simplifying, rewriting, and recording with decision procedures,
 This completes the proof of `rem_prod_3.1.2.1.1`.

`rem_prod_3.1.2.1.2:`

$$\left| \begin{array}{l} \{-1\} \quad b'_1 > 0 \\ \{-2\} \quad b'_2 > 0 \\ \{-3\} \quad \text{integer_pred}(x') \end{array} \right. \hline \{1\} \quad \text{divides}(b'_1, x' - \text{rem}(b'_1)(x')) \\ \{2\} \quad \text{divides}(b'_2, \text{ndiv}(x' - \text{rem}(b'_1)(x'), b'_1) - \text{rem}(b'_2)(\text{ndiv}(x', b'_1)))$$

Hiding formulas: 2,
 Adding type constraints for $\text{rem}(b'_2)(\text{ndiv}(x', b'_1))$,
 Repeatedly Skolemizing and flattening,
 Expanding the definition of divides,
 Instantiating quantified variables,
 Simplifying, rewriting, and recording with decision procedures,
 This completes the proof of `rem_prod_3.1.2.1.2`.

`rem_prod_3.1.2.2:`

$$\left| \begin{array}{l} \{-1\} \quad b'_1 > 0 \\ \{-2\} \quad b'_2 > 0 \\ \{-3\} \quad \text{integer_pred}(x') \end{array} \right. \hline \{1\} \quad \text{divides}(b'_1, \text{rem}(b'_2)(\text{ndiv}(x', b'_1)) \times b'_1) \\ \{2\} \quad \text{divides}(b'_2, \text{ndiv}(-1 \times \text{rem}(b'_1)(x') - \text{rem}(b'_2)(\text{ndiv}(x', b'_1)) \times b'_1 + x', b'_1))$$

Hiding formulas: 2,
 Using lemma `divides_prod2`,

C Proof scripts

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `rem_prod_3.1.2.2`.

`rem_prod_3.2`:

{-1}	$b'_1 > 0$
{-2}	$b'_2 > 0$
{-3}	<code>integer_pred(x')</code>
{1}	$\text{rem}(b'_1)(x') + \text{rem}(b'_2)(\text{ndiv}(x', b'_1)) \times b'_1 < b'_1 \times b'_2$
{2}	$\text{rem}(b'_1 \times b'_2)(x') =$ $\text{rem}(b'_1)(x') + \text{rem}(b'_2)(\text{ndiv}(x', b'_1)) \times b'_1$

Hiding formulas: 2,

Applying `number_split_less`

Instantiating the top quantifier in -1 with the terms: $\text{rem}(b'_1)(x')$, $\text{rem}(b'_2)(\text{ndiv}(x', b'_1))$, b'_1 , b'_2 ,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `rem_prod_3.2`.

Q.E.D.

C.143.17 Number_Props.ndiv_times_2

Terse proof for `ndiv_times_2`.

`ndiv_times_2`:

{1}	$\forall (a: \text{int}, b, c: \text{posnat}): \text{ndiv}(a, b \times c) = \text{ndiv}(\text{ndiv}(a, b), c)$
-----	--

Repeatedly Skolemizing and flattening,

Adding type constraints for `ndiv(a!1, b!1)`,

Adding type constraints for `ndiv(a!1, b!1 * c!1)`,

Adding type constraints for `ndiv(ndiv(a!1, b!1), c!1)`,

Replacing using formula -1,

Simplifying, rewriting, and recording with decision procedures,

Rewriting using `rem_prod_3`, matching in `*`,

Hiding formulas: -1,

Applying `both_sides_times1`

Instantiating the top quantifier in -1 with the terms: $b' \times c'$, $\text{ndiv}(a', b' \times c')$, $\text{ndiv}(\text{ndiv}(a', b'), c')$,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `ndiv_times_2`.

Q.E.D.

C.143.18 Number_Props.both_sides_ndiv_lt1

Terse proof for `both_sides_ndiv_lt1`.

`both_sides_ndiv_lt1`:

{1}	$\forall (a, b: \text{int}, c: \text{posnat}): a < b \wedge \text{divides}(c, b) \supset \text{ndiv}(a, c) < \text{ndiv}(b, c)$
-----	---

Repeatedly Skolemizing and flattening,

Adding type constraints for `ndiv(a!1, c!1)`,

Adding type constraints for `ndiv(b!1, c!1)`,

Case splitting on $\text{rem}(c!1)(b!1) = 0$,

we get 2 subgoals:

`both_sides_ndiv_lt1.1:`

{-1}	$\text{rem}(c')(b') = 0$
{-2}	$b' = \text{rem}(c')(b') + c' \times \text{ndiv}(b', c')$
{-3}	$a' = \text{rem}(c')(a') + c' \times \text{ndiv}(a', c')$
{-4}	$\text{integer_pred}(a')$
{-5}	$\text{integer_pred}(b')$
{-6}	$c' > 0$
{-7}	$a' < b'$
{-8}	$\text{divides}(c', b')$
{1}	$\text{ndiv}(a', c') < \text{ndiv}(b', c')$

Replacing using formula -1,

Hiding formulas: -1,

Simplifying, rewriting, and recording with decision procedures,

Applying `both_sides_times_pos_ge2`

Instantiating the top quantifier in -1 with the terms: c' , $\text{ndiv}(a', c')$, $\text{ndiv}(b', c')$,

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `both_sides_ndiv_lt1.1`.

`both_sides_ndiv_lt1.2:`

{-1}	$b' = \text{rem}(c')(b') + c' \times \text{ndiv}(b', c')$
{-2}	$a' = \text{rem}(c')(a') + c' \times \text{ndiv}(a', c')$
{-3}	$\text{integer_pred}(a')$
{-4}	$\text{integer_pred}(b')$
{-5}	$c' > 0$
{-6}	$a' < b'$
{-7}	$\text{divides}(c', b')$
{1}	$\text{rem}(c')(b') = 0$
{2}	$\text{ndiv}(a', c') < \text{ndiv}(b', c')$

Rewriting using `rem_def2`, matching in *,

This completes the proof of `both_sides_ndiv_lt1.2`.

Q.E.D.

C.143.19 Number_Props.rem_ndiv

Terse proof for `rem_ndiv`.

`rem_ndiv:`

{1}	$\forall (a: \text{posnat}, b: \text{int}, c: \text{posnat}):$ $\text{rem}(a)(\text{ndiv}(b, c)) = \text{ndiv}(\text{rem}(c \times a)(b), c)$
-----	--

Repeatedly Skolemizing and flattening,

Rewriting using `rem_prod_3`, matching in *,

Applying `ndiv_plus_mod`

Instantiating the top quantifier in -1 with the terms: $\text{rem}(a')(\text{ndiv}(b', c')) \times c'$, c' , $\text{rem}(c')(b')$,

Rewriting using `divides_prod2`, matching in *,

we get 2 subgoals:

rem_ndiv.1:

{-1}	$\text{ndiv}(\text{rem}(a')(\text{ndiv}(b', c')) \times c' + \text{rem}(c')(b'), c') =$ $\text{ndiv}(\text{rem}(a')(\text{ndiv}(b', c')) \times c', c')$
{-2}	$a' > 0$
{-3}	$\text{integer_pred}(b')$
{-4}	$c' > 0$
{1}	$\text{rem}(a')(\text{ndiv}(b', c')) =$ $\text{ndiv}(\text{rem}(c')(b') + \text{rem}(a')(\text{ndiv}(b', c')) \times c', c')$

Replacing using formula -1,

Hiding formulas: -1,

Rewriting using ndiv_times_divident_2, matching in *,

we get 2 subgoals:

rem_ndiv.1.1:

{-1}	$a' > 0$
{-2}	$\text{integer_pred}(b')$
{-3}	$c' > 0$
{1}	$\text{rem}(a')(\text{ndiv}(b', c')) = \text{rem}(a')(\text{ndiv}(b', c')) \times \text{ndiv}(c', c')$

Rewriting using ndiv_self, matching in *,

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of rem_ndiv.1.1.

rem_ndiv.1.2:

{-1}	$a' > 0$
{-2}	$\text{integer_pred}(b')$
{-3}	$c' > 0$
{1}	$\text{divides}(c', c')$
{2}	$\text{rem}(a')(\text{ndiv}(b', c')) = \text{ndiv}(\text{rem}(a')(\text{ndiv}(b', c')) \times c', c')$

Rewriting using divides_reflexive, matching in *,

This completes the proof of rem_ndiv.1.2.

rem_ndiv.2:

{-1}	$\text{divides}(c', \text{rem}(a')(\text{ndiv}(b', c')) \times c') \supset$ $\text{ndiv}(\text{rem}(c')(b') + \text{rem}(a')(\text{ndiv}(b', c')) \times c', c') =$ $\text{ndiv}(\text{rem}(a')(\text{ndiv}(b', c')) \times c', c')$
{-2}	$a' > 0$
{-3}	$\text{integer_pred}(b')$
{-4}	$c' > 0$
{1}	$\text{divides}(c', c')$
{2}	$\text{rem}(a')(\text{ndiv}(b', c')) =$ $\text{ndiv}(\text{rem}(c')(b') + \text{rem}(a')(\text{ndiv}(b', c')) \times c', c')$

Rewriting using divides_reflexive, matching in *,

This completes the proof of rem_ndiv.2.

Q.E.D.

C.143.20 Number_Props.mod_rem

Terse proof for mod_rem.

mod_rem:

{1}	$\forall (i: \text{int}, b: \text{posnat}): \text{mod}(i, b) = \text{rem}(b)(i)$
-----	--

Repeatedly Skolemizing and flattening,
 Expanding the definition of mod,
 Using lemma rem_floor,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 This completes the proof of mod_rem.
 Q.E.D.

C.143.21 Number_Props.ndiv_rem_divisible_TCC1

Terse proof for ndiv_rem_divisible_TCC1.

ndiv_rem_divisible_TCC1:

$$\frac{}{\{1\} \quad \forall (a, c: \text{posnat}): \text{divides}(c, a) \wedge c \leq a \supset \text{ndiv}(a, c) > 0}$$

Repeatedly Skolemizing and flattening,
 Adding type constraints for ndiv(a!1, c!1),
 Case splitting on rem(c!1)(a!1) = 0,
 we get 2 subgoals:

ndiv_rem_divisible_TCC1.1:

$$\frac{\begin{array}{l} \{-1\} \quad \text{rem}(c')(a') = 0 \\ \{-2\} \quad a' = \text{rem}(c')(a') + c' \times \text{ndiv}(a', c') \\ \{-3\} \quad \text{ndiv}(a', c') \leq a' \\ \{-4\} \quad a' > 0 \\ \{-5\} \quad c' > 0 \\ \{-6\} \quad \text{divides}(c', a') \\ \{-7\} \quad c' \leq a' \end{array}}{\{1\} \quad \text{ndiv}(a', c') > 0}$$

Replacing using formula -1,
 Simplifying, rewriting, and recording with decision procedures,
 This completes the proof of ndiv_rem_divisible_TCC1.1.

ndiv_rem_divisible_TCC1.2:

$$\frac{\begin{array}{l} \{-1\} \quad a' = \text{rem}(c')(a') + c' \times \text{ndiv}(a', c') \\ \{-2\} \quad \text{ndiv}(a', c') \leq a' \\ \{-3\} \quad a' > 0 \\ \{-4\} \quad c' > 0 \\ \{-5\} \quad \text{divides}(c', a') \\ \{-6\} \quad c' \leq a' \end{array}}{\begin{array}{l} \{1\} \quad \text{rem}(c')(a') = 0 \\ \{2\} \quad \text{ndiv}(a', c') > 0 \end{array}}$$

Rewriting using rem_def2, matching in *,
 This completes the proof of ndiv_rem_divisible_TCC1.2.
 Q.E.D.

C.143.22 Number_Props.ndiv_rem_divisible

Terse proof for ndiv_rem_divisible.

C Proof scripts

`ndiv_rem_divisible:`

$\{1\}$	$\forall (a, c: \text{posnat}, b: \text{int}):$ $\text{divides}(c, a) \wedge c \leq a \supset$ $\text{ndiv}(\text{rem}(a)(b), c) = \text{rem}(\text{ndiv}(a, c))(\text{ndiv}(b, c))$
---------	--

Repeatedly Skolemizing and flattening,

Expanding the definition of divides,

Repeatedly Skolemizing and flattening,

Replacing using formula -5,

Rewriting using `rem_ndiv`, matching in *,

we get 2 subgoals:

`ndiv_rem_divisible.1:`

$\{-1\}$ <code>integer_pred(x')</code> $\{-2\}$ $c' \times x' > 0$ $\{-3\}$ $c' > 0$ $\{-4\}$ <code>integer_pred(b')</code> $\{-5\}$ $a' = c' \times x'$ $\{-6\}$ $c' \leq c' \times x'$	$\{1\}$ $\text{rem}(x')(\text{ndiv}(b', c')) = \text{rem}(\text{ndiv}(c' \times x', c'))(\text{ndiv}(b', c'))$
---	--

Rewriting using `ndiv_times_divident_1`, matching in *,

we get 2 subgoals:

`ndiv_rem_divisible.1.1:`

$\{-1\}$ <code>integer_pred(x')</code> $\{-2\}$ $c' \times x' > 0$ $\{-3\}$ $c' > 0$ $\{-4\}$ <code>integer_pred(b')</code> $\{-5\}$ $a' = c' \times x'$ $\{-6\}$ $c' \leq c' \times x'$	$\{1\}$ $\text{rem}(x')(\text{ndiv}(b', c')) = \text{rem}(\text{ndiv}(c', c') \times x')(\text{ndiv}(b', c'))$
---	--

Rewriting using `ndiv_self`, matching in *,

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `ndiv_rem_divisible.1.1`.

`ndiv_rem_divisible.1.2:`

$\{-1\}$ <code>integer_pred(x')</code> $\{-2\}$ $c' \times x' > 0$ $\{-3\}$ $c' > 0$ $\{-4\}$ <code>integer_pred(b')</code> $\{-5\}$ $a' = c' \times x'$ $\{-6\}$ $c' \leq c' \times x'$	$\{1\}$ <code>divides(c', c')</code> $\{2\}$ $\text{rem}(x')(\text{ndiv}(b', c')) = \text{rem}(\text{ndiv}(c' \times x', c'))(\text{ndiv}(b', c'))$
---	--

Rewriting using `divides_reflexive`, matching in *,

This completes the proof of `ndiv_rem_divisible.1.2`.

ndiv_rem_divisible.2:

{-1}	integer_pred(x')
{-2}	$c' \times x' > 0$
{-3}	$c' > 0$
{-4}	integer_pred(b')
{-5}	$a' = c' \times x'$
{-6}	$c' \leq c' \times x'$
{1}	$x' \geq 0 \wedge x' > 0$
{2}	$\text{ndiv}(\text{rem}(c' \times x')(b'), c') =$ $\text{rem}(\text{ndiv}(c' \times x', c'))(\text{ndiv}(b', c'))$

Applying both_sides_times_pos_gt2

Instantiating the top quantifier in -1 with the terms: c' , x' , 0,
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of ndiv_rem_divisible.2.
Q.E.D.

C.143.23 Number_Props.expt_2_8

Terse proof for expt_2_8.

expt_2_8:

{1}	$\text{expt}(2, 8) = 256$
-----	---------------------------

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of expt_2_8.
Q.E.D.

C.143.24 Number_Props.ndiv_0

Terse proof for ndiv_0.

ndiv_0:

{1}	$\forall (b: \text{posnat}): \text{ndiv}(0, b) = 0$
-----	---

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of ndiv_0.
Q.E.D.

C.143.25 Number_Props.ndiv_1

Terse proof for ndiv_1.

ndiv_1:

{1}	$\forall (i: \text{int}): \text{ndiv}(i, 1) = i$
-----	--

Repeatedly Skolemizing and flattening,
Adding type constraints for ndiv(i!1, 1),
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of ndiv_1.
Q.E.D.

C.143.26 Number_Props.ndiv_bigger

Terse proof for `ndiv_bigger`.

`ndiv_bigger`:

$$\{1\} \quad \forall (n: \text{nat}, m: \text{posnat}): n < m \supset \text{ndiv}(n, m) = 0$$

Repeatedly Skolemizing and flattening,
 Adding type constraints for `ndiv(n!1, m!1)`,
 Rewriting using `rem_mod2`, matching in `*`,
 Simplifying, rewriting, and recording with decision procedures,
 Applying `both_sides_times2`
 Instantiating the top quantifier in -1 with the terms: `m', 0, ndiv(n', m')`,
 Simplifying, rewriting, and recording with decision procedures,
 This completes the proof of `ndiv_bigger`.
 Q.E.D.

C.143.27 Number_Props.ndiv_reduce

Terse proof for `ndiv_reduce`.

`ndiv_reduce`:

$$\{1\} \quad \forall (a: \text{int}, b: \text{posnat}, c: \text{int}, d: \text{posnat}): \\ a \times d = b \times c \supset \text{ndiv}(a, b) = \text{ndiv}(c, d)$$

Repeatedly Skolemizing and flattening,
 Rewriting using `ndiv_floor`, matching in `*`,
 Rewriting using `ndiv_floor`, matching in `*`,
 Case splitting on `a!1 / b!1 = c!1 / d!1`,
 we get 2 subgoals:

`ndiv_reduce.1`:

$$\begin{array}{l} \{-1\} \quad a'/b' = c'/d' \\ \{-2\} \quad \text{integer_pred}(a') \\ \{-3\} \quad b' > 0 \\ \{-4\} \quad \text{integer_pred}(c') \\ \{-5\} \quad d' > 0 \\ \{-6\} \quad a' \times d' = b' \times c' \\ \hline \{1\} \quad \text{floor}(a'/b') = \text{floor}(c'/d') \end{array}$$

Simplifying, rewriting, and recording with decision procedures,
 This completes the proof of `ndiv_reduce.1`.

`ndiv_reduce.2`:

$$\begin{array}{l} \{-1\} \quad \text{integer_pred}(a') \\ \{-2\} \quad b' > 0 \\ \{-3\} \quad \text{integer_pred}(c') \\ \{-4\} \quad d' > 0 \\ \{-5\} \quad a' \times d' = b' \times c' \\ \hline \{1\} \quad a'/b' = c'/d' \\ \{2\} \quad \text{floor}(a'/b') = \text{floor}(c'/d') \end{array}$$

Case splitting on `a!1 / b!1 * b!1 = c!1 / d!1 * b!1`,
 we get 2 subgoals:

ndiv_reduce.2.1:

{-1}	$a'/b' \times b' = c'/d' \times b'$
{-2}	$\text{integer_pred}(a')$
{-3}	$b' > 0$
{-4}	$\text{integer_pred}(c')$
{-5}	$d' > 0$
{-6}	$a' \times d' = b' \times c'$
{1}	$a'/b' = c'/d'$
{2}	$\text{floor}(a'/b') = \text{floor}(c'/d')$

Simplifying, rewriting, and recording with decision procedures,
This completes the proof of ndiv_reduce.2.1.

ndiv_reduce.2.2:

{-1}	$\text{integer_pred}(a')$
{-2}	$b' > 0$
{-3}	$\text{integer_pred}(c')$
{-4}	$d' > 0$
{-5}	$a' \times d' = b' \times c'$
{1}	$a'/b' \times b' = c'/d' \times b'$
{2}	$a'/b' = c'/d'$
{3}	$\text{floor}(a'/b') = \text{floor}(c'/d')$

Applying both_sides_times1
Instantiating the top quantifier in -1 with the terms: $d', a'/b' \times b', c'/d' \times b'$,
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of ndiv_reduce.2.2.
Q.E.D.

C.143.28 Number_Props.ndiv_expt_expt_TCC1

Terse proof for ndiv_expt_expt_TCC1.

ndiv_expt_expt_TCC1:

{1}	$\forall (a: \text{posnat}, n, m: \text{nat}): \neg n < m \wedge a > 1 \supset n - m \geq 0$
-----	--

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of ndiv_expt_expt_TCC1.
Q.E.D.

C.143.29 Number_Props.ndiv_expt_expt

Terse proof for ndiv_expt_expt.

ndiv_expt_expt:

{1}	$\forall (a: \text{posnat}, n, m: \text{nat}):$ $a > 1 \supset$ $\text{ndiv}(\text{expt}(a, n), \text{expt}(a, m)) =$ $\text{IF } n < m \text{ THEN } 0 \text{ ELSE } \text{expt}(a, n - m) \text{ ENDIF}$
-----	---

Repeatedly Skolemizing and flattening,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
we get 2 subgoals:

C Proof scripts

`ndiv_expt_expt.1:`

{-1}	$a' > 0$
{-2}	$n' \geq 0$
{-3}	$m' \geq 0$
{-4}	$a' > 1$
{-5}	$n' < m'$
{1}	
	$\text{ndiv}(\text{expt}(a', n'), \text{expt}(a', m')) = 0$

Rewriting using `ndiv_bigger`, matching in `*`,

Applying `both_sides_expt_gt1_lt_aux`

Instantiating the top quantifier in -1 with the terms: $a', n' - 1, m' - 1$,
we get 2 subgoals:

`ndiv_expt_expt.1.1:`

{-1}	$\text{expt}(a', n' - 1 + 1) < \text{expt}(a', m' - 1 + 1) \equiv$ $n' - 1 < m' - 1$
{-2}	$a' > 0$
{-3}	$n' \geq 0$
{-4}	$m' \geq 0$
{-5}	$a' > 1$
{-6}	$n' < m'$
{1}	
	$\text{expt}(a', n') < \text{expt}(a', m')$
{2}	
	$\text{ndiv}(\text{expt}(a', n'), \text{expt}(a', m')) = 0$

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `ndiv_expt_expt.1.1`.

`ndiv_expt_expt.1.2:`

{-1}	$a' > 0$
{-2}	$n' \geq 0$
{-3}	$m' \geq 0$
{-4}	$a' > 1$
{-5}	$n' < m'$
{1}	
	$n' - 1 \geq 0$
{2}	
	$\text{expt}(a', n') < \text{expt}(a', m')$
{3}	
	$\text{ndiv}(\text{expt}(a', n'), \text{expt}(a', m')) = 0$

Expanding the definition of `expt`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Applying `increasing_expt_aux`

Instantiating the top quantifier in -1 with the terms: $a', m' - 2$,
we get 2 subgoals:

`ndiv_expt_expt.1.2.1:`

{-1}	$\text{expt}(a', m' - 2 + 2) > a'$
{-2}	$a' > 0$
{-3}	$n' \geq 0$
{-4}	$m' \geq 0$
{-5}	$a' > 1$
{-6}	$n' < m'$
{1}	
	$n' - 1 \geq 0$
{2}	
	$1 < \text{expt}(a', m')$
{3}	
	$\text{ndiv}(\text{expt}(a', n'), \text{expt}(a', m')) = 0$

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `ndiv_expt_expt.1.2.1`.

`ndiv_expt_expt.1.2.2`:

{-1}	$a' > 0$
{-2}	$n' \geq 0$
{-3}	$m' \geq 0$
{-4}	$a' > 1$
{-5}	$n' < m'$
{1}	$m' - 2 \geq 0$
{2}	$n' - 1 \geq 0$
{3}	$1 < \text{expt}(a', m')$
{4}	$\text{ndiv}(\text{expt}(a', n'), \text{expt}(a', m')) = 0$

Expanding the definition of `expt`,

Expanding the definition of `expt`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `ndiv_expt_expt.1.2.2`.

`ndiv_expt_expt.2`:

{-1}	$a' > 0$
{-2}	$n' \geq 0$
{-3}	$m' \geq 0$
{-4}	$a' > 1$
{1}	$n' < m'$
{2}	$\text{ndiv}(\text{expt}(a', n'), \text{expt}(a', m')) = \text{expt}(a', n' - m')$

Applying `ndiv_reduce`

Instantiating the top quantifier in -1 with the terms: `expt(a', n')`, `expt(a', m')`, `expt(a', n' - m')`, 1,

Rewriting using `expt_plus_aux`, matching in `*`,

Rewriting using `ndiv_1`, matching in `*`,

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `ndiv_expt_expt.2`.

Q.E.D.

C.143.30 Number_Props.min_plus1_TCC1

Terse proof for `min_plus1_TCC1`.

`min_plus1_TCC1`:

{1}	$\forall (P: (\text{nonempty?}[\text{nat}])): \neg P(0) \supset \text{nonempty?}[\text{nat}](\lambda (n: \text{nat}): P(n + 1))$
-----	--

Repeatedly Skolemizing and flattening,

Expanding the definition of `nonempty?`,

Expanding the definition of `empty?`,

Repeatedly Skolemizing and flattening,

Expanding the definition of `member`,

Case splitting on `x!1 = 0`,

we get 2 subgoals:

min_plus1_TCC1.1:

{-1}	$x' = 0$
{-2}	$x' \geq 0$
{-3}	$P'(x')$
{-4}	$\forall (x: \text{nat}): \neg P'(1+x)$
{1}	
	$P'(0)$

Simplifying, rewriting, and recording with decision procedures,
This completes the proof of min_plus1_TCC1.1.

min_plus1_TCC1.2:

{-1}	$x' \geq 0$
{-2}	$P'(x')$
{-3}	$\forall (x: \text{nat}): \neg P'(1+x)$
{1}	
	$x' = 0$
{2}	$P'(0)$

Instantiating the top quantifier in -3 with the terms: $x' - 1$,
we get 2 subgoals:

min_plus1_TCC1.2.1:

{-1}	$x' \geq 0$
{-2}	$P'(x')$
{1}	
	$x' = 0$
{2}	$P'(0)$
{3}	$P'(1+x'-1)$

Simplifying, rewriting, and recording with decision procedures,
This completes the proof of min_plus1_TCC1.2.1.

min_plus1_TCC1.2.2:

{-1}	$x' \geq 0$
{-2}	$P'(x')$
{1}	
	$x' - 1 \geq 0$
{2}	$x' = 0$
{3}	$P'(0)$

Simplifying, rewriting, and recording with decision procedures,
This completes the proof of min_plus1_TCC1.2.2.

Q.E.D.

C.143.31 Number_Props.min_plus1

Terse proof for min_plus1.

min_plus1:

{1}	$\forall (P: (\text{nonempty?}[\text{nat}])): \neg P(0) \supset \min(P) = \min(\lambda (n: \text{nat}): P(n+1)) + 1$
-----	--

Repeatedly Skolemizing and flattening,
Using lemma min_def,
we get 3 subgoals:

min_plus1.1:

$$\frac{\begin{array}{l} \{-1\} \quad \min(\lambda (n: \text{nat}): P'(n+1)) = \min(P') - 1 \equiv \\ \quad \text{minimum?}(\min(P') - 1, \lambda (n: \text{nat}): P'(n+1)) \\ \{-2\} \quad \text{nonempty?}[\text{nat}](P') \end{array}}{\begin{array}{l} \{1\} \quad P'(0) \\ \{2\} \quad \min(P') = \min(\lambda (n: \text{nat}): P'(n+1)) + 1 \end{array}}$$

Simplifying, rewriting, and recording with decision procedures,
 Using lemma min_def,
 Simplifying, rewriting, and recording with decision procedures,
 Expanding the definition of minimum?,
 Hiding formulas: 3,
 Repeatedly Skolemizing and flattening,
 Instantiating quantified variables,
 Simplifying, rewriting, and recording with decision procedures,
 This completes the proof of min_plus1.1.

min_plus1.2:

$$\frac{\{-1\} \quad \text{nonempty?}[\text{nat}](P')}{\begin{array}{l} \{1\} \quad \text{nonempty?}[\text{nat}](\lambda (n: \text{nat}): P'(1+n)) \\ \{2\} \quad P'(0) \\ \{3\} \quad \min(P') = \min(\lambda (n: \text{nat}): P'(n+1)) + 1 \end{array}}$$

Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of min_plus1.2.

min_plus1.3:

$$\frac{\{-1\} \quad \text{nonempty?}[\text{nat}](P')}{\begin{array}{l} \{1\} \quad \min[\text{nat}](P') - 1 \geq 0 \\ \{2\} \quad P'(0) \\ \{3\} \quad \min(P') = \min(\lambda (n: \text{nat}): P'(n+1)) + 1 \end{array}}$$

Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of min_plus1.3.

Q.E.D.

C.143.32 Number_Props.min_member

Terse proof for min_member.

min_member:

$$\frac{}{\{1\} \quad \forall (P: (\text{nonempty?}[\text{nat}])), n: \text{nat}: \min(P) = n \supset P(n)}$$

Repeatedly Skolemizing and flattening,
 Rewriting using min_def, matching in *,
 Expanding the definition of minimum?,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 This completes the proof of min_member.
 Q.E.D.

C.144 Proofs for Ok_Result_Rewrite (state-transformer.pvs)

C.144.1 Ok_Result_Rewrite.ok_result_q_ok

Terse proof for ok_result_q_ok.

ok_result_q_ok:

$$\{1\} \quad \forall (\text{data: Data1}, q: [\text{State} \rightarrow \text{ExprResult}[\text{State}, \text{Data2}]], s: \text{State}):$$

$$\text{OK?}[\text{State}, \text{Data1}]((q \ \#\# \ \text{ok_result}(\text{data}))(s)) = \text{OK?}(q(s))$$

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of ok_result_q_ok.

Q.E.D.

C.144.2 Ok_Result_Rewrite.ok_result_q_state_TCC1

Terse proof for ok_result_q_state_TCC1.

ok_result_q_state_TCC1:

$$\{1\} \quad \forall (\text{data: Data1}, q: [\text{State} \rightarrow \text{ExprResult}[\text{State}, \text{Data2}]], s: \text{State}):$$

$$\text{OK?}(q(s)) \supset$$
$$\text{OK?}[\text{State}, \text{Data1}]$$
$$((\#\#[\text{State}, \text{Data2}, \text{Data1}](q, \text{ok_result}[\text{State}, \text{Data1}](\text{data}))(s))$$
$$\vee$$
$$\text{Exception?}[\text{State}, \text{Data1}]$$
$$((\#\#[\text{State}, \text{Data2}, \text{Data1}](q, \text{ok_result}[\text{State}, \text{Data1}](\text{data}))(s))$$

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of ok_result_q_state_TCC1.

Q.E.D.

C.144.3 Ok_Result_Rewrite.ok_result_q_state_TCC2

Terse proof for ok_result_q_state_TCC2.

ok_result_q_state_TCC2:

$$\{1\} \quad \forall (q: [\text{State} \rightarrow \text{ExprResult}[\text{State}, \text{Data2}]], s: \text{State}):$$

$$\text{OK?}(q(s)) \supset \text{OK?}[\text{State}, \text{Data2}](q(s)) \vee \text{Exception?}[\text{State}, \text{Data2}](q(s))$$

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of ok_result_q_state_TCC2.

Q.E.D.

C.144.4 Ok_Result_Rewrite.ok_result_q_state

Terse proof for ok_result_q_state.

ok_result_q_state:

$$\frac{\{1\} \quad \forall (\text{data}: \text{Data1}, q: [\text{State} \rightarrow \text{ExprResult}[\text{State}, \text{Data2}]]), s: \text{State}):}{\text{OK?}(q(s)) \supset \text{state}[\text{State}, \text{Data1}]((q \text{ ## } \text{ok_result}(\text{data}))(s)) = \text{state}(q(s))}$$

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `ok_result_q_state`.

Q.E.D.

C.144.5 `Ok_Result_Rewrite.ok_result_q_data_TCC1`

Terse proof for `ok_result_q_data_TCC1`.

ok_result_q_data_TCC1:

$$\frac{\{1\} \quad \forall (\text{data}: \text{Data1}, q: [\text{State} \rightarrow \text{ExprResult}[\text{State}, \text{Data2}]]), s: \text{State}):}{\text{OK?}(q(s)) \supset \text{OK?}[\text{State}, \text{Data1}]((\text{##}[\text{State}, \text{Data2}, \text{Data1}](q, \text{ok_result}[\text{State}, \text{Data1}](\text{data}))(s)))}$$

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `ok_result_q_data_TCC1`.

Q.E.D.

C.144.6 `Ok_Result_Rewrite.ok_result_q_data`

Terse proof for `ok_result_q_data`.

ok_result_q_data:

$$\frac{\{1\} \quad \forall (\text{data1}: \text{Data1}, q: [\text{State} \rightarrow \text{ExprResult}[\text{State}, \text{Data2}]]), s: \text{State}):}{\text{OK?}(q(s)) \supset \text{data}((q \text{ ## } \text{ok_result}(\text{data1}))(s)) = \text{data1}}$$

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `ok_result_q_data`.

Q.E.D.

C.144.7 `Ok_Result_Rewrite.ok_result_elimination_1`

Terse proof for `ok_result_elimination_1`.

ok_result_elimination_1:

$$\frac{\{1\} \quad \forall (\text{data}: \text{Data1}, q: [\text{State} \rightarrow \text{ExprResult}[\text{State}, \text{Data2}]]):}{(\text{ok_result}[\text{State}, \text{Data1}](\text{data}) \text{ ## } q) = q}$$

Repeatedly Skolemizing and flattening,

Applying decompose-equality,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `ok_result_elimination_1`.

Q.E.D.

C.145 Proofs for PDBR_Data_Model (paging-data-models.pvs)

C.145.1 PDBR_Data_Model.pdbr_valid?_TCC1

Terse proof for pdbr_valid?_TCC1.

pdbr_valid?_TCC1:

$$\{1\} \quad \forall (\text{data}: \text{list}[\text{Byte}]): \text{length}(\text{data}) = 4 \supset 1 < \text{length}[\text{Byte}](\text{data})$$

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of pdbr_valid?_TCC1.

Q.E.D.

C.145.2 PDBR_Data_Model.pdbr_valid?_TCC2

Terse proof for pdbr_valid?_TCC2.

pdbr_valid?_TCC2:

$$\{1\} \quad \forall (\text{data}: \text{list}[\text{Byte}]):$$
$$\text{cut_bits}(\text{nth}(\text{data}, 1), 0, 4) = 0 \wedge \text{length}(\text{data}) = 4 \supset 0 < \text{length}[\text{Byte}](\text{data})$$

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of pdbr_valid?_TCC2.

Q.E.D.

C.145.3 PDBR_Data_Model.pdbr_to_byte_TCC1

Terse proof for pdbr_to_byte_TCC1.

pdbr_to_byte_TCC1:

$$\{1\} \quad \forall (\text{pdbr}: \text{Pdbr_type}):$$
$$\text{overwrite_bool_bit}(\text{overwrite_bool_bit}(0, \text{pdbr}'\text{pwt}, 3), \text{pdbr}'\text{pcd}, 4) < \text{max_byte}$$

Repeatedly Skolemizing and flattening,

Installing automatic rewrites from: every less_than_max_byte overwrite_bool_bit_less

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of pdbr_to_byte_TCC1.

Q.E.D.

C.145.4 PDBR_Data_Model.pdbr_to_byte_TCC2

Terse proof for pdbr_to_byte_TCC2.

pdbr_to_byte_TCC2:

$$\{1\} \quad \forall (\text{pdbr}: \text{Pdbr_type}): \text{offset}(\text{pdbr}'\text{base_addr}) \geq 0$$

Repeatedly Skolemizing and flattening,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of pdbr_to_byte_TCC2.

Q.E.D.

C.145.5 PDBR_Data_Model.pdbr_to_byte_TCC3

Terse proof for `pdbr_to_byte_TCC3`.

`pdbr_to_byte_TCC3`:

$\{1\} \quad \forall (\text{pdbr}: \text{Pdbr_type}):$ $\quad \text{every}_{[\text{nat}]}$ $\quad \quad (\{s: \text{nat} \mid s < \text{max_byte}\})$ $\quad \quad \quad (:(\text{cut_bits}(\text{offset}(\text{pdbr}'\text{base_addr}), 8, 8),$ $\quad \quad \quad \quad \text{cut_bits}(\text{offset}(\text{pdbr}'\text{base_addr}), 16, 8),$ $\quad \quad \quad \quad \text{cut_bits}(\text{offset}(\text{pdbr}'\text{base_addr}), 24, 8):))$
--

Installing automatic rewrites from: `every cut_bits_below max_byte min_bits_per_byte bits_per_byte_minimum`

Repeatedly Skolemizing and flattening,

Using lemma `bits_per_byte_minimum`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 3 subgoals:

`pdbr_to_byte_TCC3.1`:

$\{-1\} \quad \text{bits_per_byte} \geq 8$	$\{1\} \quad \text{cut_bits}(\text{offset}(\text{pdbr}'\text{base_addr}), 24, 8) < \text{expt}(2, \text{bits_per_byte})$
--	--

Using lemma `cut_bits_below`,

Using lemma `both_sides_expt_gt1_le_aux`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `pdbr_to_byte_TCC3.1`.

`pdbr_to_byte_TCC3.2`:

$\{-1\} \quad \text{bits_per_byte} \geq 8$	$\{1\} \quad \text{cut_bits}(\text{offset}(\text{pdbr}'\text{base_addr}), 16, 8) < \text{expt}(2, \text{bits_per_byte})$
--	--

Using lemma `cut_bits_below`,

Using lemma `both_sides_expt_gt1_le_aux`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `pdbr_to_byte_TCC3.2`.

`pdbr_to_byte_TCC3.3`:

$\{-1\} \quad \text{bits_per_byte} \geq 8$	$\{1\} \quad \text{cut_bits}(\text{offset}(\text{pdbr}'\text{base_addr}), 8, 8) < \text{expt}(2, \text{bits_per_byte})$
--	---

Using lemma `cut_bits_below`,

Using lemma `both_sides_expt_gt1_le_aux`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `pdbr_to_byte_TCC3.3`.

Q.E.D.

C.145.6 PDBR_Data_Model.pdbr_from_byte_TCC1

Terse proof for `pdbr_from_byte_TCC1`.

`pdbr_from_byte_TCC1`:

$\{1\} \quad \forall (\text{data}: \text{list}[\text{Byte}], a: \text{Address}): \text{pdbr_valid?}(\text{data}, a) \supset 1 < \text{length}[\text{Byte}](\text{data})$

Trying repeated skolemization, instantiation, and if-lifting,

C Proof scripts

This completes the proof of `pdbr_from_byte_TCC1`.

Q.E.D.

C.145.7 PDBR_Data_Model.pdbr_from_byte_TCC2

Terse proof for `pdbr_from_byte_TCC2`.

`pdbr_from_byte_TCC2`:

$$\frac{}{\{1\} \quad \forall (\text{data}: \text{list}[\text{Byte}], a: \text{Address}): \text{pdbr_valid?}(\text{data}, a) \supset 2 < \text{length}[\text{Byte}](\text{data})}$$

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `pdbr_from_byte_TCC2`.

Q.E.D.

C.145.8 PDBR_Data_Model.pdbr_from_byte_TCC3

Terse proof for `pdbr_from_byte_TCC3`.

`pdbr_from_byte_TCC3`:

$$\frac{}{\{1\} \quad \forall (\text{data}: \text{list}[\text{Byte}], a: \text{Address}): \text{pdbr_valid?}(\text{data}, a) \supset 3 < \text{length}[\text{Byte}](\text{data})}$$

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `pdbr_from_byte_TCC3`.

Q.E.D.

C.145.9 PDBR_Data_Model.pdbr_from_byte_TCC4

Terse proof for `pdbr_from_byte_TCC4`.

pdbr_from_byte_TCC4:

```

{1}  ∀ (data: list[Byte], a: Address):
      pdbr_valid?(data, a) ⊃
      Mem?(Mem(overwrite_shift_bits(overwrite_shift_bits(overwrite_shift_bits(0,
                                                                              nth[Byte](data, 1),
                                                                              8,
                                                                              8),
                                                                              nth[Byte](data, 2), 16, 8),
                                                                              nth[Byte](data, 3), 24, 8))'type_of)
      ^
      (0 ≤
        Mem(overwrite_shift_bits(overwrite_shift_bits(overwrite_shift_bits(0,
                                                                              nth[Byte](data, 1),
                                                                              8,
                                                                              8),
                                                                              nth[Byte](data, 2), 16, 8),
                                                                              nth[Byte](data, 3), 24, 8))'offset)
      ^
        Mem(overwrite_shift_bits(overwrite_shift_bits(overwrite_shift_bits(0,
                                                                              nth[Byte](data, 1),
                                                                              8,
                                                                              8),
                                                                              nth[Byte](data, 2), 16, 8),
                                                                              nth[Byte](data, 3), 24, 8))'offset)
      < max_linear_offset)
      ^
      aligned?(12)
        (offset
          (Mem(overwrite_shift_bits(overwrite_shift_bits(overwrite_shift_bits
                                                                              (0,
                                                                              nth[Byte](data, 1),
                                                                              8,
                                                                              8),
                                                                              nth[Byte](data, 2),
                                                                              16,
                                                                              8),
                                                                              nth[Byte](data, 3), 24, 8))))

```

Repeatedly Skolemizing and flattening,

Installing automatic rewrites from: aligned_address? aligned_overwrite_shift_bits_more
aligned_overwrite_shift_bits_less aligned_zero max_linear_expt_val overwrite_shift_bits_less pde_valid?
max_linear_offset bus_width

Expanding the definition of Mem,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Rewriting using aligned_overwrite_shift_bits_more, matching in *,

Rewriting using aligned_overwrite_shift_bits_more, matching in *,

Rewriting using aligned_overwrite_shift_bits_less, matching in *,

Hiding formulas: 2, 3, 4,

C Proof scripts

Expanding the definition of `pdbr_valid?`,
Applying disjunctive simplification to flatten sequent,
Using lemma `cut_bits_aligned`,
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `pdbr_from_byte_TCC4`.
Q.E.D.

C.145.10 PDBR_Data_Model.pdbr_from_byte_TCC5

Terse proof for `pdbr_from_byte_TCC5`.

`pdbr_from_byte_TCC5`:

$$\frac{}{\{1\} \quad \forall (data: \text{list}[\text{Byte}], a: \text{Address}): \text{pdbr_valid?}(data, a) \supset 0 < \text{length}[\text{Byte}](data)}$$

Repeatedly Skolemizing and flattening,
Installing automatic rewrites from: `aligned_address?` `aligned_overwrite_shift_bits_more`
`aligned_overwrite_shift_bits_less` `aligned_zero` `max_linear_expt_val` `overwrite_shift_bits_less` `pde_valid?`
`max_linear_offset` `bus_width`

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Expanding the definition of `pdbr_valid?`,
which is trivially true.
This completes the proof of `pdbr_from_byte_TCC5`.
Q.E.D.

C.145.11 PDBR_Data_Model.pdbr_model_TCC1

Terse proof for `pdbr_model_TCC1`.

`pdbr_model_TCC1`:

$$\frac{}{\{1\} \quad \forall (f: [[\text{Pdbr_type}, \text{Address}] \rightarrow [\text{list}[\text{Byte}] \rightarrow \text{bool}]]): \\ (f = (\lambda (d: \text{Pdbr_type}, a: \text{Address}): (\text{zero_mask?}(4)))) \supset \\ (\forall (x_1: [\text{Pdbr_type}, \text{Address}]): \text{singleton?}[\text{list}[\text{Byte}]](f(x_1)))}$$

Repeatedly Skolemizing and flattening,
Replacing using formula -1,
Using lemma `singleton_zero_mask`,
Expanding the definition of `singleton?`,
which is trivially true.
This completes the proof of `pdbr_model_TCC1`.
Q.E.D.

C.145.12 PDBR_Data_Model.pdbr_valid_to_byte

Terse proof for `pdbr_valid_to_byte`.

`pdbr_valid_to_byte`:

$$\frac{}{\{1\} \quad \forall (d: \text{Pdbr_type}, a: \text{Address}): \\ \text{valid?}(\text{uidt}(\text{pdbr_model}))(\text{to_byte}(\text{pdbr_model})(d, a), a)}$$

Repeatedly Skolemizing and flattening,
Installing automatic rewrites from: `pdbr_model` `pdbr_to_byte` `pdbr_valid?` `nth` `length` `cut_bits_zero`
`cut_bits_overwrite_bool_bit_disjoined` `cut_bits_overwrite_bits_disjoined` `cut_bits_cut_bits`

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 Adding type constraints for d!1'base_addr,
 Using lemma aligned_cut_bits,
 Simplifying, rewriting, and recording with decision procedures,
 This completes the proof of pabr_valid_to_byte.
 Q.E.D.

C.145.13 PDBR_Data_Model.pabr_is_pod

Terse proof for pabr_is_pod.

pabr_is_pod:

$$\frac{}{\{1\} \text{ pod_data_type?}(\text{pabr_model})}$$

Expanding the definition of pod_data_type?,
 Expanding the definition of interpreted_data_type?,
 Expanding the definition of uninterpreted_data_type?,
 Applying propositional simplification,
 we get 6 subgoals:

pabr_is_pod.1:

$$\frac{}{\{1\} \forall (l: \text{list}[\text{Byte}], a: \text{Address}): \neg \text{length}(l) = \text{size}(\text{pabr_model}'\text{uidt}) \supset \neg \text{valid?}(\text{pabr_model}'\text{uidt})(l, a)}$$

Repeatedly Skolemizing and flattening,
 Expanding the definition of pabr_model,
 Expanding the definition of pabr_valid?,
 Simplifying, rewriting, and recording with decision procedures,
 This completes the proof of pabr_is_pod.1.

pabr_is_pod.2:

$$\frac{}{\{1\} \forall (d: \text{Pdbr_type}, a: \text{Address}): \text{valid?}(\text{uidt}(\text{pabr_model}))(\text{to_byte}(\text{pabr_model})(d, a), a)}$$

Applying pabr_valid_to_byte
 which is trivially true.
 This completes the proof of pabr_is_pod.2.

pabr_is_pod.3:

$$\frac{}{\{1\} \forall (l: \text{list}[\text{Byte}], a: \text{Address}): \text{valid?}(\text{uidt}(\text{pabr_model}))(l, a) \equiv \text{up?}(\text{from_byte}(\text{pabr_model})(l, a))}$$

Repeatedly Skolemizing and flattening,
 Hiding formulas: -1,
 Expanding the definition of pabr_model,
 Expanding the definition of pabr_from_byte,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 This completes the proof of pabr_is_pod.3.

pabr_is_pod.4:

$$\frac{}{\{1\} \forall (d: \text{Pdbr_type}, a: \text{Address}): \text{down}(\text{from_byte}(\text{pabr_model})(\text{to_byte}(\text{pabr_model})(d, a), a)) = d}$$

C Proof scripts

Repeatedly Skolemizing and flattening,

Using lemma `pdbr_valid_to_byte`,

Installing automatic rewrites from: `pdbr_model` `pdbr_to_byte` `pdbr_from_byte` `pdbr_valid?`
`nth` `length` `cut_bits_overwrite_bool_bit_disjoined` `cut_bits_zero` `cut_bits_overwrite_bits_disjoined`
`cut_bits_cut_bits` `cut_bits_overwrite_bits_contained` `cut_bit_overwrite_bool_bit` `overwrite_shift_bits_zero`
`overwrite_shift_bits_merge` `aligned_address_aligned`

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Adding type constraints for `d!1'base_addr`,

Expanding the definition of `max_linear_offset`,

Expanding the definition of `bus_width`,

Rewriting using `shift_bits_left_cut_bits`, matching in `*`,

we get 2 subgoals:

`pdbr_is_pod.4.1:`

$$\begin{array}{l|l}
 \{-1\} & \text{Mem?}(d' \text{'base_addr' type_of}) \\
 \{-2\} & 0 \leq d' \text{'base_addr' offset} \\
 \{-3\} & d' \text{'base_addr' offset} < \text{expt}(2, 32) \\
 \{-4\} & \text{aligned?}(12)(\text{offset}(d' \text{'base_addr})) \\
 \{-5\} & \text{cut_bits}(\text{offset}(d' \text{'base_addr}), 8, 4) = 0 \\
 \hline
 \{1\} & (\# \text{base_addr} := \text{Mem}(\text{offset}(d' \text{'base_addr})), \text{pcd} := d' \text{'pcd}, \text{pwt} := d' \text{'pwt\#}) = d'
 \end{array}$$

Applying extensionality,

Applying extensionality,

we get 2 subgoals:

`pdbr_is_pod.4.1.1:`

$$\begin{array}{l|l}
 \{-1\} & \text{Mem?}(d' \text{'base_addr' type_of}) \\
 \{-2\} & 0 \leq d' \text{'base_addr' offset} \\
 \{-3\} & d' \text{'base_addr' offset} < \text{expt}(2, 32) \\
 \{-4\} & \text{aligned?}(12)(\text{offset}(d' \text{'base_addr})) \\
 \{-5\} & \text{cut_bits}(\text{offset}(d' \text{'base_addr}), 8, 4) = 0 \\
 \hline
 \{1\} & \text{Mem}(\text{offset}(d' \text{'base_addr})) \text{'offset} = d' \text{'base_addr' offset}
 \end{array}$$

Expanding the definition of `Mem`,

which is trivially true.

This completes the proof of `pdbr_is_pod.4.1.1`.

`pdbr_is_pod.4.1.2:`

$$\begin{array}{l|l}
 \{-1\} & \text{Mem?}(d' \text{'base_addr' type_of}) \\
 \{-2\} & 0 \leq d' \text{'base_addr' offset} \\
 \{-3\} & d' \text{'base_addr' offset} < \text{expt}(2, 32) \\
 \{-4\} & \text{aligned?}(12)(\text{offset}(d' \text{'base_addr})) \\
 \{-5\} & \text{cut_bits}(\text{offset}(d' \text{'base_addr}), 8, 4) = 0 \\
 \hline
 \{1\} & \text{Mem}(\text{offset}(d' \text{'base_addr})) \text{'type_of} = d' \text{'base_addr' type_of}
 \end{array}$$

Expanding the definition of `Mem`,

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `pdbr_is_pod.4.1.2`.

pdbr_is_pod.4.2:

{-1}	Mem?(d'base_addr'type_of)
{-2}	$0 \leq d'base_addr\ offset$
{-3}	$d'base_addr\ offset < \text{expt}(2, 32)$
{-4}	aligned?(12)(offset(d'base_addr))
{-5}	cut_bits(offset(d'base_addr), 8, 4) = 0
{1}	aligned?(8)(offset(d'base_addr))
{2}	(#base_addr := Mem(shift_bits_left(cut_bits(offset(d'base_addr), 8, 24), 8)), pcd := d'pcd, pwt := d'pwt#) = d'

Applying aligned_bigger

Instantiating the top quantifier in -1 with the terms: offset(d'base_addr), 12, 8,

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of pdbr_is_pod.4.2.

pdbr_is_pod.5:

{1}	$\forall (l: \text{list}[\text{Byte}], a_1, a_2: \text{Address}):$ valid?(uidt(pdbr_model))(l, a ₁) \equiv valid?(uidt(pdbr_model))(l, a ₂)
-----	--

Expanding the definition of pdbr_model,

Expanding the definition of pdbr_valid?,

which is trivially true.

This completes the proof of pdbr_is_pod.5.

pdbr_is_pod.6:

{1}	size(uidt(pdbr_model)) > 0
-----	----------------------------

Expanding the definition of pdbr_model,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of pdbr_is_pod.6.

Q.E.D.

C.146 Proofs for PDBR_Datatype (paging-data.pvs)

C.146.1 PDBR_Datatype.pdbr_data_type_TCC1

Terse proof for pdbr_data_type_TCC1.

pdbr_data_type_TCC1:

{1}	$\exists (x: (\text{pod_data_type?}[\text{Pdbr_type}])): \text{TRUE}$
-----	--

Using lemma pdbr_data_type_exists,

This completes the proof of pdbr_data_type_TCC1.

Q.E.D.

C.147 Proofs for PDBR_Type (paging-data.pvs)

C.147.1 PDBR_Type.Pdbr_type_TCC1

Terse proof for Pdbr_type_TCC1.

Pdbr_type_TCC1:

$$\{1\} \quad \forall (a: \text{Memory_Address_4G}): \text{offset}(a) \geq 0$$

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of Pdbr_type_TCC1.

Q.E.D.

C.148 Proofs for PDE_Datatype (paging-data.pvs)

C.148.1 PDE_Datatype.pde_data_type_TCC1

Terse proof for pde_data_type_TCC1.

pde_data_type_TCC1:

$$\{1\} \quad \exists (x: (\text{pod_data_type?}[Pde_type])): \text{TRUE}$$

Using lemma pde_data_type_exists,

This completes the proof of pde_data_type_TCC1.

Q.E.D.

C.149 Proofs for PDE_Model (paging-data-models.pvs)

C.149.1 PDE_Model.pde_valid?_TCC1

Terse proof for pde_valid?_TCC1.

pde_valid?_TCC1:

$$\{1\} \quad \forall (\text{data}: \text{list}[\text{Byte}]): \text{length}(\text{data}) = 4 \supset 0 < \text{length}[\text{Byte}] (\text{data})$$

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of pde_valid?_TCC1.

Q.E.D.

C.149.2 PDE_Model.pde_valid?_TCC2

Terse proof for pde_valid?_TCC2.

pde_valid?_TCC2:

$$\{1\} \quad \forall (\text{data}: \text{list}[\text{Byte}]): \\ \text{cut_bit}(\text{nth}(\text{data}, 0), 7) \wedge \text{cut_bit}(\text{nth}(\text{data}, 0), 0) \wedge \text{length}(\text{data}) = 4 \supset \\ 1 < \text{length}[\text{Byte}] (\text{data})$$

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of pde_valid?_TCC2.

Q.E.D.

C.149.3 PDE_Model.pde_valid?_TCC3

Terse proof for `pde_valid?_TCC3`.

`pde_valid?_TCC3`:

{1}	$\forall (\text{data: list[Byte]}):$ $\text{cut_bits}(\text{nth}(\text{data}, 1), 5, 3) = 0 \wedge$ $\text{cut_bit}(\text{nth}(\text{data}, 0), 7) \wedge \text{cut_bit}(\text{nth}(\text{data}, 0), 0) \wedge \text{length}(\text{data}) = 4$ $\supset 2 < \text{length}[Byte](\text{data})$
-----	--

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `pde_valid?_TCC3`.

Q.E.D.

C.149.4 PDE_Model.pde_pte_to_byte_TCC1

Terse proof for `pde_pte_to_byte_TCC1`.

`pde_pte_to_byte_TCC1`:

{1}	$\forall (\text{pde: Pde_pt_type}):$ $\text{overwrite_bool_bit}(\text{overwrite_bool_bit}(\text{overwrite_bool_bit}(\text{overwrite_bool_bit}$ <div style="text-align: right; margin-right: 20px;"> $(\text{overwrite_bool_bit}$ $(\text{overwrite_bool_bit}$ $(0, \text{TRUE}, 0),$ mem- </div> $\text{ory_access_to_bools}$ <div style="text-align: right; margin-right: 20px;"> $(\text{pde}'\text{access})'1,$ $1),$ mem- </div> $\text{ory_privilege_to_bool}$ <div style="text-align: right; margin-right: 20px;"> $(\text{pde}'\text{privilege}),$ $2),$ $\text{pde}'\text{write_through},$ $3),$ $\text{pde}'\text{cache_disabled}, 4),$ </div> $\text{pde}'\text{accessed}, 5)$ $< \text{max_byte}$
-----	---

Repeatedly Skolemizing and flattening,

Installing automatic rewrites from: `every less_than_max_byte overwrite_bool_bit_less overwrite_shift_bits_less`

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `pde_pte_to_byte_TCC1`.

Q.E.D.

C.149.5 PDE_Model.pde_pte_to_byte_TCC2

Terse proof for `pde_pte_to_byte_TCC2`.

C Proof scripts

`pde_pte_to_byte_TCC2`:

$$\{1\} \quad \forall (pde: Pde_pt_type): \text{offset}(pde'page_table_base) \geq 0$$

Repeatedly Skolemizing and flattening,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `pde_pte_to_byte_TCC2`.

Q.E.D.

C.149.6 PDE_Model.pde_pte_to_byte_TCC3

Terse proof for `pde_pte_to_byte_TCC3`.

`pde_pte_to_byte_TCC3`:

$$\{1\} \quad \forall (pde: Pde_pt_type):$$
$$\text{every}_{[\text{nat}]}$$
$$(\{s: \text{nat} \mid s < \text{max_byte}\})$$
$$((:\text{overwrite_shift_bits}(\text{overwrite_bool_bit}(0, pde'global_page, 0),$$
$$\text{cut_bits}(\text{offset}(pde'page_table_base), 12, 4), 4, 4),$$
$$\text{cut_bits}(\text{offset}(pde'page_table_base), 16, 8),$$
$$\text{cut_bits}(\text{offset}(pde'page_table_base), 24, 8):))$$

Installing automatic rewrites from: `cut_bits_byte every`

Repeatedly Skolemizing and flattening,

Installing automatic rewrites from: `every less_than_max_byte overwrite_bool_bit_less overwrite_shift_bits_less`

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `pde_pte_to_byte_TCC3`.

Q.E.D.

C.149.7 PDE_Model.pde_super_to_byte_TCC1

Terse proof for `pde_super_to_byte_TCC1`.

pde_super_to_byte_TCC1:

<p>{1} \forall (spage: Pde_4m_type):</p> <p style="padding-left: 20px;"> overwrite_bool_bit(overwrite_bool_bit(overwrite_bool_bit(overwrite_bool_bit (overwrite_bool_bit (overwrite_bool_bit (overwrite_bool_bit (overwrite_bool_bit (0, TRUE, 0), mem- (spage' access)' 1, 1), mem- (spage' privilege), 2), spage' write_through, 3), spage' cache_disabled, 4), spage' accessed, 5), spage' dirty, 6), TRUE, 7) < max_byte </p>	
---	--

Repeatedly Skolemizing and flattening,
 Installing automatic rewrites from: every less_than_max_byte overwrite_bool_bit_less over-
 write_shift_bits_less shift_bits_left_less
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 This completes the proof of pde_super_to_byte_TCC1.
 Q.E.D.

C.149.8 PDE_Model.pde_super_to_byte_TCC2

Terse proof for pde_super_to_byte_TCC2.
 pde_super_to_byte_TCC2:

<p>{1} \forall (spage: Pde_4m_type): offset(spage'page_base) \geq 0</p>	
---	--

Repeatedly Skolemizing and flattening,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 This completes the proof of pde_super_to_byte_TCC2.
 Q.E.D.

C.149.9 PDE_Model.pde_super_to_byte_TCC3

Terse proof for pde_super_to_byte_TCC3.

pde_super_to_byte_TCC3:

<pre>{1} ∃ (spage: Pde_4m_type): every [nat] ({s: nat s < max_byte}) ((:overwrite_bool_bit(overwrite_bool_bit(0, spage'global_page, 0), spage'page_table_attribute_index, 4), shift_bits_left(cut_bits(offset(spage'page_base), 22, 2), 6), cut_bits(offset(spage'page_base), 24, 8):))</pre>

Repeatedly Skolemizing and flattening,
 Installing automatic rewrites from: every less_than_max_byte overwrite_bool_bit_less cut_bits_byte
 overwrite_shift_bits_less

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 Expanding the definition of shift_bits_left,
 Using lemma cut_bits_below,
 Expanding the definition of max_byte,
 Using lemma bits_per_byte_minimum,
 Expanding the definition of min_bits_per_byte,
 Using lemma both_sides_expt_gt1_le_aux,
 we get 2 subgoals:

pde_super_to_byte_TCC3.1:

<pre>{-1} expt(2, 7 + 1) ≤ expt(2, bits_per_byte - 1 + 1) ≡ 7 ≤ bits_per_byte - 1 {-2} bits_per_byte ≥ 8 {-3} cut_bits(offset(spage'page_base), 22, 2) < expt(2, 2)</pre>
<pre>{1} cut_bits(offset(spage'page_base), 22, 2) × expt(2, 6) < expt(2, bits_per_byte)</pre>

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 Using lemma expt_plus_aux,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of pde_super_to_byte_TCC3.1.

pde_super_to_byte_TCC3.2:

<pre>{-1} bits_per_byte ≥ 8 {-2} cut_bits(offset(spage'page_base), 22, 2) < expt(2, 2)</pre>
<pre>{1} bits_per_byte - 1 ≥ 0 {2} cut_bits(offset(spage'page_base), 22, 2) × expt(2, 6) < expt(2, bits_per_byte)</pre>

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 This completes the proof of pde_super_to_byte_TCC3.2.

Q.E.D.

C.149.10 PDE_Model.pde_to_byte_TCC1

Terse proof for pde_to_byte_TCC1.

pde_to_byte_TCC1:

<pre>{1} ∃ (pde: Pde_type): pde = Not_present ⊃ 0 < max_byte</pre>
--

Repeatedly Skolemizing and flattening,
 Installing automatic rewrites from: every less_than_max_byte expt_2_8

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of pde_to_byte_TCC1.
Q.E.D.

C.149.11 PDE_Model.pde_to_byte_TCC2

Terse proof for pde_to_byte_TCC2.

pde_to_byte_TCC2:

$\{1\} \quad \forall (\text{pde: Pde_type}):$ $\text{pde} = \text{Not_present} \supset$ $\text{every}[\text{real}]$ $(\lambda (x: \text{real}): \text{rational_pred}(x) \wedge \text{integer_pred}(x) \wedge x \geq 0 \wedge x < \text{max_byte})$ $((:0, 0, 0:))$

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of pde_to_byte_TCC2.
Q.E.D.

C.149.12 PDE_Model.pde_4m_from_byte_TCC1

Terse proof for pde_4m_from_byte_TCC1.

pde_4m_from_byte_TCC1:

$\{1\} \quad \forall (\text{data: list}[\text{Byte}], a: \text{Address}): \text{pde_valid?}(\text{data}, a) \supset 0 < \text{length}[\text{Byte}](\text{data})$

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of pde_4m_from_byte_TCC1.
Q.E.D.

C.149.13 PDE_Model.pde_4m_from_byte_TCC2

Terse proof for pde_4m_from_byte_TCC2.

pde_4m_from_byte_TCC2:

$\{1\} \quad \forall (\text{data: list}[\text{Byte}], a: \text{Address}):$ $\text{pde_valid?}(\text{data}, a) \wedge \text{cut_bit}(\text{nth}(\text{data}, 0), 0) \wedge \text{cut_bit}(\text{nth}(\text{data}, 0), 7) \supset$ $((\text{singleton} [\text{Memory_access}] (\text{Read}) \cup \{\text{Execute}\}) \subseteq \text{bools_to_memory_access}(\text{cut_bit}(\text{nth} [\text{Byte}](\text{data}, 0),$
--

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of pde_4m_from_byte_TCC2.
Q.E.D.

C.149.14 PDE_Model.pde_4m_from_byte_TCC3

Terse proof for pde_4m_from_byte_TCC3.

pde_4m_from_byte_TCC3:

$\{1\} \quad \forall (\text{data: list}[\text{Byte}], a: \text{Address}):$ $\text{pde_valid?}(\text{data}, a) \wedge \text{cut_bit}(\text{nth}(\text{data}, 0), 0) \wedge \text{cut_bit}(\text{nth}(\text{data}, 0), 7) \supset$ $1 < \text{length}[\text{Byte}](\text{data})$

C Proof scripts

Repeatedly Skolemizing and flattening,
Expanding the definition of `pde_valid?`,
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `pde_4m_from_byte_TCC3`.
Q.E.D.

C.149.15 PDE_Model.pde_4m_from_byte_TCC4

Terse proof for `pde_4m_from_byte_TCC4`.

`pde_4m_from_byte_TCC4`:

$$\{1\} \quad \forall (\text{data: list[Byte]}, a: \text{Address}):$$
$$\text{pde_valid?}(\text{data}, a) \wedge \text{cut_bit}(\text{nth}(\text{data}, 0), 0) \wedge \text{cut_bit}(\text{nth}(\text{data}, 0), 7) \supset$$
$$2 < \text{length[Byte]}(\text{data})$$

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `pde_4m_from_byte_TCC4`.
Q.E.D.

C.149.16 PDE_Model.pde_4m_from_byte_TCC5

Terse proof for `pde_4m_from_byte_TCC5`.

`pde_4m_from_byte_TCC5`:

$$\{1\} \quad \forall (\text{data: list[Byte]}, a: \text{Address}):$$
$$\text{pde_valid?}(\text{data}, a) \wedge \text{cut_bit}(\text{nth}(\text{data}, 0), 0) \wedge \text{cut_bit}(\text{nth}(\text{data}, 0), 7) \supset$$
$$3 < \text{length[Byte]}(\text{data})$$

Repeatedly Skolemizing and flattening,
Installing automatic rewrites from: `aligned_address?` `aligned_overwrite_shift_bits_more`
`aligned_overwrite_shift_bits_less` `aligned_zero` `max_linear_expt_val` `overwrite_shift_bits_less` `pde_valid?`
`bus_width` `max_linear_offset`

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `pde_4m_from_byte_TCC5`.
Q.E.D.

C.149.17 PDE_Model.pde_4m_from_byte_TCC6

Terse proof for `pde_4m_from_byte_TCC6`.

pde_4m_from_byte_TCC6:

```

{1}  ∀ (data: list[Byte], a: Address):
      pde_valid?(data, a) ∧ cut_bit(nth(data, 0), 0) ∧ cut_bit(nth(data, 0), 7) ⊃
      Mem?(Mem(overwrite_shift_bits(overwrite_shift_bits(0,
                                                    cut_bits(nth[Byte](data, 2),
                                                                6,
                                                                2),
                                                                22, 2),
                                                    nth[Byte](data, 3), 24, 8))'type_of)
      ∧
      (0 ≤
      Mem(overwrite_shift_bits(overwrite_shift_bits(0,
                                                    cut_bits(nth[Byte](data, 2),
                                                                6,
                                                                2),
                                                                22, 2),
                                                    nth[Byte](data, 3), 24, 8))'offset)
      ∧
      Mem(overwrite_shift_bits(overwrite_shift_bits(0,
                                                    cut_bits(nth[Byte](data, 2),
                                                                6,
                                                                2),
                                                                22, 2),
                                                    nth[Byte](data, 3), 24, 8))'offset)
      < max_linear_offset)
      ∧
      aligned?(22)
      (offset
      (Mem(overwrite_shift_bits(overwrite_shift_bits(0,
                                                    cut_bits
                                                    (nth[Byte](data, 2),
                                                                6,
                                                                2),
                                                                22,
                                                                2),
                                                    nth[Byte](data, 3), 24, 8))))

```

Repeatedly Skolemizing and flattening,

Installing automatic rewrites from: aligned_address? aligned_overwrite_shift_bits_more
aligned_overwrite_shift_bits_less aligned_zero max_linear_expt_val overwrite_shift_bits_less pde_valid?
bus_width max_linear_offset

Expanding the definition of Mem,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of pde_4m_from_byte_TCC6.

Q.E.D.

C.149.18 PDE_Model.pde_4k_from_byte_TCC1

Terse proof for pde_4k_from_byte_TCC1.

pde_4k_from_byte_TCC1:

$$\{1\} \quad \forall (data: \text{list}[\text{Byte}], a: \text{Address}):$$

$$\text{pde_valid?}(data, a) \wedge \text{cut_bit}(\text{nth}(data, 0), 0) \wedge \neg \text{cut_bit}(\text{nth}(data, 0), 7) \supset$$

$$((\text{singleton} [\text{Memory_access}](\text{Read}) \cup \{\text{Execute}\}) \subseteq \text{bools_to_memory_access}(\text{cut_bit}(\text{nth} [\text{Byte}]$$

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of pde_4k_from_byte_TCC1.
Q.E.D.

C.149.19 PDE_Model.pde_4k_from_byte_TCC2

Terse proof for pde_4k_from_byte_TCC2.

pde_4k_from_byte_TCC2:

$$\{1\} \quad \forall (data: \text{list}[\text{Byte}], a: \text{Address}):$$

$$\text{pde_valid?}(data, a) \wedge \text{cut_bit}(\text{nth}(data, 0), 0) \wedge \neg \text{cut_bit}(\text{nth}(data, 0), 7) \supset$$

$$1 < \text{length}[\text{Byte}](data)$$

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of pde_4k_from_byte_TCC2.
Q.E.D.

C.149.20 PDE_Model.pde_4k_from_byte_TCC3

Terse proof for pde_4k_from_byte_TCC3.

pde_4k_from_byte_TCC3:

$$\{1\} \quad \forall (data: \text{list}[\text{Byte}], a: \text{Address}):$$

$$\text{pde_valid?}(data, a) \wedge \text{cut_bit}(\text{nth}(data, 0), 0) \wedge \neg \text{cut_bit}(\text{nth}(data, 0), 7) \supset$$

$$2 < \text{length}[\text{Byte}](data)$$

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of pde_4k_from_byte_TCC3.
Q.E.D.

C.149.21 PDE_Model.pde_4k_from_byte_TCC4

Terse proof for pde_4k_from_byte_TCC4.

pde_4k_from_byte_TCC4:

$$\{1\} \quad \forall (data: \text{list}[\text{Byte}], a: \text{Address}):$$

$$\text{pde_valid?}(data, a) \wedge \text{cut_bit}(\text{nth}(data, 0), 0) \wedge \neg \text{cut_bit}(\text{nth}(data, 0), 7) \supset$$

$$3 < \text{length}[\text{Byte}](data)$$

Repeatedly Skolemizing and flattening,
Installing automatic rewrites from: aligned_address? aligned_overwrite_shift_bits_more
aligned_overwrite_shift_bits_less aligned_zero max_linear_expt_val overwrite_shift_bits_less pde_valid?
max_linear_offset bus_width
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of pde_4k_from_byte_TCC4.

Q.E.D.

C.149.22 PDE_Model.pde_4k_from_byte_TCC5

Terse proof for pde_4k_from_byte_TCC5.

pde_4k_from_byte_TCC5:

```

{1}  ∀ (data: list[Byte], a: Address):
      pde_valid?(data, a) ∧ cut_bit(nth(data, 0), 0) ∧ ¬ cut_bit(nth(data, 0), 7) ⊃
      Mem?(Mem(overwrite_shift_bits(overwrite_shift_bits(overwrite_shift_bits(0,
      nth[Byte](data, 2), 16, 8),
      nth[Byte](data, 3), 24, 8))'type_of)
      ∧
      (0 ≤
      Mem(overwrite_shift_bits(overwrite_shift_bits(overwrite_shift_bits(0,
      nth[Byte](data, 2), 16, 8),
      nth[Byte](data, 3), 24, 8))'offset
      ∧
      Mem(overwrite_shift_bits(overwrite_shift_bits(overwrite_shift_bits(0,
      nth[Byte](data, 2), 16, 8),
      nth[Byte](data, 3), 24, 8))'offset
      < max_linear_offset)
      ∧
      aligned?(12)
      (offset
      (Mem(overwrite_shift_bits(overwrite_shift_bits(overwrite_shift_bits
      (0,
      cut_bits
      (nth[Byte](data
      4,
      4),
      12,
      4),
      nth[Byte](data, 2
      16,
      8),
      8),
      nth[Byte](data, 3), 24, 8))))

```

Repeatedly Skolemizing and flattening,
 Installing automatic rewrites from: aligned_address? aligned_overwrite_shift_bits_more
 aligned_overwrite_shift_bits_less aligned_zero max_linear_expt_val overwrite_shift_bits_less pde_valid?
 bus_width max_linear_offset
 Expanding the definition of Mem,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 This completes the proof of pde_4k_from_byte_TCC5.
 Q.E.D.

C.149.23 PDE_Model.pde_from_byte_TCC1

Terse proof for pde_from_byte_TCC1.

pde_from_byte_TCC1:

$$\{1\} \quad \forall (\text{data}: \text{list}[\text{Byte}], a: \text{Address}):$$

$$\text{cut_bit}(\text{nth}(\text{data}, 0), 7) \wedge \text{cut_bit}(\text{nth}(\text{data}, 0), 0) \wedge \text{pde_valid}?(a, \text{data}) \supset$$

$$\text{up}?[Pde_4m_type](\text{pde_4m_from_byte}(\text{data}, a))$$

Repeatedly Skolemizing and flattening,
 Expanding the definition of pde_4m_from_byte,
 Simplifying, rewriting, and recording with decision procedures,
 This completes the proof of pde_from_byte_TCC1.
 Q.E.D.

C.149.24 PDE_Model.pde_from_byte_TCC2

Terse proof for pde_from_byte_TCC2.

pde_from_byte_TCC2:

$$\{1\} \quad \forall (\text{data}: \text{list}[\text{Byte}], a: \text{Address}):$$

$$\neg \text{cut_bit}(\text{nth}(\text{data}, 0), 7) \wedge \text{cut_bit}(\text{nth}(\text{data}, 0), 0) \wedge \text{pde_valid}?(a, \text{data}) \supset$$

$$\text{up}?[Pde_pt_type](\text{pde_4k_from_byte}(\text{data}, a))$$

Repeatedly Skolemizing and flattening,
 Expanding the definition of pde_4k_from_byte,
 Simplifying, rewriting, and recording with decision procedures,
 This completes the proof of pde_from_byte_TCC2.
 Q.E.D.

C.149.25 PDE_Model.pde_model_TCC1

Terse proof for pde_model_TCC1.

pde_model_TCC1:

$$\{1\} \quad \forall (f: [[Pde_type, \text{Address}] \rightarrow [\text{list}[\text{Byte}] \rightarrow \text{bool}]]):$$

$$(f = (\lambda (d: Pde_type, a: \text{Address}): (\text{zero_mask}?(4)))) \supset$$

$$(\forall (x_1: [Pde_type, \text{Address}]): \text{singleton}?[[\text{list}[\text{Byte}]]](f(x_1)))$$

Repeatedly Skolemizing and flattening,
 Replacing using formula -1,
 Using lemma singleton_zero_mask,

C Proof scripts

Expanding the definition of singleton?,
which is trivially true.
This completes the proof of `pde_model_TCC1`.
Q.E.D.

C.149.26 PDE_Model.pde_valid_to_byte

The \LaTeX code for this proof is broken.

C.149.27 PDE_Model.pde_is_pod

Terse proof for `pde_is_pod`.

`pde_is_pod`:

$$\{1\} \quad \text{pod_data_type?}(\text{pde_model})$$

Expanding the definition of `pod_data_type?`,
Expanding the definition of `interpreted_data_type?`,
Expanding the definition of `uninterpreted_data_type?`,
Applying propositional simplification,
we get 6 subgoals:

`pde_is_pod.1`:

$$\{1\} \quad \forall (l: \text{list}[\text{Byte}], a: \text{Address}):$$
$$\neg \text{length}(l) = \text{size}(\text{pde_model}'\text{uidt}) \supset \neg \text{valid?}(\text{pde_model}'\text{uidt})(l, a)$$

Repeatedly Skolemizing and flattening,
Expanding the definition of `pde_model`,
Expanding the definition of `pde_valid?`,
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `pde_is_pod.1`.

`pde_is_pod.2`:

$$\{1\} \quad \forall (d: \text{Pde_type}, a: \text{Address}): \text{valid?}(\text{uidt}(\text{pde_model}))(\text{to_byte}(\text{pde_model})(d, a), a)$$

Repeatedly Skolemizing and flattening,
Rewriting using `pde_valid_to_byte`, matching in *,
This completes the proof of `pde_is_pod.2`.

`pde_is_pod.3`:

$$\{1\} \quad \forall (l: \text{list}[\text{Byte}], a: \text{Address}):$$
$$\text{valid?}(\text{uidt}(\text{pde_model}))(l, a) \equiv \text{up?}(\text{from_byte}(\text{pde_model}))(l, a)$$

Repeatedly Skolemizing and flattening,
Expanding the definition of `pde_model`,
Expanding the definition of `pde_from_byte`,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `pde_is_pod.3`.

pde_is_pod.4:

$$\frac{}{\{1\} \quad \forall (d: \text{Pde_type}, a: \text{Address}): \quad \text{down}(\text{from_byte}(\text{pde_model}))(\text{to_byte}(\text{pde_model})(d, a), a) = d}$$

Repeatedly Skolemizing and flattening,
 Expanding the definition of pde_model,
 Expanding the definition of pde_to_byte,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 we get 3 subgoals:

pde_is_pod.4.1:

$$\frac{\{-1\} \quad \text{not_present?}(d')}{\{1\} \quad \text{down}(\text{pde_from_byte}(:0, 0, 0, 0), a') = d'}$$

Installing automatic rewrites from: pde_from_byte! pde_valid? nth length length_cdr
 length_is_cons cut_bits_overwrite_bool_bit_disjoined cut_bits_zero cut_bits_overwrite_bits_disjoined
 cut_bits_cut_bits cut_bits_overwrite_bits_contained cut_bit_overwrite_bool_bit overwrite_shift_bits_zero
 overwrite_shift_bits_merge aligned_address_aligned cut_bit_cut_bits cut_bit_overwrite_shift_bits
 length_to_byte from_byte_to_byte valid_to_byte cut_bit_zero

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of pde_is_pod.4.1.

pde_is_pod.4.2:

$$\frac{\{-1\} \quad \text{pde_pt?}(d')}{\{1\} \quad \text{down}(\text{pde_from_byte}(\text{pde_pte_to_byte}(\text{pde_pt}(d'), a'), a')) = d'}$$

Installing automatic rewrites from: pde_pte_to_byte! pde_4k_from_byte pde_from_byte!
 pde_valid? nth length length_cdr length_is_cons cut_bits_overwrite_bool_bit_disjoined cut_bits_zero
 cut_bits_overwrite_bits_disjoined cut_bits_cut_bits cut_bits_overwrite_bits_contained cut_bit_overwrite_bool_bit
 overwrite_shift_bits_zero overwrite_shift_bits_merge aligned_address_aligned cut_bit_cut_bits
 cut_bit_overwrite_shift_bits cut_bits_overwrite_shift_bits_contained length_to_byte from_byte_to_byte
 valid_to_byte cut_bit_zero bool_to_memory_access_iso bool_to_memory_privilege_iso max_linear_offset
 bus_width

Adding type constraints for pde_pt(d!)'page_table_base,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Rewriting using shift_bits_left_cut_bits, matching in *,

Applying extensionality,

we get 3 subgoals:

pde_is_pod.4.2.1:

$$\frac{\begin{array}{l} \{-1\} \quad \text{Mem?}(\text{pde_pt}(d') \text{'page_table_base'} \text{'type_of}) \\ \{-2\} \quad 0 \leq \text{pde_pt}(d') \text{'page_table_base'} \text{'offset} \\ \{-3\} \quad \text{pde_pt}(d') \text{'page_table_base'} \text{'offset} < \text{expt}(2, 32) \\ \{-4\} \quad \text{aligned?}(12)(\text{offset}(\text{pde_pt}(d') \text{'page_table_base})) \\ \{-5\} \quad \text{pde_pt?}(d') \end{array}}{\{1\} \quad (\# \text{page_table_base} := \text{Mem}(\text{offset}(\text{pde_pt}(d') \text{'page_table_base'})), \\ \text{global_page} := \text{pde_pt}(d') \text{'global_page}, \\ \text{accessed} := \text{pde_pt}(d') \text{'accessed}, \\ \text{cache_disabled} := \text{pde_pt}(d') \text{'cache_disabled}, \\ \text{write_through} := \text{pde_pt}(d') \text{'write_through}, \\ \text{privilege} := \text{pde_pt}(d') \text{'privilege}, \\ \text{access} := \text{bools_to_memory_access}(\text{memory_access_to_bools}(\text{pde_pt}(d') \text{'access'}) \text{'1}, \text{TRUE}) \#) \\ = \text{pde_pt}(d')}$$

C Proof scripts

Expanding the definition of `bools_to_memory_access`,

Expanding the definition of `Mem`,

Expanding the definition of `memory_access_to_bools`,

Applying extensionality,

we get 2 subgoals:

`pde_is_pod.4.2.1.1:`

{-1}	<code>Mem?(pde_pt(d') 'page_table_base' type_of)</code>
{-2}	<code>0 ≤ pde_pt(d') 'page_table_base' offset</code>
{-3}	<code>pde_pt(d') 'page_table_base' offset < expt(2, 32)</code>
{-4}	<code>aligned?(12)(offset(pde_pt(d') 'page_table_base))</code>
{-5}	<code>pde_pt?(d')</code>
{1}	<code>IF (Write ∈ pde_pt(d') 'access)</code> <code> THEN ((singleton(Read) ∪ {Write}) ∪ {Execute})</code> <code> ELSE (singleton(Read) ∪ {Execute})</code> <code>ENDIF</code> <code>= pde_pt(d') 'access</code>

Adding type constraints for `pde_pt(d!1) 'access`,

Keeping (-1 1) and hiding *,

Applying extensionality,

Expanding the definition of `subset?`,

Instantiating the top quantifier in -1 with the terms: x' ,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `pde_is_pod.4.2.1.1`.

`pde_is_pod.4.2.1.2:`

{-1}	<code>Mem?(pde_pt(d') 'page_table_base' type_of)</code>
{-2}	<code>0 ≤ pde_pt(d') 'page_table_base' offset</code>
{-3}	<code>pde_pt(d') 'page_table_base' offset < expt(2, 32)</code>
{-4}	<code>aligned?(12)(offset(pde_pt(d') 'page_table_base))</code>
{-5}	<code>pde_pt?(d')</code>
{1}	<code>(#type_of := Mem_, offset := offset(pde_pt(d') 'page_table_base)#) =</code> <code>pde_pt(d') 'page_table_base</code>

Applying extensionality,

This completes the proof of `pde_is_pod.4.2.1.2`.

`pde_is_pod.4.2.2:`

{-1}	<code>Mem?(pde_pt(d') 'page_table_base' type_of)</code>
{-2}	<code>0 ≤ pde_pt(d') 'page_table_base' offset</code>
{-3}	<code>pde_pt(d') 'page_table_base' offset < expt(2, 32)</code>
{-4}	<code>aligned?(12)(offset(pde_pt(d') 'page_table_base))</code>
{-5}	<code>pde_pt?(d')</code>
{1}	<code>Mem?(Mem(offset(pde_pt(d') 'page_table_base)) 'type_of) ∧</code> <code> (0 ≤ Mem(offset(pde_pt(d') 'page_table_base)) 'offset ∧</code> <code> Mem(offset(pde_pt(d') 'page_table_base)) 'offset < expt(2, 32))</code> <code> ∧ aligned?(12)(offset(Mem(offset(pde_pt(d') 'page_table_base))))</code>

Expanding the definition of `Mem`,

which is trivially true.

This completes the proof of `pde_is_pod.4.2.2`.

pde_is_pod.4.2.3:

{-1}	Mem?(pde_pt(d') 'page_table_base' type_of)
{-2}	$0 \leq \text{pde_pt}(d') \text{'page_table_base' offset}$
{-3}	$\text{pde_pt}(d') \text{'page_table_base' offset} < \text{expt}(2, 32)$
{-4}	$\text{aligned?}(12)(\text{offset}(\text{pde_pt}(d') \text{'page_table_base}))$
{-5}	$\text{pde_pt?}(d')$
{1}	$((\text{singleton}[\text{Memory_access}] (\text{Read}) \cup \{\text{Execute}\}) \subseteq \text{bools_to_memory_access}(\text{memory_access_to_bools}(\text{pde_pt}(d')$

Adding type constraints for pde_pt(d!) 'access,

Keeping (-1 1) and hiding *,

Expanding the definition of subset?,

Repeatedly Skolemizing and flattening,

Instantiating the top quantifier in -1 with the terms: x',

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of pde_is_pod.4.2.3.

pde_is_pod.4.3:

{1}	$\text{not_present?}(d')$
{2}	$\text{pde_pt?}(d')$
{3}	$\text{down}(\text{pde_from_byte}(\text{pde_super_to_byte}(\text{pde_4m}(d'), a'), a')) = d'$

Installing automatic rewrites from: pde_super_to_byte! pde_4m_from_byte pde_from_byte!
pde_valid? nth length length_cdr length_is_cons cut_bits_overwrite_bool_bit_disjoined cut_bits_zero
cut_bits_overwrite_bits_disjoined cut_bits_cut_bits cut_bits_overwrite_bits_contained cut_bit_overwrite_bool_bit
overwrite_shift_bits_zero overwrite_shift_bits_merge aligned_address_aligned cut_bit_cut_bits
cut_bit_overwrite_shift_bits cut_bits_overwrite_shift_bits_contained cut_bits_overwrite_shift_bits_disjoint
cut_bits_shift_bits_left_outside cut_bits_shift_bits_left_in length_to_byte from_byte_to_byte valid_to_byte
cut_bit_zero bool_to_memory_access_iso bool_to_memory_privilege_iso max_linear_offset bus_width

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Adding type constraints for pde_4m(d!) 'page_base,

Rewriting using shift_bits_left_cut_bits, matching in *,

Applying extensionality,

we get 3 subgoals:

pde_is_pod.4.3.1:

{-1}	Mem?(pde_4m(d') 'page_base' type_of)
{-2}	$0 \leq \text{pde_4m}(d') \text{'page_base' offset}$
{-3}	$\text{pde_4m}(d') \text{'page_base' offset} < \text{expt}(2, 32)$
{-4}	$\text{aligned?}(22)(\text{offset}(\text{pde_4m}(d') \text{'page_base}))$
{1}	$(\# \text{page_base} := \text{Mem}(\text{offset}(\text{pde_4m}(d') \text{'page_base})),$ $\text{page_table_attribute_index} := \text{pde_4m}(d') \text{'page_table_attribute_index},$ $\text{global_page} := \text{pde_4m}(d') \text{'global_page},$ $\text{dirty} := \text{pde_4m}(d') \text{'dirty},$ $\text{accessed} := \text{pde_4m}(d') \text{'accessed},$ $\text{cache_disabled} := \text{pde_4m}(d') \text{'cache_disabled},$ $\text{write_through} := \text{pde_4m}(d') \text{'write_through},$ $\text{privilege} := \text{pde_4m}(d') \text{'privilege},$ $\text{access} := \text{bools_to_memory_access}(\text{memory_access_to_bools}(\text{pde_4m}(d') \text{'access'}) \text{'1, TRUE}) \#)$ $= \text{pde_4m}(d')$
{2}	$\text{not_present?}(d')$
{3}	$\text{pde_pt?}(d')$

Applying extensionality,

C Proof scripts

we get 2 subgoals:

`pde_is_pod.4.3.1.1:`

{-1}	<code>Mem?(pde_4m(d') 'page_base' type_of)</code>
{-2}	<code>0 ≤ pde_4m(d') 'page_base' offset</code>
{-3}	<code>pde_4m(d') 'page_base' offset < expt(2, 32)</code>
{-4}	<code>aligned?(22)(offset(pde_4m(d') 'page_base'))</code>
{1}	<code>bools_to_memory_access(memory_access_to_bools(pde_4m(d') 'access') '1, TRUE) = pde_4m(d') 'access</code>
{2}	<code>not_present?(d')</code>
{3}	<code>pde_pt?(d')</code>

Expanding the definition of `bools_to_memory_access`,

Expanding the definition of `memory_access_to_bools`,

Adding type constraints for `pde_4m(d!1)'access`,

Applying extensionality,

Expanding the definition of `subset?`,

Instantiating the top quantifier in -1 with the terms: x' ,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `pde_is_pod.4.3.1.1`.

`pde_is_pod.4.3.1.2:`

{-1}	<code>Mem?(pde_4m(d') 'page_base' type_of)</code>
{-2}	<code>0 ≤ pde_4m(d') 'page_base' offset</code>
{-3}	<code>pde_4m(d') 'page_base' offset < expt(2, 32)</code>
{-4}	<code>aligned?(22)(offset(pde_4m(d') 'page_base'))</code>
{1}	<code>Mem(offset(pde_4m(d') 'page_base')) = pde_4m(d') 'page_base</code>
{2}	<code>not_present?(d')</code>
{3}	<code>pde_pt?(d')</code>

Expanding the definition of `Mem`,

Applying extensionality,

This completes the proof of `pde_is_pod.4.3.1.2`.

`pde_is_pod.4.3.2:`

{-1}	<code>Mem?(pde_4m(d') 'page_base' type_of)</code>
{-2}	<code>0 ≤ pde_4m(d') 'page_base' offset</code>
{-3}	<code>pde_4m(d') 'page_base' offset < expt(2, 32)</code>
{-4}	<code>aligned?(22)(offset(pde_4m(d') 'page_base'))</code>
{1}	<code>Mem?(Mem(offset(pde_4m(d') 'page_base')) 'type_of) ∧ (0 ≤ Mem(offset(pde_4m(d') 'page_base)) 'offset ∧ Mem(offset(pde_4m(d') 'page_base)) 'offset < expt(2, 32)) ∧ aligned?(22)(offset(Mem(offset(pde_4m(d') 'page_base))))</code>
{2}	<code>not_present?(d')</code>
{3}	<code>pde_pt?(d')</code>

Expanding the definition of `Mem`,

which is trivially true.

This completes the proof of `pde_is_pod.4.3.2`.

pde_is_pod.4.3.3:

{-1}	Mem?(pde_4m(d') 'page_base' type_of)
{-2}	$0 \leq \text{pde_4m}(d') \text{'page_base' offset}$
{-3}	$\text{pde_4m}(d') \text{'page_base' offset} < \text{expt}(2, 32)$
{-4}	aligned?(22)(offset(pde_4m(d') 'page_base'))
{1}	$((\text{singleton}[\text{Memory_access}](\text{Read}) \cup \{\text{Execute}\}) \subseteq \text{bools_to_memory_access}(\text{memory_access_to_bools}(\text{pde_4m}(d'))))$
{2}	not_present?(d')
{3}	pde_pt?(d')

Keeping (-1 1) and hiding *,
 Expanding the definition of subset?,
 Repeatedly Skolemizing and flattening,
 Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of pde_is_pod.4.3.3.

pde_is_pod.5:

{1}	$\forall (l: \text{list}[\text{Byte}], a_1, a_2: \text{Address}): \text{valid?}(\text{uidt}(\text{pde_model}))(l, a_1) \equiv \text{valid?}(\text{uidt}(\text{pde_model}))(l, a_2)$
-----	---

Repeatedly Skolemizing and flattening,
 Hiding formulas: -1,
 Expanding the definition of pde_model,
 Expanding the definition of pde_valid?,
 which is trivially true.
 This completes the proof of pde_is_pod.5.

pde_is_pod.6:

{1}	$\text{size}(\text{uidt}(\text{pde_model})) > 0$
-----	---

Expanding the definition of pde_model,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 This completes the proof of pde_is_pod.6.
 Q.E.D.

C.150 Proofs for PF_EC_Model (paging-data-models.pvs)

C.150.1 PF_EC_Model.pf_ec_valid?_TCC1

Terse proof for pf_ec_valid?_TCC1.

pf_ec_valid?_TCC1:

{1}	$\forall (\text{data}: \text{list}[\text{Byte}]): \text{length}(\text{data}) = 4 \supset 3 < \text{length}[\text{Byte}](\text{data})$
-----	---

Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of pf_ec_valid?_TCC1.
 Q.E.D.

C.150.2 PF_EC_Model.pf_ec_valid?_TCC2

Terse proof for pf_ec_valid?_TCC2.

pf_ec_valid?_TCC2:

$$\{1\} \quad \forall (data: \text{list}[\text{Byte}]):$$
$$\text{cut_bits}(\text{nth}(\text{data}, 3), 0, 8) = 0 \wedge \text{length}(\text{data}) = 4 \supset 2 < \text{length}[\text{Byte}](\text{data})$$

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of pf_ec_valid?_TCC2.

Q.E.D.

C.150.3 PF_EC_Model.pf_ec_valid?_TCC3

Terse proof for pf_ec_valid?_TCC3.

pf_ec_valid?_TCC3:

$$\{1\} \quad \forall (data: \text{list}[\text{Byte}]):$$
$$\text{cut_bits}(\text{nth}(\text{data}, 2), 0, 8) = 0 \wedge$$
$$\text{cut_bits}(\text{nth}(\text{data}, 3), 0, 8) = 0 \wedge \text{length}(\text{data}) = 4$$
$$\supset 1 < \text{length}[\text{Byte}](\text{data})$$

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of pf_ec_valid?_TCC3.

Q.E.D.

C.150.4 PF_EC_Model.pf_ec_valid?_TCC4

Terse proof for pf_ec_valid?_TCC4.

pf_ec_valid?_TCC4:

$$\{1\} \quad \forall (data: \text{list}[\text{Byte}]):$$
$$\text{cut_bits}(\text{nth}(\text{data}, 1), 0, 8) = 0 \wedge$$
$$\text{cut_bits}(\text{nth}(\text{data}, 2), 0, 8) = 0 \wedge$$
$$\text{cut_bits}(\text{nth}(\text{data}, 3), 0, 8) = 0 \wedge \text{length}(\text{data}) = 4$$
$$\supset 0 < \text{length}[\text{Byte}](\text{data})$$

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of pf_ec_valid?_TCC4.

Q.E.D.

C.150.5 PF_EC_Model.pf_ec_to_byte_TCC1

Terse proof for pf_ec_to_byte_TCC1.

pf_ec_to_byte_TCC1:

{1}	\forall (pf_ec: PF_error_code_type):	
	$\text{overwrite_bool_bit}(\text{overwrite_bool_bit}(\text{overwrite_bool_bit}(\text{overwrite_bool_bit}$	
		$(\text{overwrite_bool_bit}$
		$(0, \text{pf_ec}'\text{present}, 0),$
		$(\text{Write} \in \text{pf_ec}'\text{access}),$
		$1),$
	$\text{ory_privilege_to_bool}$	mem-
		$(\text{pf_ec}'\text{privilege}),$
		$2),$
		$\text{pf_ec}'\text{reserved_bit_violation}, 3),$
	$(\text{Execute} \in \text{pf_ec}'\text{access}), 4)$	
	$< \text{max_byte}$	

Repeatedly Skolemizing and flattening,

Installing automatic rewrites from: every less_than_max_byte overwrite_bool_bit_less

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of pf_ec_to_byte_TCC1.

Q.E.D.

C.150.6 PF_EC_Model.pf_ec_to_byte_TCC2

Terse proof for pf_ec_to_byte_TCC2.

pf_ec_to_byte_TCC2:

{1}	\forall (pf_ec: PF_error_code_type):	
	$\text{every}[\text{real}]$	
	$(\lambda (x: \text{real}): \text{rational_pred}(x) \wedge \text{integer_pred}(x) \wedge x \geq 0 \wedge x < \text{max_byte})$	
	$((:0, 0, 0:))$	

Expanding the definition of max_byte,

Adding type constraints for expt(2, bits_per_byte),

Installing automatic rewrites from: every

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of pf_ec_to_byte_TCC2.

Q.E.D.

C.150.7 PF_EC_Model.pf_ec_from_byte_TCC1

Terse proof for pf_ec_from_byte_TCC1.

pf_ec_from_byte_TCC1:

{1}	\forall (data: list[Byte], a: Address): pf_ec_valid?(data, a) \supset $0 < \text{length}[\text{Byte}](\text{data})$	
-----	---	--

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of pf_ec_from_byte_TCC1.

Q.E.D.

C.150.8 PF_EC_Model.pf_ec_from_byte_TCC2

Terse proof for pf_ec_from_byte_TCC2.

pf_ec_from_byte_TCC2:

$$\{1\} \quad \forall (data: \text{list}[\text{Byte}], a: \text{Address}):$$

$$\text{pf_ec_valid?}(data, a) \supset$$

$$(\text{Read} \in \text{bools_to_memory_access}(\text{cut_bit}(\text{nth}[\text{Byte}](data, 0), 1), \text{cut_bit}(\text{nth}[\text{Byte}](data, 0))))$$

Repeatedly Skolemizing and flattening,
 Expanding the definition of every,
 Expanding the definition of bools_to_memory_access,
 Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of pf_ec_from_byte_TCC2.

Q.E.D.

C.150.9 PF_EC_Model.pf_ec_model_TCC1

Terse proof for pf_ec_model_TCC1.

pf_ec_model_TCC1:

$$\{1\} \quad \forall (f: [[\text{PF_error_code_type}, \text{Address}] \rightarrow [\text{list}[\text{Byte}] \rightarrow \text{bool}]]):$$

$$(f = (\lambda (d: \text{PF_error_code_type}, a: \text{Address}): (\text{zero_mask?}(4)))) \supset$$

$$(\forall (x_1: [\text{PF_error_code_type}, \text{Address}]): \text{singleton?}[\text{list}[\text{Byte}]](f(x_1)))$$

Repeatedly Skolemizing and flattening,
 Replacing using formula -1,
 Using lemma singleton_zero_mask,
 Expanding the definition of singleton?,
 which is trivially true.
 This completes the proof of pf_ec_model_TCC1.

Q.E.D.

C.150.10 PF_EC_Model.pf_ec_valid_to_byte

Terse proof for pf_ec_valid_to_byte.

pf_ec_valid_to_byte:

$$\{1\} \quad \forall (d: \text{PF_error_code_type}, a: \text{Address}):$$

$$\text{valid?}(\text{uidt}(\text{pf_ec_model}))(\text{to_byte}(\text{pf_ec_model})(d, a), a)$$

Repeatedly Skolemizing and flattening,
 Installing automatic rewrites from: pf_ec_model pf_ec_to_byte pf_ec_valid? nth length cut_bits_zero
 cut_bits_overwrite_bool_bit_disjoined cut_bits_overwrite_bits_disjoined cut_bits_cut_bits
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 This completes the proof of pf_ec_valid_to_byte.

Q.E.D.

C.150.11 PF_EC_Model.pf_ec_is_pod

Terse proof for pf_ec_is_pod.

pf_ec_is_pod:

$$\frac{}{\{1\} \text{ pod_data_type?}(pf_ec_model)}$$

Expanding the definition of pod_data_type?,
 Expanding the definition of interpreted_data_type?,
 Expanding the definition of uninterpreted_data_type?,
 Applying propositional simplification,
 we get 6 subgoals:

pf_ec_is_pod.1:

$$\frac{}{\{1\} \forall (l: \text{list}[\text{Byte}], a: \text{Address}): \neg \text{length}(l) = \text{size}(pf_ec_model'uidt) \supset \neg \text{valid?}(pf_ec_model'uidt)(l, a)}$$

Repeatedly Skolemizing and flattening,
 Expanding the definition of pf_ec_model,
 Expanding the definition of pf_ec_valid?,
 Simplifying, rewriting, and recording with decision procedures,
 This completes the proof of pf_ec_is_pod.1.

pf_ec_is_pod.2:

$$\frac{}{\{1\} \forall (d: \text{PF_error_code_type}, a: \text{Address}): \text{valid?}(uidt(pf_ec_model))(to_byte(pf_ec_model)(d, a), a)}$$

Repeatedly Skolemizing and flattening,
 Rewriting using pf_ec_valid_to_byte, matching in *,
 This completes the proof of pf_ec_is_pod.2.

pf_ec_is_pod.3:

$$\frac{}{\{1\} \forall (l: \text{list}[\text{Byte}], a: \text{Address}): \text{valid?}(uidt(pf_ec_model))(l, a) \equiv \text{up?}(\text{from_byte}(pf_ec_model)(l, a))}$$

Repeatedly Skolemizing and flattening,
 Hiding formulas: -1,
 Expanding the definition of pf_ec_model,
 Expanding the definition of pf_ec_from_byte,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 This completes the proof of pf_ec_is_pod.3.

pf_ec_is_pod.4:

$$\frac{}{\{1\} \forall (d: \text{PF_error_code_type}, a: \text{Address}): \text{down}(\text{from_byte}(pf_ec_model)(to_byte(pf_ec_model)(d, a), a)) = d}$$

Repeatedly Skolemizing and flattening,
 Using lemma pf_ec_valid_to_byte,
 Installing automatic rewrites from: pf_ec_model pf_ec_to_byte pf_ec_from_byte pf_ec_valid? nth length
 cut_bits_overwrite_bool_bit_disjoined cut_bits_zero cut_bits_overwrite_bits_disjoined cut_bits_cut_bits
 cut_bits_overwrite_bits_contained cut_bit_overwrite_bool_bit bool_to_memory_access_iso bool_to_memory_privilege_iso
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 Applying decompose-equality,
 Expanding the definition of bools_to_memory_access,
 Keeping (1) and hiding *,

Adding type constraints for d!l'access,
 Applying extensionality,
 Hiding formulas: 2,
 Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of `pf_ec_is_pod.4`.
`pf_ec_is_pod.5`:

{1}	$\forall (l: \text{list}[\text{Byte}], a_1, a_2: \text{Address}):$ $\text{valid?}(\text{uidt}(\text{pf_ec_model}))(l, a_1) \equiv \text{valid?}(\text{uidt}(\text{pf_ec_model}))(l, a_2)$
-----	--

Expanding the definition of `pf_ec_model`,
 Expanding the definition of `pf_ec_valid?`,
 which is trivially true.
 This completes the proof of `pf_ec_is_pod.5`.
`pf_ec_is_pod.6`:

{1}	$\text{size}(\text{uidt}(\text{pf_ec_model})) > 0$
-----	--

Expanding the definition of `pf_ec_model`,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 This completes the proof of `pf_ec_is_pod.6`.
 Q.E.D.

C.151 Proofs for PF_Error_Code_Type (paging-data.pvs)

C.151.1 PF_Error_Code_Type.pf_ec_data_type_TCC1

Terse proof for `pf_ec_data_type_TCC1`.
`pf_ec_data_type_TCC1`:

{1}	$\exists (x: (\text{pod_data_type?}[\text{PF_error_code_type}])): \text{TRUE}$
-----	---

Using lemma `pf_ec_type_exists`,
 This completes the proof of `pf_ec_data_type_TCC1`.
 Q.E.D.

C.152 Proofs for PTE_Datatype (paging-data.pvs)

C.152.1 PTE_Datatype.pte_data_type_TCC1

Terse proof for `pte_data_type_TCC1`.
`pte_data_type_TCC1`:

{1}	$\exists (x: (\text{pod_data_type?}[\text{Pte_type}])): \text{TRUE}$
-----	---

Using lemma `pte_data_type_exists`,
 This completes the proof of `pte_data_type_TCC1`.
 Q.E.D.

C.153 Proofs for PTE_Model (paging-data-models.pvs)

C.153.1 PTE_Model.pte_page_to_byte_TCC1

Terse proof for pte_page_to_byte_TCC1.

pte_page_to_byte_TCC1:

<p>{1} \forall (pte: Pte_4k_type):</p> <p> overwrite_bool_bit(overwrite_bool_bit(overwrite_bool_bit(overwrite_bool_bit</p> <p> ory_access_to_bools</p> <p> ory_privilege_to_bool</p> <p> < max_byte</p>	<p>(overwrite_bool_bit</p> <p> (overwrite_bool_bit</p> <p> (overwrite_bool_bit</p> <p> (overwrite_bool_bit</p> <p> (0, TRUE, 0),</p> <p> mem-</p> <p> (pte'access)'1,</p> <p> 1),</p> <p> mem-</p> <p> (pte'privilege),</p> <p> 2),</p> <p> pte'write_through,</p> <p> 3),</p> <p> pte'cache_disabled,</p> <p> 4),</p> <p> pte'accessed,</p> <p> 5),</p> <p> pte'dirty, 6),</p> <p> pte'page_table_attribute_index, 7)</p>
---	--

Repeatedly Skolemizing and flattening,

Installing automatic rewrites from: every less_than_max_byte overwrite_bool_bit_less overwrite_shift_bits_less max_linear_offset bus_width

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of pte_page_to_byte_TCC1.

Q.E.D.

C.153.2 PTE_Model.pte_page_to_byte_TCC2

Terse proof for pte_page_to_byte_TCC2.

pte_page_to_byte_TCC2:

<p>{1} \forall (pte: Pte_4k_type):</p>	<p>offset(pte'page_base) \geq 0</p>
---	--

Repeatedly Skolemizing and flattening,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of pte_page_to_byte_TCC2.

Q.E.D.

C.153.3 PTE_Model.pte_page_to_byte_TCC3

Terse proof for pte_page_to_byte_TCC3.

pte_page_to_byte_TCC3:

```
{1}  ∃ (pte: Pte_4k_type):
      every[nat]
        ({s: nat | s < max_byte})
        ((:overwrite_shift_bits(overwrite_bool_bit(0, pte'global_page, 0),
                                         cut_bits(offset(pte'page_base), 12, 4), 4, 4),
          cut_bits(offset(pte'page_base), 16, 8),
          cut_bits(offset(pte'page_base), 24, 8):))
```

Repeatedly Skolemizing and flattening,

Installing automatic rewrites from: every less_than_max_byte cut_bits_byte overwrite_bool_bit_less overwrite_shift_bits_less

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of pte_page_to_byte_TCC3.

Q.E.D.

C.153.4 PTE_Model.pte_to_byte_TCC1

Terse proof for pte_to_byte_TCC1.

pte_to_byte_TCC1:

```
{1}  ∃ (pte: Pte_type): pte = Not_present ⊃ 0 < max_byte
```

Repeatedly Skolemizing and flattening,

Installing automatic rewrites from: every less_than_max_byte expt_2_8

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of pte_to_byte_TCC1.

Q.E.D.

C.153.5 PTE_Model.pte_to_byte_TCC2

Terse proof for pte_to_byte_TCC2.

pte_to_byte_TCC2:

```
{1}  ∃ (pte: Pte_type):
      pte = Not_present ⊃
      every[real]
        (λ (x: real): rational_pred(x) ∧ integer_pred(x) ∧ x ≥ 0 ∧ x < max_byte)
        ((:0, 0, 0:))
```

Repeatedly Skolemizing and flattening,

Installing automatic rewrites from: every

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Expanding the definition of max_byte,

Adding type constraints for expt(2, bits_per_byte),

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of pte_to_byte_TCC2.

Q.E.D.

C.153.6 PTE_Model.pte_page_from_byte_TCC1

Terse proof for `pte_page_from_byte_TCC1`.

`pte_page_from_byte_TCC1`:

$$\{1\} \quad \forall (\text{data}: \text{list}[\text{Byte}], a: \text{Address}): \text{pte_valid?}(\text{data}, a) \supset 0 < \text{length}[\text{Byte}](\text{data})$$

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `pte_page_from_byte_TCC1`.

Q.E.D.

C.153.7 PTE_Model.pte_page_from_byte_TCC2

Terse proof for `pte_page_from_byte_TCC2`.

`pte_page_from_byte_TCC2`:

$$\{1\} \quad \forall (\text{data}: \text{list}[\text{Byte}], a: \text{Address}):$$

$$\text{pte_valid?}(\text{data}, a) \wedge \text{cut_bit}(\text{nth}(\text{data}, 0), 0) \supset$$

$$((\text{singleton} [\text{Memory_access}](\text{Read}) \cup \{\text{Execute}\}) \subseteq \text{bools_to_memory_access}(\text{cut_bit}(\text{nth} [\text{Byte}](\text{data}, 0),$$

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `pte_page_from_byte_TCC2`.

Q.E.D.

C.153.8 PTE_Model.pte_page_from_byte_TCC3

Terse proof for `pte_page_from_byte_TCC3`.

`pte_page_from_byte_TCC3`:

$$\{1\} \quad \forall (\text{data}: \text{list}[\text{Byte}], a: \text{Address}):$$

$$\text{pte_valid?}(\text{data}, a) \wedge \text{cut_bit}(\text{nth}(\text{data}, 0), 0) \supset 1 < \text{length}[\text{Byte}](\text{data})$$

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `pte_page_from_byte_TCC3`.

Q.E.D.

C.153.9 PTE_Model.pte_page_from_byte_TCC4

Terse proof for `pte_page_from_byte_TCC4`.

`pte_page_from_byte_TCC4`:

$$\{1\} \quad \forall (\text{data}: \text{list}[\text{Byte}], a: \text{Address}):$$

$$\text{pte_valid?}(\text{data}, a) \wedge \text{cut_bit}(\text{nth}(\text{data}, 0), 0) \supset 2 < \text{length}[\text{Byte}](\text{data})$$

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `pte_page_from_byte_TCC4`.

Q.E.D.

C.153.10 PTE_Model.pte_page_from_byte_TCC5

Terse proof for `pte_page_from_byte_TCC5`.

`pte_page_from_byte_TCC5`:

$$\{1\} \quad \forall (\text{data}: \text{list}[\text{Byte}], a: \text{Address}):$$

$$\text{pte_valid?}(\text{data}, a) \wedge \text{cut_bit}(\text{nth}(\text{data}, 0), 0) \supset 3 < \text{length}[\text{Byte}](\text{data})$$

Repeatedly Skolemizing and flattening,

Installing automatic rewrites from: `every aligned_address?` `aligned_overwrite_shift_bits_more` `aligned_overwrite_shift_bits_less` `aligned_zero` `max_linear_expt_val` `overwrite_shift_bits_less` `pte_valid?` `max_linear_offset` `bus_width`

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `pte_page_from_byte_TCC5`.

Q.E.D.

C.153.11 PTE_Model.pte_page_from_byte_TCC6

Terse proof for `pte_page_from_byte_TCC6`.

pte_page_from_byte_TCC6:

```

{1}  ∀ (data: list[Byte], a: Address):
      pte_valid?(data, a) ∧ cut_bit(nth(data, 0), 0) ⊃
      Mem?(Mem(overwrite_shift_bits(overwrite_shift_bits(overwrite_shift_bits(0,
                                                                                               cut_bits
                                                                                               (nth[Byte]
                                                                                               (data, 1),
                                                                                               4,
                                                                                               4),
                                                                                               12,
                                                                                               4),
                                                                                               nth[Byte](data, 2), 16, 8),
                                                                                               nth[Byte](data, 3), 24, 8))'type_of)
      ∧
      (0 ≤
      Mem(overwrite_shift_bits(overwrite_shift_bits(overwrite_shift_bits(0,
                                                                                               cut_bits
                                                                                               (nth[Byte]
                                                                                               (data, 1),
                                                                                               4,
                                                                                               4),
                                                                                               12,
                                                                                               4),
                                                                                               nth[Byte](data, 2), 16, 8),
                                                                                               nth[Byte](data, 3), 24, 8))'offset
      ∧
      Mem(overwrite_shift_bits(overwrite_shift_bits(overwrite_shift_bits(0,
                                                                                               cut_bits
                                                                                               (nth[Byte]
                                                                                               (data, 1),
                                                                                               4,
                                                                                               4),
                                                                                               12,
                                                                                               4),
                                                                                               nth[Byte](data, 2), 16, 8),
                                                                                               nth[Byte](data, 3), 24, 8))'offset
      < max_linear_offset)
      ∧
      aligned?(12)
      (offset
      (Mem(overwrite_shift_bits(overwrite_shift_bits(overwrite_shift_bits
                                                                                               (0,
                                                                                               cut_bits
                                                                                               (nth[Byte](data, 1),
                                                                                               4,
                                                                                               4),
                                                                                               12,
                                                                                               4),
                                                                                               nth[Byte](data, 2),
                                                                                               16,
                                                                                               8),
                                                                                               nth[Byte](data, 3), 24, 8))))

```

Repeatedly Skolemizing and flattening,
 Installing automatic rewrites from: every aligned_address? aligned_overwrite_shift_bits_more
 aligned_overwrite_shift_bits_less aligned_zero max_linear_expt_val overwrite_shift_bits_less pte_valid?
 max_linear_offset bus_width
 Expanding the definition of Mem,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 This completes the proof of pte_page_from_byte_TCC6.
 Q.E.D.

C.153.12 PTE_Model.pte_from_byte_TCC1

Terse proof for pte_from_byte_TCC1.

pte_from_byte_TCC1:

$$\{1\} \quad \forall (data: \text{list}[\text{Byte}], a: \text{Address}):$$

$$\text{cut_bit}(\text{nth}(data, 0), 0) \wedge \text{pte_valid?}(data, a) \supset$$

$$\text{up?}[\text{Pte_4k_type}](\text{pte_page_from_byte}(data, a))$$

Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of pte_from_byte_TCC1.
 Q.E.D.

C.153.13 PTE_Model.pte_model_TCC1

Terse proof for pte_model_TCC1.

pte_model_TCC1:

$$\{1\} \quad \forall (f: [[\text{Pte_type}, \text{Address}] \rightarrow [\text{list}[\text{Byte}] \rightarrow \text{bool}]]):$$

$$(f = (\lambda (d: \text{Pte_type}, a: \text{Address}): (\text{zero_mask?}(4)))) \supset$$

$$(\forall (x_1: [\text{Pte_type}, \text{Address}]): \text{singleton?}[\text{list}[\text{Byte}]](f(x_1)))$$

Repeatedly Skolemizing and flattening,
 Replacing using formula -1,
 Using lemma singleton_zero_mask,
 Expanding the definition of singleton?,
 which is trivially true.
 This completes the proof of pte_model_TCC1.
 Q.E.D.

C.153.14 PTE_Model.pte_valid_to_byte

The L^AT_EX code for this proof is broken.

C.153.15 PTE_Model.pte_is_pod

Terse proof for pte_is_pod.

pte_is_pod:

$$\{1\} \quad \text{pod_data_type?}(\text{pte_model})$$

Expanding the definition of pod_data_type?,

Expanding the definition of interpreted_data_type?,
 Expanding the definition of uninterpreted_data_type?,
 Applying propositional simplification,
 we get 6 subgoals:

pte_is_pod.1:

$$\frac{}{\{1\} \quad \forall (l: \text{list}[\text{Byte}], a: \text{Address}): \quad \neg \text{length}(l) = \text{size}(\text{pte_model}'\text{uidt}) \supset \neg \text{valid?}(\text{pte_model}'\text{uidt})(l, a)}$$

Repeatedly Skolemizing and flattening,
 Expanding the definition of pte_model,
 Expanding the definition of pte_valid?,
 which is trivially true.
 This completes the proof of pte_is_pod.1.

pte_is_pod.2:

$$\frac{}{\{1\} \quad \forall (d: \text{Pte_type}, a: \text{Address}): \text{valid?}(\text{uidt}(\text{pte_model}))(\text{to_byte}(\text{pte_model})(d, a), a)}$$

Repeatedly Skolemizing and flattening,
 Rewriting using pte_valid_to_byte, matching in *,
 This completes the proof of pte_is_pod.2.

pte_is_pod.3:

$$\frac{}{\{1\} \quad \forall (l: \text{list}[\text{Byte}], a: \text{Address}): \quad \text{valid?}(\text{uidt}(\text{pte_model}))(l, a) \equiv \text{up?}(\text{from_byte}(\text{pte_model}))(l, a)}$$

Repeatedly Skolemizing and flattening,
 Expanding the definition of pte_model,
 Expanding the definition of pte_from_byte,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 This completes the proof of pte_is_pod.3.

pte_is_pod.4:

$$\frac{}{\{1\} \quad \forall (d: \text{Pte_type}, a: \text{Address}): \quad \text{down}(\text{from_byte}(\text{pte_model})(\text{to_byte}(\text{pte_model})(d, a), a)) = d}$$

Repeatedly Skolemizing and flattening,
 Expanding the definition of pte_model,
 Expanding the definition of pte_to_byte,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 we get 2 subgoals:

pte_is_pod.4.1:

$$\frac{\{-1\} \quad \text{not_present?}(d')}{\{1\} \quad \text{down}(\text{pte_from_byte}((:0, 0, 0, 0:), a')) = d'}$$

Installing automatic rewrites from: pte_from_byte cut_bit_zero nth pte_valid? length pte_page_from_byte

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 This completes the proof of pte_is_pod.4.1.

C Proof scripts

pte_is_pod.4.2:

{1}	not_present?(d')
{2}	down(pte_from_byte(pte_page_to_byte(pte(d'), a'), a')) = d'

Installing automatic rewrites from: pte_page_to_byte! pte_from_byte pte_page_from_byte pte_valid? nth length length_cdr length_is_cons cut_bits_overwrite_bool_bit_disjoined cut_bits_zero cut_bits_overwrite_bits_disjoined cut_bits_cut_bits cut_bits_overwrite_bits_contained cut_bit_overwrite_bool_bit overwrite_shift_bits_zero overwrite_shift_bits_merge aligned_address_aligned cut_bit_cut_bits cut_bit_overwrite_shift_bits cut_bits_overwrite_shift_bits_contained length_to_byte from_byte_to_byte valid_to_byte cut_bit_zero bool_to_memory_access_iso bool_to_memory_privilege_iso max_linear_offset bus_width

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Adding type constraints for pte(d!1)'page_base,

Rewriting using shift_bits_left_cut_bits, matching in *,

Applying extensionality,

we get 3 subgoals:

pte_is_pod.4.2.1:

{-1}	Mem?(pte(d)'page_base'type_of)
{-2}	$0 \leq \text{pte}(d)'page_base'offset$
{-3}	$\text{pte}(d)'page_base'offset < \text{expt}(2, 32)$
{-4}	aligned?(12)(offset(pte(d)'page_base))
{1}	(#page_base := Mem(offset(pte(d)'page_base)), global_page := pte(d)'global_page, page_table_attribute_index := pte(d)'page_table_attribute_index, dirty := pte(d)'dirty, accessed := pte(d)'accessed, cache_disabled := pte(d)'cache_disabled, write_through := pte(d)'write_through, privilege := pte(d)'privilege, access := bools_to_memory_access(memory_access_to_bools(pte(d)'access)'1, TRUE)#) = pte(d')
{2}	not_present?(d')

Applying extensionality,

we get 2 subgoals:

pte_is_pod.4.2.1.1:

{-1}	Mem?(pte(d)'page_base'type_of)
{-2}	$0 \leq \text{pte}(d)'page_base'offset$
{-3}	$\text{pte}(d)'page_base'offset < \text{expt}(2, 32)$
{-4}	aligned?(12)(offset(pte(d)'page_base))
{1}	bools_to_memory_access(memory_access_to_bools(pte(d)'access)'1, TRUE) = pte(d)'access
{2}	not_present?(d')

Applying extensionality,

Expanding the definition of bools_to_memory_access,

Expanding the definition of memory_access_to_bools,

Adding type constraints for pte(d!1)'access,

Expanding the definition of subset?,

Instantiating the top quantifier in -1 with the terms: x',

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of pte_is_pod.4.2.1.1.

pte_is_pod.4.2.1.2:

{-1}	Mem?(pte(d')'page_base'type_of)
{-2}	$0 \leq$ pte(d')'page_base'offset
{-3}	pte(d')'page_base'offset < expt(2, 32)
{-4}	aligned?(12)(offset(pte(d')'page_base))
{1}	Mem(offset(pte(d')'page_base)) = pte(d')'page_base
{2}	not_present?(d')

Expanding the definition of Mem,

Applying extensionality,

This completes the proof of pte_is_pod.4.2.1.2.

pte_is_pod.4.2.2:

{-1}	Mem?(pte(d')'page_base'type_of)
{-2}	$0 \leq$ pte(d')'page_base'offset
{-3}	pte(d')'page_base'offset < expt(2, 32)
{-4}	aligned?(12)(offset(pte(d')'page_base))
{1}	Mem?(Mem(offset(pte(d')'page_base))'type_of) \wedge $(0 \leq$ Mem(offset(pte(d')'page_base))'offset \wedge Mem(offset(pte(d')'page_base))'offset < expt(2, 32)) \wedge aligned?(12)(offset(Mem(offset(pte(d')'page_base))))
{2}	not_present?(d')

Expanding the definition of Mem,

which is trivially true.

This completes the proof of pte_is_pod.4.2.2.

pte_is_pod.4.2.3:

{-1}	Mem?(pte(d')'page_base'type_of)
{-2}	$0 \leq$ pte(d')'page_base'offset
{-3}	pte(d')'page_base'offset < expt(2, 32)
{-4}	aligned?(12)(offset(pte(d')'page_base))
{1}	$((\text{singleton}[\text{Memory_access}](\text{Read}) \cup \{\text{Execute}\}) \subseteq \text{bools_to_memory_access}(\text{memory_access_to_bools}(\text{pte}(\mathit{d}')\text{'access})))$
{2}	not_present?(d')

Expanding the definition of bools_to_memory_access,

Expanding the definition of memory_access_to_bools,

Adding type constraints for pte(d')'access,

Expanding the definition of subset?,

Repeatedly Skolemizing and flattening,

Instantiating the top quantifier in -1 with the terms: x' ,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of pte_is_pod.4.2.3.

pte_is_pod.5:

{1}	$\forall (l: \text{list}[\text{Byte}], a_1, a_2: \text{Address}):$ $\text{valid?}(\text{uidt}(\text{pte_model}))(l, a_1) \equiv \text{valid?}(\text{uidt}(\text{pte_model}))(l, a_2)$
-----	--

Repeatedly Skolemizing and flattening,

Hiding formulas: -1,

Expanding the definition of pte_model,

Expanding the definition of pte_valid?,

which is trivially true.

This completes the proof of pte_is_pod.5.

pte_is_pod.6:

{1} size(uidt(pte_model)) > 0

Expanding the definition of pte_model,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of pte_is_pod.6.
Q.E.D.

C.154 Proofs for Page_Directory_Types (paging-data.pvs)

C.154.1 Page_Directory_Types.Pde_pt_type_TCC1

Terse proof for Pde_pt_type_TCC1.

Pde_pt_type_TCC1:

{1} $\forall (a: \text{Memory_Address_4G}): \text{offset}(a) \geq 0$

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of Pde_pt_type_TCC1.
Q.E.D.

C.155 Proofs for Paging_Datatype (paging-data.pvs)

C.155.1 Paging_Datatype.paging_data_type_TCC1

Terse proof for paging_data_type_TCC1.

paging_data_type_TCC1:

{1} $\forall (\text{lvl}: \text{Level}): \text{lvl} = 0 \supset (\forall (x: ((\text{range_pt}(\text{lvl})))): (\text{pdir?})(x))$

Repeatedly Skolemizing and flattening,
Expanding the definition of range_pt,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of paging_data_type_TCC1.
Q.E.D.

C.155.2 Paging_Datatype.paging_data_type_TCC2

Terse proof for paging_data_type_TCC2.

paging_data_type_TCC2:

{1} $\forall (\text{lvl}: \text{Level}): \text{lvl} = 0 \supset (\forall (x_1: \text{Pde_type}): \text{range_pt}(\text{lvl})(\text{Pdir}(x_1)))$

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of paging_data_type_TCC2.
Q.E.D.

C.155.3 Paging_Datatype.paging_data_type_TCC3

Terse proof for paging_data_type_TCC3.

paging_data_type_TCC3:

<pre>{1} ∀ (lvl: Level): lvl = 0 ⊃ pod_data_type?[(range_pt(lvl))] (dt_lift[Pde_type, (range_pt(lvl))] (pde_data_type, restrict[(pdir?), (range_pt(lvl)), Pde_type](pdir), Pdir))</pre>

Repeatedly Skolemizing and flattening,

Expanding the definition of dt_lift,

Adding type constraints for pde_data_type,

Expanding the definition of pod_data_type?,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Expanding the definition of interpreted_data_type?,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 3 subgoals:

paging_data_type_TCC3.1:

<pre>{-1} uninterpreted_data_type?(pde_data_type'uidt) {-2} ∀ (d: Pde_type, a: Address): valid?(uidt(pde_data_type))(to_byte(pde_data_type)(d, a), a) {-3} ∀ (l: list[Byte], a: Address): valid?(uidt(pde_data_type))(l, a) ≡ up?(from_byte(pde_data_type)(l, a)) {-4} ∀ (d: Pde_type, a: Address): down(from_byte(pde_data_type)(to_byte(pde_data_type)(d, a), a)) = d {-5} ∀ (l: list[Byte], a₁, a₂: Address): valid?(uidt(pde_data_type))(l, a₁) ≡ valid?(uidt(pde_data_type))(l, a₂) {-6} size(uidt(pde_data_type)) > 0 {-7} lvl' < max_level {-8} lvl' = 0</pre>
<pre>{1} ∀ (d: ((range_pt(lvl')), a₁: Address): down(CASES from_byte(pde_data_type) (to_byte(pde_data_type) (restrict[(pdir?), ((range_pt(lvl'))), Pde_type](pdir)(d), a₁), a₁) OF up(d₁): up(Pdir(d₁)), bottom: bottom ENDCASES) = d</pre>

Repeatedly Skolemizing and flattening,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 2 subgoals:

paging_data_type_TCC3.1.1.1:

<pre> {-1} range_pt(lv')(d') {-2} uninterpreted_data_type?(pde_data_type'uidt) {-3} ∀ (d: Pde_type, a: Address): valid?(uidt(pde_data_type))(to_byte(pde_data_type)(d, a), a) {-4} ∀ (l: list[Byte], a: Address): valid?(uidt(pde_data_type))(l, a) ≡ up?(from_byte(pde_data_type)(l, a)) {-5} ∀ (d: Pde_type, a: Address): down(from_byte(pde_data_type)(to_byte(pde_data_type)(d, a), a)) = d {-6} ∀ (l: list[Byte], a₁, a₂: Address): valid?(uidt(pde_data_type))(l, a₁) ≡ valid?(uidt(pde_data_type))(l, a₂) {-7} size(uidt(pde_data_type)) > 0 {-8} lv' < max_level {-9} lv' = 0 {-10} up?(from_byte(pde_data_type) (to_byte(pde_data_type) (restrict[(pdir?), ((range_pt(lv'))), Pde_type](pdir)(d'), a'), a')) </pre>	<pre> {1} Pdir(down(from_byte(pde_data_type) (to_byte(pde_data_type) (restrict[(pdir?), ((range_pt(lv'))), Pde_type](pdir)(d'), a'), a'))) = d' </pre>
---	--

Instantiating the top quantifier in -5 with the terms: (pdir(d!1) a!1),
we get 2 subgoals:

paging_data_type_TCC3.1.1.1.1:

<pre> {-1} range_pt(lv')(d') {-2} uninterpreted_data_type?(pde_data_type'uidt) {-3} ∀ (d: Pde_type, a: Address): valid?(uidt(pde_data_type))(to_byte(pde_data_type)(d, a), a) {-4} ∀ (l: list[Byte], a: Address): valid?(uidt(pde_data_type))(l, a) ≡ up?(from_byte(pde_data_type)(l, a)) {-5} down(from_byte(pde_data_type)(to_byte(pde_data_type)(pdir(d'), a'), a')) = pdir(d') {-6} ∀ (l: list[Byte], a₁, a₂: Address): valid?(uidt(pde_data_type))(l, a₁) ≡ valid?(uidt(pde_data_type))(l, a₂) {-7} size(uidt(pde_data_type)) > 0 {-8} lv' < max_level {-9} lv' = 0 {-10} up?(from_byte(pde_data_type) (to_byte(pde_data_type) (restrict[(pdir?), ((range_pt(lv'))), Pde_type](pdir)(d'), a'), a')) </pre>	<pre> {1} Pdir(down(from_byte(pde_data_type) (to_byte(pde_data_type) (restrict[(pdir?), ((range_pt(lv'))), Pde_type](pdir)(d'), a'), a'))) = d' </pre>
--	--

Expanding the definition of restrict,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Replacing using formula -5,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Using lemma Paging_type_Pdir_eta,

This completes the proof of paging_data_type_TCC3.1.1.1.

paging_data_type_TCC3.1.1.2:

<pre> {-1} range_pt(lvl')(d') {-2} uninterpreted_data_type?(pde_data_type'uidt) {-3} ∀ (d: Pde_type, a: Address): valid?(uidt(pde_data_type))(to_byte(pde_data_type)(d, a), a) {-4} ∀ (l: list[Byte], a: Address): valid?(uidt(pde_data_type))(l, a) ≡ up?(from_byte(pde_data_type)(l, a)) {-5} ∀ (l: list[Byte], a1, a2: Address): valid?(uidt(pde_data_type))(l, a1) ≡ valid?(uidt(pde_data_type))(l, a2) {-6} size(uidt(pde_data_type)) > 0 {-7} lvl' < max_level {-8} lvl' = 0 {-9} up?(from_byte(pde_data_type) (to_byte(pde_data_type) (restrict[(pdir?), ((range_pt(lvl'))), Pde_type](pdir)(d'), a'), a')) </pre>	<hr style="border: 0.5px solid black;"/> <pre> {1} pdir'(d') {2} Pdir(down(from_byte(pde_data_type) (to_byte(pde_data_type) (restrict[(pdir?), ((range_pt(lvl'))), Pde_type](pdir)(d'), a'), a')))) = d' </pre>
---	---

Expanding the definition of range_pt,

which is trivially true.

This completes the proof of paging_data_type_TCC3.1.1.2.

paging_data_type_TCC3.1.2:

<pre> {-1} range_pt(lvl')(d') {-2} uninterpreted_data_type?(pde_data_type'uidt) {-3} ∀ (d: Pde_type, a: Address): valid?(uidt(pde_data_type))(to_byte(pde_data_type)(d, a), a) {-4} ∀ (l: list[Byte], a: Address): valid?(uidt(pde_data_type))(l, a) ≡ up?(from_byte(pde_data_type)(l, a)) {-5} ∀ (d: Pde_type, a: Address): down(from_byte(pde_data_type)(to_byte(pde_data_type)(d, a), a)) = d {-6} ∀ (l: list[Byte], a1, a2: Address): valid?(uidt(pde_data_type))(l, a1) ≡ valid?(uidt(pde_data_type))(l, a2) {-7} size(uidt(pde_data_type)) > 0 {-8} lvl' < max_level {-9} lvl' = 0 </pre>	<hr style="border: 0.5px solid black;"/> <pre> {1} up?(from_byte(pde_data_type) (to_byte(pde_data_type) (restrict[(pdir?), ((range_pt(lvl'))), Pde_type](pdir)(d'), a'), a')) {2} down(bottom) = d' </pre>
---	--

Expanding the definition of restrict,

Instantiating quantified variables,

C Proof scripts

Instantiating quantified variables,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `paging_data_type_TCC3.1.2`.

`paging_data_type_TCC3.2`:

{-1}	<code>uninterpreted_data_type?(pde_data_type'uidt)</code>
{-2}	$\forall (d: \text{Pde_type}, a: \text{Address}):$ <code>valid?(uidt(pde_data_type))(to_byte(pde_data_type)(d, a), a)</code>
{-3}	$\forall (l: \text{list[Byte]}, a: \text{Address}):$ <code>valid?(uidt(pde_data_type))(l, a) \equiv \text{up?}(from_byte(pde_data_type)(l, a))</code>
{-4}	$\forall (d: \text{Pde_type}, a: \text{Address}):$ <code>down(from_byte(pde_data_type)(to_byte(pde_data_type)(d, a), a)) = d</code>
{-5}	$\forall (l: \text{list[Byte]}, a_1, a_2: \text{Address}):$ <code>valid?(uidt(pde_data_type))(l, a_1) \equiv \text{valid?}(uidt(pde_data_type))(l, a_2)</code>
{-6}	<code>size(uidt(pde_data_type)) > 0</code>
{-7}	<code>lvl' < max_level</code>
{-8}	<code>lvl' = 0</code>
{1}	$\forall (L1: \text{list[Byte]}, a_1: \text{Address}):$ <code>valid?(uidt(pde_data_type))(L1, a_1) \equiv</code> <code>up?(CASES from_byte(pde_data_type)(L1, a_1) OF up(d₁): up(Pdir(d₁)), bot-</code> <code>tom: bottom</code> <code>ENDCASES)</code>

Repeatedly Skolemizing and flattening,

Instantiating quantified variables,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `paging_data_type_TCC3.2`.

`paging_data_type_TCC3.3`:

{-1}	<code>uninterpreted_data_type?(pde_data_type'uidt)</code>
{-2}	$\forall (d: \text{Pde_type}, a: \text{Address}):$ <code>valid?(uidt(pde_data_type))(to_byte(pde_data_type)(d, a), a)</code>
{-3}	$\forall (l: \text{list[Byte]}, a: \text{Address}):$ <code>valid?(uidt(pde_data_type))(l, a) \equiv \text{up?}(from_byte(pde_data_type)(l, a))</code>
{-4}	$\forall (d: \text{Pde_type}, a: \text{Address}):$ <code>down(from_byte(pde_data_type)(to_byte(pde_data_type)(d, a), a)) = d</code>
{-5}	$\forall (l: \text{list[Byte]}, a_1, a_2: \text{Address}):$ <code>valid?(uidt(pde_data_type))(l, a_1) \equiv \text{valid?}(uidt(pde_data_type))(l, a_2)</code>
{-6}	<code>size(uidt(pde_data_type)) > 0</code>
{-7}	<code>lvl' < max_level</code>
{-8}	<code>lvl' = 0</code>
{1}	$\forall (d: ((\text{range_pt}(lvl'))), a_1: \text{Address}):$ <code>valid?(uidt(pde_data_type))</code> <code>(to_byte(pde_data_type)</code> <code>(restrict[(pdir?), ((range_pt)(lvl'))], Pde_type](pdir)(d), a_1),</code> <code>a_1)</code>

Repeatedly Skolemizing and flattening,

Expanding the definition of `restrict`,

Instantiating quantified variables,

This completes the proof of `paging_data_type_TCC3.3`.

Q.E.D.

C.155.4 Paging_Datatype.paging_data_type_TCC4

Terse proof for paging_data_type_TCC4.

paging_data_type_TCC4:

$$\{1\} \quad \forall (lvl: \text{Level}): lvl = 1 \supset (\forall (x: ((\text{range_pt}(lvl)))): (\text{ptab?})(x))$$

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of paging_data_type_TCC4.

Q.E.D.

C.155.5 Paging_Datatype.paging_data_type_TCC5

Terse proof for paging_data_type_TCC5.

paging_data_type_TCC5:

$$\{1\} \quad \forall (lvl: \text{Level}): lvl = 1 \supset (\forall (x_1: \text{Pte_type}): \text{range_pt}(lvl)(\text{Ptab}(x_1)))$$

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of paging_data_type_TCC5.

Q.E.D.

C.155.6 Paging_Datatype.paging_data_type_TCC6

Terse proof for paging_data_type_TCC6.

paging_data_type_TCC6:

$$\{1\} \quad \forall (lvl: \text{Level}):$$

$$lvl = 1 \supset$$

$$\text{pod_data_type?}[(\text{range_pt}(lvl))]$$

$$(\text{dt_lift}[\text{Pte_type}, (\text{range_pt}(lvl))])$$

$$(\text{pte_data_type}, \text{restrict}[(\text{ptab?}), (\text{range_pt}(lvl)), \text{Pte_type}](\text{ptab}), \text{Ptab}))$$

Expanding the definition of restrict,

Expanding the definition of dt_lift,

Repeatedly Skolemizing and flattening,

Adding type constraints for pte_data_type,

Expanding the definition of pod_data_type?,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Expanding the definition of interpreted_data_type?,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 3 subgoals:

`paging_data_type_TCC6.1:`

{-1}	<code>uninterpreted_data_type?(pte_data_type'uidt)</code>
{-2}	<code>∀ (d: Pte_type, a: Address): valid?(uidt(pte_data_type))(to_byte(pte_data_type)(d, a), a)</code>
{-3}	<code>∀ (l: list[Byte], a: Address): valid?(uidt(pte_data_type))(l, a) ≡ up?(from_byte(pte_data_type)(l, a))</code>
{-4}	<code>∀ (d: Pte_type, a: Address): down(from_byte(pte_data_type)(to_byte(pte_data_type)(d, a), a)) = d</code>
{-5}	<code>∀ (l: list[Byte], a₁, a₂: Address): valid?(uidt(pte_data_type))(l, a₁) ≡ valid?(uidt(pte_data_type))(l, a₂)</code>
{-6}	<code>size(uidt(pte_data_type)) > 0</code>
{-7}	<code>lvl' < max_level</code>
{-8}	<code>lvl' = 1</code>
{1}	<code>∀ (d: ((range_pt(lvl'))), a₁: Address): down(CASES from_byte(pte_data_type)(to_byte(pte_data_type)(ptab(d), a₁), a₁) OF up(d₁): up(Ptab(d₁)), bottom: ENDCASES) = d</code>

Repeatedly Skolemizing and flattening,

Instantiating quantified variables,

Instantiating quantified variables,

Instantiating quantified variables,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Replacing using formula -5,

Using lemma `Paging_type_Ptab_eta`,

This completes the proof of `paging_data_type_TCC6.1`.

`paging_data_type_TCC6.2:`

{-1}	<code>uninterpreted_data_type?(pte_data_type'uidt)</code>
{-2}	<code>∀ (d: Pte_type, a: Address): valid?(uidt(pte_data_type))(to_byte(pte_data_type)(d, a), a)</code>
{-3}	<code>∀ (l: list[Byte], a: Address): valid?(uidt(pte_data_type))(l, a) ≡ up?(from_byte(pte_data_type)(l, a))</code>
{-4}	<code>∀ (d: Pte_type, a: Address): down(from_byte(pte_data_type)(to_byte(pte_data_type)(d, a), a)) = d</code>
{-5}	<code>∀ (l: list[Byte], a₁, a₂: Address): valid?(uidt(pte_data_type))(l, a₁) ≡ valid?(uidt(pte_data_type))(l, a₂)</code>
{-6}	<code>size(uidt(pte_data_type)) > 0</code>
{-7}	<code>lvl' < max_level</code>
{-8}	<code>lvl' = 1</code>
{1}	<code>∀ (l₁: list[Byte], a₁: Address): valid?(uidt(pte_data_type))(l₁, a₁) ≡ up?(CASES from_byte(pte_data_type)(l₁, a₁) OF up(d₁): up(Ptab(d₁)), bot- tom: bottom ENDCASES)</code>

Repeatedly Skolemizing and flattening,

Instantiating quantified variables,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `paging_data_type_TCC6.2`.

paging_data_type_TCC6.3:

{-1}	uninterpreted_data_type?(pte_data_type uidt)
{-2}	$\forall (d: \text{Pte_type}, a: \text{Address}):$ valid?(uidt(pte_data_type))(to_byte(pte_data_type)(d, a), a)
{-3}	$\forall (l: \text{list}[\text{Byte}], a: \text{Address}):$ valid?(uidt(pte_data_type))(l, a) \equiv up?(from_byte(pte_data_type)(l, a))
{-4}	$\forall (d: \text{Pte_type}, a: \text{Address}):$ down(from_byte(pte_data_type)(to_byte(pte_data_type)(d, a), a)) = d
{-5}	$\forall (l: \text{list}[\text{Byte}], a_1, a_2: \text{Address}):$ valid?(uidt(pte_data_type))(l, a_1) \equiv valid?(uidt(pte_data_type))(l, a_2)
{-6}	size(uidt(pte_data_type)) > 0
{-7}	lvl' < max_level
{-8}	lvl' = 1
{1}	$\forall (d: (\text{range_pt}(\text{lvl}')), a_1: \text{Address}):$ valid?(uidt(pte_data_type))(to_byte(pte_data_type)(ptab(d), a_1), a_1)

Repeatedly Skolemizing and flattening,
 Instantiating quantified variables,
 This completes the proof of paging_data_type_TCC6.3.
 Q.E.D.

C.155.7 Paging_Datatype.paging_data_type_TCC7

Terse proof for paging_data_type_TCC7.

paging_data_type_TCC7:

{1}	$\forall (\text{lvl}: \text{Level}): \text{lvl} = 0 \vee \text{lvl} = 1$
-----	--

Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of paging_data_type_TCC7.
 Q.E.D.

C.156 Proofs for Paging_type (paging-data.pvs)

This theory contains no provable formal statements.

C.157 Proofs for Paging_type_adt (Paging_type_adt.pvs)

This theory contains no provable formal statements.

C.158 Proofs for Paging_type_adt_reduce (Paging_type_adt.pvs)

This theory contains no provable formal statements.

C.159 Proofs for Paging_type_helpers (paging-data.pvs)

C.159.1 Paging_type_helpers.range_pt_TCC1

Terse proof for range_pt_TCC1.

range_pt_TCC1:

$$\{1\} \quad \forall (lvl: Level): lvl = 0 \vee lvl = 1$$

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of range_pt_TCC1.

Q.E.D.

C.159.2 Paging_type_helpers.base_TCC1

Terse proof for base_TCC1.

base_TCC1:

$$\{1\} \quad \forall (pt: (present?), ptab: Pte_type): pt = Ptab(ptab) \supset pte?(ptab)$$

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of base_TCC1.

Q.E.D.

C.159.3 Paging_type_helpers.base_TCC2

Terse proof for base_TCC2.

base_TCC2:

$$\{1\} \quad \forall (pt: (present?), pdir: Pdir_type): pt = Pdir(pdir) \supset \neg \text{not_present?}(pdir)$$

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of base_TCC2.

Q.E.D.

C.159.4 Paging_type_helpers.set_reference_range

Terse proof for set_reference_range.

set_reference_range:

$$\{1\} \quad \forall (lvl: Level, pt: (range_pt(lvl)), access: Memory_access): \\ \text{range_pt}(lvl)(\text{set_reference}(pt, access))$$

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of set_reference_range.

Q.E.D.

C.159.5 Paging_type_helpers.set_reference_pdir_entry

Terse proof for set_reference_pdir_entry.

`set_reference_pdir_entry:`

$\{1\} \quad \forall (pt1, pt2: \text{Paging_type}, \text{access}: \text{Memory_access}):$ $\text{set_reference}(pt1, \text{access}) = \text{set_reference}(pt2, \text{access}) \supset$ $\text{pdir_entry?}(pt1) = \text{pdir_entry?}(pt2)$
--

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `set_reference_pdir_entry`.
Q.E.D.

C.159.6 *Paging_type_helpers.set_reference_ptab_entry*

Terse proof for `set_reference_ptab_entry`.

`set_reference_ptab_entry:`

$\{1\} \quad \forall (pt1, pt2: \text{Paging_type}, \text{access}: \text{Memory_access}):$ $\text{set_reference}(pt1, \text{access}) = \text{set_reference}(pt2, \text{access}) \supset$ $\text{ptab_entry?}(pt1) = \text{ptab_entry?}(pt2)$
--

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `set_reference_ptab_entry`.
Q.E.D.

C.159.7 *Paging_type_helpers.set_reference_paging_type*

Terse proof for `set_reference_paging_type`.

`set_reference_paging_type:`

$\{1\} \quad \forall (pt1, pt2: \text{Paging_type}, \text{access}: \text{Memory_access}, \text{lvl}: \text{Level}):$ $\text{set_reference}(pt1, \text{access}) = \text{set_reference}(pt2, \text{access}) \supset$ $\text{paging_type?}(\text{lvl}, pt1) = \text{paging_type?}(\text{lvl}, pt2)$
--

Repeatedly Skolemizing and flattening,
Expanding the definition of `paging_type?`,
Case splitting on `lvl = 0`,
we get 2 subgoals:

`set_reference_paging_type.1:`

$\{-1\} \quad \text{lvl}' = 0$ $\{-2\} \quad \text{lvl}' < \text{max_level}$ $\{-3\} \quad \text{set_reference}(pt1', \text{access}') = \text{set_reference}(pt2', \text{access}')$
$\{1\} \quad \text{COND } \text{lvl}' = 0 \rightarrow \text{pdir_entry?}(pt1'), \text{ ELSE } \rightarrow \text{ptab_entry?}(pt1') \text{ ENDCOND} =$ $\text{COND } \text{lvl}' = 0 \rightarrow \text{pdir_entry?}(pt2'), \text{ ELSE } \rightarrow \text{ptab_entry?}(pt2') \text{ ENDCOND}$

Using lemma `set_reference_pdir_entry`,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `set_reference_paging_type.1`.

`set_reference_paging_type.2:`

$\{-1\} \quad \text{lvl}' < \text{max_level}$ $\{-2\} \quad \text{set_reference}(pt1', \text{access}') = \text{set_reference}(pt2', \text{access}')$
$\{1\} \quad \text{lvl}' = 0$ $\{2\} \quad \text{COND } \text{lvl}' = 0 \rightarrow \text{pdir_entry?}(pt1'), \text{ ELSE } \rightarrow \text{ptab_entry?}(pt1') \text{ ENDCOND} =$ $\text{COND } \text{lvl}' = 0 \rightarrow \text{pdir_entry?}(pt2'), \text{ ELSE } \rightarrow \text{ptab_entry?}(pt2') \text{ ENDCOND}$

C Proof scripts

Using lemma `set_reference_ptab_entry`,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `set_reference_paging_type.2`.
Q.E.D.

C.159.8 Paging_type_helpers.set_reference_present

Terse proof for `set_reference_present`.

`set_reference_present`:

{1} $\forall (pt1, pt2: \text{Paging_type}, \text{access}: \text{Memory_access}):$ $\text{set_reference}(pt1, \text{access}) = \text{set_reference}(pt2, \text{access}) \supset \text{present?}(pt1) = \text{present?}(pt2)$

Repeatedly Skolemizing and flattening,
Trying repeated skolemization, instantiation, and if-lifting,
we get 12 subgoals:

`set_reference_present.1`:

{-1} <code>Write?(access')</code> {-2} <code>ptab?(pt1')</code> {-3} <code>Ptab(Pte(pte(ptab(pt1'))) WITH [dirty := TRUE, accessed := TRUE]) = Ptab(Not_present)</code> {-4} <code>ptab?(pt2')</code> {-5} <code>not_present?(ptab(pt2'))</code>
{1} <code>not_present?(ptab(pt1'))</code>

Applying `decompose-equality`,
This completes the proof of `set_reference_present.1`.

`set_reference_present.2`:

{-1} <code>Write?(access')</code> {-2} <code>pde_pt?(pdir(pt1'))</code> {-3} <code>Pdir(Pde_pt(pde_pt(pdir(pt1'))) WITH [accessed := TRUE]) = Pdir(Not_present)</code> {-4} <code>not_present?(pdir(pt2'))</code>
{1} <code>ptab?(pt1')</code> {2} <code>ptab?(pt2')</code>

Applying `decompose-equality`,
This completes the proof of `set_reference_present.2`.

`set_reference_present.3`:

{-1} <code>Write?(access')</code> {-2} <code>Pdir(Pde_4m(pde_4m(pdir(pt1'))) WITH [dirty := TRUE, accessed := TRUE]) = Pdir(Not_present)</code> {-3} <code>not_present?(pdir(pt2'))</code>
{1} <code>ptab?(pt1')</code> {2} <code>not_present?(pdir(pt1'))</code> {3} <code>pde_pt?(pdir(pt1'))</code> {4} <code>ptab?(pt2')</code>

Applying `decompose-equality`,
This completes the proof of `set_reference_present.3`.

set_reference_present.4:

{-1}	ptab?(pt1')
{-2}	Ptab(Pte(pte(ptab(pt1'))) WITH [accessed := TRUE])) = Ptab(Not_present)
{-3}	ptab?(pt2')
{-4}	not_present?(ptab(pt2'))
{1}	Write?(access')
{2}	not_present?(ptab(pt1'))

Applying decompose-equality,
This completes the proof of **set_reference_present.4**.

set_reference_present.5:

{-1}	pde_pt?(pdir(pt1'))
{-2}	Pdir(Pde_pt(pde_pt(pdir(pt1'))) WITH [accessed := TRUE])) = Pdir(Not_present)
{-3}	not_present?(pdir(pt2'))
{1}	Write?(access')
{2}	ptab?(pt1')
{3}	ptab?(pt2')

Applying decompose-equality,
This completes the proof of **set_reference_present.5**.

set_reference_present.6:

{-1}	Pdir(Pde_4m(pde_4m(pdir(pt1'))) WITH [accessed := TRUE])) = Pdir(Not_present)
{-2}	not_present?(pdir(pt2'))
{1}	Write?(access')
{2}	ptab?(pt1')
{3}	not_present?(pdir(pt1'))
{4}	pde_pt?(pdir(pt1'))
{5}	ptab?(pt2')

Applying decompose-equality,
This completes the proof of **set_reference_present.6**.

set_reference_present.7:

{-1}	Write?(access')
{-2}	ptab?(pt1')
{-3}	not_present?(ptab(pt1'))
{-4}	Ptab(Not_present) = Ptab(Pte(pte(ptab(pt2'))) WITH [dirty := TRUE, accessed := TRUE]))
{-5}	ptab?(pt2')
{1}	not_present?(ptab(pt2'))

Applying decompose-equality,
This completes the proof of **set_reference_present.7**.

set_reference_present.8:

{-1}	Write?(access')
{-2}	not_present?(pdir(pt1'))
{-3}	pde_pt?(pdir(pt2'))
{-4}	Pdir(Not_present) = Pdir(Pde_pt(pde_pt(pdir(pt2'))) WITH [accessed := TRUE]))
{1}	ptab?(pt1')
{2}	ptab?(pt2')

Applying decompose-equality,
This completes the proof of **set_reference_present.8**.

`set_reference_present.9:`

{-1}	Write?(access')
{-2}	not_present?(pdir(pt1'))
{-3}	Pdir(Not_present) = Pdir(Pde_4m(pde_4m(pdir(pt2'))) WITH [dirty := TRUE, accessed := TRUE]))
{1}	ptab?(pt1')
{2}	pde_pt?(pdir(pt2'))
{3}	ptab?(pt2')
{4}	not_present?(pdir(pt2'))

Applying decompose-equality,

This completes the proof of `set_reference_present.9`.

`set_reference_present.10:`

{-1}	ptab?(pt1')
{-2}	not_present?(ptab(pt1'))
{-3}	Ptab(Not_present) = Ptab(Pte(pte(ptab(pt2'))) WITH [accessed := TRUE]))
{-4}	ptab?(pt2')
{1}	Write?(access')
{2}	not_present?(ptab(pt2'))

Applying decompose-equality,

This completes the proof of `set_reference_present.10`.

`set_reference_present.11:`

{-1}	not_present?(pdir(pt1'))
{-2}	pde_pt?(pdir(pt2'))
{-3}	Pdir(Not_present) = Pdir(Pde_pt(pde_pt(pdir(pt2'))) WITH [accessed := TRUE]))
{1}	Write?(access')
{2}	ptab?(pt1')
{3}	ptab?(pt2')

Applying decompose-equality,

This completes the proof of `set_reference_present.11`.

`set_reference_present.12:`

{-1}	not_present?(pdir(pt1'))
{-2}	Pdir(Not_present) = Pdir(Pde_4m(pde_4m(pdir(pt2'))) WITH [accessed := TRUE]))
{1}	Write?(access')
{2}	ptab?(pt1')
{3}	pde_pt?(pdir(pt2'))
{4}	ptab?(pt2')
{5}	not_present?(pdir(pt2'))

Applying decompose-equality,

This completes the proof of `set_reference_present.12`.

Q.E.D.

C.159.9 Paging_type_helpers.set_reference_base_TCC1

Terse proof for `set_reference_base_TCC1`.

set_reference_base_TCC1:

$$\frac{}{\{1\} \quad \forall (pt1, pt2: \text{Paging_type}, \text{access}: \text{Memory_access}): \quad \text{set_reference}(pt1, \text{access}) = \text{set_reference}(pt2, \text{access}) \wedge \text{present?}(pt1) \supset \text{present?}(pt2)}$$

Repeatedly Skolemizing and flattening,

Using lemma set_reference_present,

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of set_reference_base_TCC1.

Q.E.D.

C.159.10 Paging_type_helpers.set_reference_base

Terse proof for set_reference_base.

set_reference_base:

$$\frac{}{\{1\} \quad \forall (pt1, pt2: \text{Paging_type}, \text{access}: \text{Memory_access}): \quad \text{set_reference}(pt1, \text{access}) = \text{set_reference}(pt2, \text{access}) \wedge \text{present?}(pt1) \supset \quad \text{base}(pt1) = \text{base}(pt2)}$$

Trying repeated skolemization, instantiation, and if-lifting,

we get 16 subgoals:

set_reference_base.1:

$$\frac{\begin{array}{l} \{-1\} \quad \text{Write?}(\text{access}') \\ \{-2\} \quad \text{ptab?}(pt1') \\ \{-3\} \quad \text{ptab?}(pt2') \\ \{-4\} \quad \text{not_present?}(\text{ptab}(pt2')) \\ \{-5\} \quad \text{Ptab}(\text{Pte}(\text{pte}(\text{ptab}(pt1')) \text{ WITH } [\text{dirty} := \text{TRUE}, \text{accessed} := \text{TRUE}]))) = \text{Ptab}(\text{Not_present}) \end{array}}{\begin{array}{l} \{1\} \quad \text{not_present?}(\text{ptab}(pt1')) \\ \{2\} \quad \text{page_base}(\text{pte}(\text{ptab}(pt1'))) = \text{page_base}(\text{pte}(\text{ptab}(pt2'))) \end{array}}$$

Applying decompose-equality,

This completes the proof of set_reference_base.1.

set_reference_base.2:

$$\frac{\begin{array}{l} \{-1\} \quad \text{Write?}(\text{access}') \\ \{-2\} \quad \text{ptab?}(pt1') \\ \{-3\} \quad \text{ptab?}(pt2') \\ \{-4\} \quad \text{Ptab}(\text{Pte}(\text{pte}(\text{ptab}(pt1')) \text{ WITH } [\text{dirty} := \text{TRUE}, \text{accessed} := \text{TRUE}]))) = \quad \text{Ptab}(\text{Pte}(\text{pte}(\text{ptab}(pt2')) \text{ WITH } [\text{dirty} := \text{TRUE}, \text{accessed} := \text{TRUE}]))) \end{array}}{\begin{array}{l} \{1\} \quad \text{not_present?}(\text{ptab}(pt1')) \\ \{2\} \quad \text{not_present?}(\text{ptab}(pt2')) \\ \{3\} \quad \text{page_base}(\text{pte}(\text{ptab}(pt1'))) = \text{page_base}(\text{pte}(\text{ptab}(pt2'))) \end{array}}$$

Applying decompose-equality,

Applying decompose-equality,

Applying decompose-equality,

Applying decompose-equality,

This completes the proof of set_reference_base.2.

set_reference_base.3:

{-1}	Write?(access')
{-2}	pde_pt?(pdir(pt1'))
{-3}	not_present?(pdir(pt2'))
{-4}	Pdir(Pde_pt(pde_pt(pdir(pt1'))) WITH [accessed := TRUE])) = Pdir(Not_present)
{1}	ptab?(pt1')
{2}	ptab?(pt2')
{3}	page_table_base(pde_pt(pdir(pt1'))) = page_base(pde_4m(pdir(pt2')))

Applying decompose-equality,

This completes the proof of **set_reference_base.3**.

set_reference_base.4:

{-1}	Write?(access')
{-2}	pde_pt?(pdir(pt1'))
{-3}	pde_pt?(pdir(pt2'))
{-4}	Pdir(Pde_pt(pde_pt(pdir(pt1'))) WITH [accessed := TRUE])) = Pdir(Pde_pt(pde_pt(pdir(pt2'))) WITH [accessed := TRUE]))
{1}	ptab?(pt1')
{2}	ptab?(pt2')
{3}	page_table_base(pde_pt(pdir(pt1'))) = page_table_base(pde_pt(pdir(pt2')))

Applying decompose-equality,

Applying decompose-equality,

Applying decompose-equality,

This completes the proof of **set_reference_base.4**.

set_reference_base.5:

{-1}	Write?(access')
{-2}	pde_pt?(pdir(pt1'))
{-3}	Pdir(Pde_pt(pde_pt(pdir(pt1'))) WITH [accessed := TRUE])) = Pdir(Pde_4m(pde_4m(pdir(pt2'))) WITH [dirty := TRUE, accessed := TRUE]))
{1}	ptab?(pt1')
{2}	ptab?(pt2')
{3}	not_present?(pdir(pt2'))
{4}	pde_pt?(pdir(pt2'))
{5}	page_table_base(pde_pt(pdir(pt1'))) = page_base(pde_4m(pdir(pt2')))

Applying decompose-equality,

This completes the proof of **set_reference_base.5**.

set_reference_base.6:

{-1}	Write?(access')
{-2}	not_present?(pdir(pt2'))
{-3}	Pdir(Pde_4m(pde_4m(pdir(pt1'))) WITH [dirty := TRUE, accessed := TRUE])) = Pdir(Not_present)
{1}	ptab?(pt1')
{2}	not_present?(pdir(pt1'))
{3}	pde_pt?(pdir(pt1'))
{4}	ptab?(pt2')
{5}	page_base(pde_4m(pdir(pt1'))) = page_base(pde_4m(pdir(pt2')))

Applying decompose-equality,

This completes the proof of **set_reference_base.6**.

set_reference_base.7:

<pre>{-1} Write?(access') {-2} pde_pt?(pdir(pt2')) {-3} Pdir(Pde_4m(pde_4m(pdir(pt1'))) WITH [dirty := TRUE, accessed := TRUE])) = Pdir(Pde_pt(pde_pt(pdir(pt2'))) WITH [accessed := TRUE]))</pre>	<hr style="border: 0.5px solid black;"/> <pre>{1} ptab?(pt1') {2} not_present?(pdir(pt1')) {3} pde_pt?(pdir(pt1')) {4} ptab?(pt2') {5} page_base(pde_4m(pdir(pt1'))) = page_table_base(pde_pt(pdir(pt2')))</pre>
--	--

Applying decompose-equality,

This completes the proof of **set_reference_base.7**.

set_reference_base.8:

<pre>{-1} Write?(access') {-2} Pdir(Pde_4m(pde_4m(pdir(pt1'))) WITH [dirty := TRUE, accessed := TRUE])) = Pdir(Pde_4m(pde_4m(pdir(pt2'))) WITH [dirty := TRUE, accessed := TRUE]))</pre>	<hr style="border: 0.5px solid black;"/> <pre>{1} ptab?(pt1') {2} not_present?(pdir(pt1')) {3} pde_pt?(pdir(pt1')) {4} ptab?(pt2') {5} not_present?(pdir(pt2')) {6} pde_pt?(pdir(pt2')) {7} page_base(pde_4m(pdir(pt1'))) = page_base(pde_4m(pdir(pt2')))</pre>
--	---

Applying decompose-equality,

Applying decompose-equality,

Applying decompose-equality,

This completes the proof of **set_reference_base.8**.

set_reference_base.9:

<pre>{-1} ptab?(pt1') {-2} ptab?(pt2') {-3} not_present?(ptab(pt2')) {-4} Ptab(Pte(pte(ptab(pt1'))) WITH [accessed := TRUE])) = Ptab(Not_present)</pre>	<hr style="border: 0.5px solid black;"/> <pre>{1} Write?(access') {2} not_present?(ptab(pt1')) {3} page_base(pte(ptab(pt1'))) = page_base(pte(ptab(pt2')))</pre>
---	--

Applying decompose-equality,

This completes the proof of **set_reference_base.9**.

set_reference_base.10:

<pre>{-1} ptab?(pt1') {-2} ptab?(pt2') {-3} Ptab(Pte(pte(ptab(pt1'))) WITH [accessed := TRUE])) = Ptab(Pte(pte(ptab(pt2'))) WITH [accessed := TRUE]))</pre>	<hr style="border: 0.5px solid black;"/> <pre>{1} Write?(access') {2} not_present?(ptab(pt1')) {3} not_present?(ptab(pt2')) {4} page_base(pte(ptab(pt1'))) = page_base(pte(ptab(pt2')))</pre>
---	---

Applying decompose-equality,

Applying decompose-equality,

C Proof scripts

Applying decompose-equality,

This completes the proof of `set_reference_base.10`.

`set_reference_base.11`:

{-1}	<code>pde_pt?(pdir(pt1'))</code>
{-2}	<code>not_present?(pdir(pt2'))</code>
{-3}	<code>Pdir(Pde_pt(pde_pt(pdir(pt1'))) WITH [accessed := TRUE]) = Pdir(Not_present)</code>
{1}	<code>Write?(access')</code>
{2}	<code>ptab?(pt1')</code>
{3}	<code>ptab?(pt2')</code>
{4}	<code>page_table_base(pde_pt(pdir(pt1'))) = page_base(pde_4m(pdir(pt2')))</code>

Applying decompose-equality,

This completes the proof of `set_reference_base.11`.

`set_reference_base.12`:

{-1}	<code>pde_pt?(pdir(pt1'))</code>
{-2}	<code>pde_pt?(pdir(pt2'))</code>
{-3}	<code>Pdir(Pde_pt(pde_pt(pdir(pt1'))) WITH [accessed := TRUE]) = Pdir(Pde_pt(pde_pt(pdir(pt2'))) WITH [accessed := TRUE])</code>
{1}	<code>Write?(access')</code>
{2}	<code>ptab?(pt1')</code>
{3}	<code>ptab?(pt2')</code>
{4}	<code>page_table_base(pde_pt(pdir(pt1'))) = page_table_base(pde_pt(pdir(pt2')))</code>

Applying decompose-equality,

Applying decompose-equality,

Applying decompose-equality,

This completes the proof of `set_reference_base.12`.

`set_reference_base.13`:

{-1}	<code>pde_pt?(pdir(pt1'))</code>
{-2}	<code>Pdir(Pde_pt(pde_pt(pdir(pt1'))) WITH [accessed := TRUE]) = Pdir(Pde_4m(pde_4m(pdir(pt2'))) WITH [accessed := TRUE])</code>
{1}	<code>Write?(access')</code>
{2}	<code>ptab?(pt1')</code>
{3}	<code>ptab?(pt2')</code>
{4}	<code>not_present?(pdir(pt2'))</code>
{5}	<code>pde_pt?(pdir(pt2'))</code>
{6}	<code>page_table_base(pde_pt(pdir(pt1'))) = page_base(pde_4m(pdir(pt2')))</code>

Applying decompose-equality,

This completes the proof of `set_reference_base.13`.

`set_reference_base.14`:

{-1}	<code>not_present?(pdir(pt2'))</code>
{-2}	<code>Pdir(Pde_4m(pde_4m(pdir(pt1'))) WITH [accessed := TRUE]) = Pdir(Not_present)</code>
{1}	<code>Write?(access')</code>
{2}	<code>ptab?(pt1')</code>
{3}	<code>not_present?(pdir(pt1'))</code>
{4}	<code>pde_pt?(pdir(pt1'))</code>
{5}	<code>ptab?(pt2')</code>
{6}	<code>page_base(pde_4m(pdir(pt1'))) = page_base(pde_4m(pdir(pt2')))</code>

Applying decompose-equality,

This completes the proof of `set_reference_base.14`.

set_reference_base.15:

{-1}	pde_pt?(pdir(pt2'))
{-2}	Pdir(Pde_4m(pde_4m(pdir(pt1'))) WITH [accessed := TRUE])) = Pdir(Pde_pt(pde_pt(pdir(pt2'))) WITH [accessed := TRUE]))
{1}	Write?(access')
{2}	ptab?(pt1')
{3}	not_present?(pdir(pt1'))
{4}	pde_pt?(pdir(pt1'))
{5}	ptab?(pt2')
{6}	page_base(pde_4m(pdir(pt1'))) = page_table_base(pde_pt(pdir(pt2')))

Applying decompose-equality,

This completes the proof of **set_reference_base.15**.

set_reference_base.16:

{-1}	Pdir(Pde_4m(pde_4m(pdir(pt1'))) WITH [accessed := TRUE])) = Pdir(Pde_4m(pde_4m(pdir(pt2'))) WITH [accessed := TRUE]))
{1}	Write?(access')
{2}	ptab?(pt1')
{3}	not_present?(pdir(pt1'))
{4}	pde_pt?(pdir(pt1'))
{5}	ptab?(pt2')
{6}	not_present?(pdir(pt2'))
{7}	pde_pt?(pdir(pt2'))
{8}	page_base(pde_4m(pdir(pt1'))) = page_base(pde_4m(pdir(pt2')))

Applying decompose-equality,

Applying decompose-equality,

Applying decompose-equality,

This completes the proof of **set_reference_base.16**.

Q.E.D.

C.159.11 *Paging_type_helpers.set_reference_accessible*

Terse proof for **set_reference_accessible**.

set_reference_accessible:

{1}	\forall (pt1, pt2: <i>Paging_type</i> , access, ac1: <i>Memory_access</i>): set_reference(pt1, access) = set_reference(pt2, access) \wedge present?(pt1) \supset accessible?(pt1, ac1) = accessible?(pt2, ac1)
-----	---

Trying repeated skolemization, instantiation, and if-lifting,

we get 32 subgoals:

set_reference_accessible.1:

{-1}	Write?(access')
{-2}	ptab?(pt1')
{-3}	ptab?(pt2')
{-4}	not_present?(ptab(pt2'))
{-5}	Ptab(Pte(pte(ptab(pt1'))) WITH [dirty := TRUE, accessed := TRUE])) = Ptab(Not_present)
{-6}	access(pte(ptab(pt1')))(ac1')
{1}	not_present?(ptab(pt1'))
{2}	access(pte(ptab(pt2')))(ac1')

C Proof scripts

Applying decompose-equality,

This completes the proof of `set_reference_accessible.1`.

`set_reference_accessible.2`:

{-1}	Write?(access')
{-2}	ptab?(pt1')
{-3}	ptab?(pt2')
{-4}	Ptab(Pte(pte(ptab(pt1'))) WITH [dirty := TRUE, accessed := TRUE])) = Ptab(Pte(pte(ptab(pt2'))) WITH [dirty := TRUE, accessed := TRUE]))
{-5}	access(pte(ptab(pt1')))(ac1')
{1}	not_present?(ptab(pt1'))
{2}	not_present?(ptab(pt2'))
{3}	access(pte(ptab(pt2')))(ac1')

Applying decompose-equality,

Applying decompose-equality,

Applying decompose-equality,

This completes the proof of `set_reference_accessible.2`.

`set_reference_accessible.3`:

{-1}	Write?(access')
{-2}	ptab?(pt1')
{-3}	ptab?(pt2')
{-4}	not_present?(ptab(pt2'))
{-5}	Ptab(Pte(pte(ptab(pt1'))) WITH [dirty := TRUE, accessed := TRUE])) = Ptab(Not_present)
{-6}	access(pte(ptab(pt2')))(ac1')
{1}	not_present?(ptab(pt1'))
{2}	access(pte(ptab(pt1')))(ac1')

Applying decompose-equality,

This completes the proof of `set_reference_accessible.3`.

`set_reference_accessible.4`:

{-1}	Write?(access')
{-2}	ptab?(pt1')
{-3}	ptab?(pt2')
{-4}	Ptab(Pte(pte(ptab(pt1'))) WITH [dirty := TRUE, accessed := TRUE])) = Ptab(Pte(pte(ptab(pt2'))) WITH [dirty := TRUE, accessed := TRUE]))
{-5}	access(pte(ptab(pt2')))(ac1')
{1}	not_present?(ptab(pt1'))
{2}	not_present?(ptab(pt2'))
{3}	access(pte(ptab(pt1')))(ac1')

Applying decompose-equality,

Applying decompose-equality,

Applying decompose-equality,

This completes the proof of `set_reference_accessible.4`.

set_reference_accessible.5:

{-1}	Write?(access')
{-2}	pde_pt?(pdir(pt1'))
{-3}	not_present?(pdir(pt2'))
{-4}	Pdir(Pde_pt(pde_pt(pdir(pt1'))) WITH [accessed := TRUE])) = Pdir(Not_present)
{-5}	access(pde_4m(pdir(pt2')))(ac1')
{1}	ptab?(pt1')
{2}	ptab?(pt2')
{3}	access(pde_pt(pdir(pt1')))(ac1')

Applying decompose-equality,

This completes the proof of set_reference_accessible.5.

set_reference_accessible.6:

{-1}	Write?(access')
{-2}	pde_pt?(pdir(pt1'))
{-3}	pde_pt?(pdir(pt2'))
{-4}	Pdir(Pde_pt(pde_pt(pdir(pt1'))) WITH [accessed := TRUE])) = Pdir(Pde_pt(pde_pt(pdir(pt2'))) WITH [accessed := TRUE]))
{-5}	access(pde_pt(pdir(pt2')))(ac1')
{1}	ptab?(pt1')
{2}	ptab?(pt2')
{3}	access(pde_pt(pdir(pt1')))(ac1')

Applying decompose-equality,

Applying decompose-equality,

Applying decompose-equality,

This completes the proof of set_reference_accessible.6.

set_reference_accessible.7:

{-1}	Write?(access')
{-2}	pde_pt?(pdir(pt1'))
{-3}	Pdir(Pde_pt(pde_pt(pdir(pt1'))) WITH [accessed := TRUE])) = Pdir(Pde_4m(pde_4m(pdir(pt2'))) WITH [dirty := TRUE, accessed := TRUE]))
{-4}	access(pde_4m(pdir(pt2')))(ac1')
{1}	ptab?(pt1')
{2}	ptab?(pt2')
{3}	not_present?(pdir(pt2'))
{4}	pde_pt?(pdir(pt2'))
{5}	access(pde_pt(pdir(pt1')))(ac1')

Applying decompose-equality,

This completes the proof of set_reference_accessible.7.

set_reference_accessible.8:

{-1}	Write?(access')
{-2}	pde_pt?(pdir(pt1'))
{-3}	not_present?(pdir(pt2'))
{-4}	Pdir(Pde_pt(pde_pt(pdir(pt1'))) WITH [accessed := TRUE])) = Pdir(Not_present)
{-5}	access(pde_pt(pdir(pt1')))(ac1')
{1}	ptab?(pt1')
{2}	ptab?(pt2')
{3}	access(pde_4m(pdir(pt2')))(ac1')

Applying decompose-equality,

C Proof scripts

This completes the proof of `set_reference_accessible.8`.

`set_reference_accessible.9`:

{-1}	Write?(access')
{-2}	pde_pt?(pdir(pt1'))
{-3}	pde_pt?(pdir(pt2'))
{-4}	Pdir(Pde_pt(pde_pt(pdir(pt1'))) WITH [accessed := TRUE])) = Pdir(Pde_pt(pde_pt(pdir(pt2'))) WITH [accessed := TRUE]))
{-5}	access(pde_pt(pdir(pt1')))(ac1')
{1}	ptab?(pt1')
{2}	ptab?(pt2')
{3}	access(pde_pt(pdir(pt2')))(ac1')

Applying decompose-equality,

Applying decompose-equality,

Applying decompose-equality,

This completes the proof of `set_reference_accessible.9`.

`set_reference_accessible.10`:

{-1}	Write?(access')
{-2}	pde_pt?(pdir(pt1'))
{-3}	Pdir(Pde_pt(pde_pt(pdir(pt1'))) WITH [accessed := TRUE])) = Pdir(Pde_4m(pde_4m(pdir(pt2'))) WITH [dirty := TRUE, accessed := TRUE]))
{-4}	access(pde_pt(pdir(pt1')))(ac1')
{1}	ptab?(pt1')
{2}	ptab?(pt2')
{3}	not_present?(pdir(pt2'))
{4}	pde_pt?(pdir(pt2'))
{5}	access(pde_4m(pdir(pt2')))(ac1')

Applying decompose-equality,

This completes the proof of `set_reference_accessible.10`.

`set_reference_accessible.11`:

{-1}	Write?(access')
{-2}	not_present?(pdir(pt2'))
{-3}	Pdir(Pde_4m(pde_4m(pdir(pt1'))) WITH [dirty := TRUE, accessed := TRUE])) = Pdir(Not_present)
{-4}	access(pde_4m(pdir(pt2')))(ac1')
{1}	ptab?(pt1')
{2}	not_present?(pdir(pt1'))
{3}	pde_pt?(pdir(pt1'))
{4}	ptab?(pt2')
{5}	access(pde_4m(pdir(pt1')))(ac1')

Applying decompose-equality,

This completes the proof of `set_reference_accessible.11`.

`set_reference_accessible.12:`

{-1}	Write?(access')
{-2}	pde_pt?(pdir(pt2'))
{-3}	Pdir(Pde_4m(pde_4m(pdir(pt1'))) WITH [dirty := TRUE, accessed := TRUE])) = Pdir(Pde_pt(pde_pt(pdir(pt2'))) WITH [accessed := TRUE]))
{-4}	access(pde_pt(pdir(pt2')))(ac1')
{1}	ptab?(pt1')
{2}	not_present?(pdir(pt1'))
{3}	pde_pt?(pdir(pt1'))
{4}	ptab?(pt2')
{5}	access(pde_4m(pdir(pt1')))(ac1')

Applying decompose-equality,

This completes the proof of `set_reference_accessible.12`.

`set_reference_accessible.13:`

{-1}	Write?(access')
{-2}	Pdir(Pde_4m(pde_4m(pdir(pt1'))) WITH [dirty := TRUE, accessed := TRUE])) = Pdir(Pde_4m(pde_4m(pdir(pt2'))) WITH [dirty := TRUE, accessed := TRUE]))
{-3}	access(pde_4m(pdir(pt2')))(ac1')
{1}	ptab?(pt1')
{2}	not_present?(pdir(pt1'))
{3}	pde_pt?(pdir(pt1'))
{4}	ptab?(pt2')
{5}	not_present?(pdir(pt2'))
{6}	pde_pt?(pdir(pt2'))
{7}	access(pde_4m(pdir(pt1')))(ac1')

Applying decompose-equality,

Applying decompose-equality,

Applying decompose-equality,

This completes the proof of `set_reference_accessible.13`.

`set_reference_accessible.14:`

{-1}	Write?(access')
{-2}	not_present?(pdir(pt2'))
{-3}	Pdir(Pde_4m(pde_4m(pdir(pt1'))) WITH [dirty := TRUE, accessed := TRUE])) = Pdir(Not_present)
{-4}	access(pde_4m(pdir(pt1')))(ac1')
{1}	ptab?(pt1')
{2}	not_present?(pdir(pt1'))
{3}	pde_pt?(pdir(pt1'))
{4}	ptab?(pt2')
{5}	access(pde_4m(pdir(pt2')))(ac1')

Applying decompose-equality,

This completes the proof of `set_reference_accessible.14`.

C Proof scripts

`set_reference_accessible.15:`

{-1}	Write?(access')
{-2}	pde_pt?(pdir(pt2'))
{-3}	Pdir(Pde_4m(pde_4m(pdir(pt1'))) WITH [dirty := TRUE, accessed := TRUE])) = Pdir(Pde_pt(pde_pt(pdir(pt2'))) WITH [accessed := TRUE]))
{-4}	access(pde_4m(pdir(pt1')))(ac1')
{1}	ptab?(pt1')
{2}	not_present?(pdir(pt1'))
{3}	pde_pt?(pdir(pt1'))
{4}	ptab?(pt2')
{5}	access(pde_pt(pdir(pt2')))(ac1')

Applying decompose-equality,

This completes the proof of `set_reference_accessible.15`.

`set_reference_accessible.16:`

{-1}	Write?(access')
{-2}	Pdir(Pde_4m(pde_4m(pdir(pt1'))) WITH [dirty := TRUE, accessed := TRUE])) = Pdir(Pde_4m(pde_4m(pdir(pt2'))) WITH [dirty := TRUE, accessed := TRUE]))
{-3}	access(pde_4m(pdir(pt1')))(ac1')
{1}	ptab?(pt1')
{2}	not_present?(pdir(pt1'))
{3}	pde_pt?(pdir(pt1'))
{4}	ptab?(pt2')
{5}	not_present?(pdir(pt2'))
{6}	pde_pt?(pdir(pt2'))
{7}	access(pde_4m(pdir(pt2')))(ac1')

Applying decompose-equality,

Applying decompose-equality,

Applying decompose-equality,

This completes the proof of `set_reference_accessible.16`.

`set_reference_accessible.17:`

{-1}	ptab?(pt1')
{-2}	ptab?(pt2')
{-3}	not_present?(ptab(pt2'))
{-4}	Ptab(Pte(pte(ptab(pt1'))) WITH [accessed := TRUE])) = Ptab(Not_present)
{-5}	access(pte(ptab(pt1')))(ac1')
{1}	Write?(access')
{2}	not_present?(ptab(pt1'))
{3}	access(pte(ptab(pt2')))(ac1')

Applying decompose-equality,

This completes the proof of `set_reference_accessible.17`.

set_reference_accessible.18:

{-1}	ptab?(pt1')
{-2}	ptab?(pt2')
{-3}	Ptab(Pte(pte(ptab(pt1'))) WITH [accessed := TRUE])) = Ptab(Pte(pte(ptab(pt2'))) WITH [accessed := TRUE]))
{-4}	access(pte(ptab(pt1')))(ac1')
{1}	Write?(access')
{2}	not_present?(ptab(pt1'))
{3}	not_present?(ptab(pt2'))
{4}	access(pte(ptab(pt2')))(ac1')

Applying decompose-equality,

Applying decompose-equality,

Applying decompose-equality,

This completes the proof of set_reference_accessible.18.

set_reference_accessible.19:

{-1}	ptab?(pt1')
{-2}	ptab?(pt2')
{-3}	not_present?(ptab(pt2'))
{-4}	Ptab(Pte(pte(ptab(pt1'))) WITH [accessed := TRUE])) = Ptab(Not_present)
{-5}	access(pte(ptab(pt2')))(ac1')
{1}	Write?(access')
{2}	not_present?(ptab(pt1'))
{3}	access(pte(ptab(pt1')))(ac1')

Applying decompose-equality,

This completes the proof of set_reference_accessible.19.

set_reference_accessible.20:

{-1}	ptab?(pt1')
{-2}	ptab?(pt2')
{-3}	Ptab(Pte(pte(ptab(pt1'))) WITH [accessed := TRUE])) = Ptab(Pte(pte(ptab(pt2'))) WITH [accessed := TRUE]))
{-4}	access(pte(ptab(pt2')))(ac1')
{1}	Write?(access')
{2}	not_present?(ptab(pt1'))
{3}	not_present?(ptab(pt2'))
{4}	access(pte(ptab(pt1')))(ac1')

Applying decompose-equality,

Applying decompose-equality,

Applying decompose-equality,

This completes the proof of set_reference_accessible.20.

set_reference_accessible.21:

{-1}	pde_pt?(pdir(pt1'))
{-2}	not_present?(pdir(pt2'))
{-3}	Pdir(Pde_pt(pde_pt(pdir(pt1'))) WITH [accessed := TRUE])) = Pdir(Not_present)
{-4}	access(pde_4m(pdir(pt2')))(ac1')
{1}	Write?(access')
{2}	ptab?(pt1')
{3}	ptab?(pt2')
{4}	access(pde_pt(pdir(pt1')))(ac1')

C Proof scripts

Applying decompose-equality,

This completes the proof of `set_reference_accessible.21`.

`set_reference_accessible.22`:

{-1}	<code>pde_pt?(pdir(pt1'))</code>	
{-2}	<code>pde_pt?(pdir(pt2'))</code>	
{-3}	<code>Pdir(Pde_pt(pde_pt(pdir(pt1'))) WITH [accessed := TRUE])) =</code>	
	<code>Pdir(Pde_pt(pde_pt(pdir(pt2'))) WITH [accessed := TRUE]))</code>	
{-4}	<code>access(pde_pt(pdir(pt2')))(ac1')</code>	
{1}	<code>Write?(access')</code>	
{2}	<code>ptab?(pt1')</code>	
{3}	<code>ptab?(pt2')</code>	
{4}	<code>access(pde_pt(pdir(pt1')))(ac1')</code>	

Applying decompose-equality,

Applying decompose-equality,

Applying decompose-equality,

This completes the proof of `set_reference_accessible.22`.

`set_reference_accessible.23`:

{-1}	<code>pde_pt?(pdir(pt1'))</code>	
{-2}	<code>Pdir(Pde_pt(pde_pt(pdir(pt1'))) WITH [accessed := TRUE])) =</code>	
	<code>Pdir(Pde_4m(pde_4m(pdir(pt2'))) WITH [accessed := TRUE]))</code>	
{-3}	<code>access(pde_4m(pdir(pt2')))(ac1')</code>	
{1}	<code>Write?(access')</code>	
{2}	<code>ptab?(pt1')</code>	
{3}	<code>ptab?(pt2')</code>	
{4}	<code>not_present?(pdir(pt2'))</code>	
{5}	<code>pde_pt?(pdir(pt2'))</code>	
{6}	<code>access(pde_pt(pdir(pt1')))(ac1')</code>	

Applying decompose-equality,

This completes the proof of `set_reference_accessible.23`.

`set_reference_accessible.24`:

{-1}	<code>pde_pt?(pdir(pt1'))</code>	
{-2}	<code>not_present?(pdir(pt2'))</code>	
{-3}	<code>Pdir(Pde_pt(pde_pt(pdir(pt1'))) WITH [accessed := TRUE])) = Pdir(Not_present)</code>	
{-4}	<code>access(pde_pt(pdir(pt1')))(ac1')</code>	
{1}	<code>Write?(access')</code>	
{2}	<code>ptab?(pt1')</code>	
{3}	<code>ptab?(pt2')</code>	
{4}	<code>access(pde_4m(pdir(pt2')))(ac1')</code>	

Applying decompose-equality,

This completes the proof of `set_reference_accessible.24`.

set_reference_accessible.25:

{-1}	pde_pt?(pdir(pt1'))
{-2}	pde_pt?(pdir(pt2'))
{-3}	Pdir(Pde_pt(pde_pt(pdir(pt1'))) WITH [accessed := TRUE])) = Pdir(Pde_pt(pde_pt(pdir(pt2'))) WITH [accessed := TRUE]))
{-4}	access(pde_pt(pdir(pt1')))(ac1')
{1}	Write?(access')
{2}	ptab?(pt1')
{3}	ptab?(pt2')
{4}	access(pde_pt(pdir(pt2')))(ac1')

Applying decompose-equality,

Applying decompose-equality,

Applying decompose-equality,

This completes the proof of set_reference_accessible.25.

set_reference_accessible.26:

{-1}	pde_pt?(pdir(pt1'))
{-2}	Pdir(Pde_pt(pde_pt(pdir(pt1'))) WITH [accessed := TRUE])) = Pdir(Pde_4m(pde_4m(pdir(pt2'))) WITH [accessed := TRUE]))
{-3}	access(pde_pt(pdir(pt1')))(ac1')
{1}	Write?(access')
{2}	ptab?(pt1')
{3}	ptab?(pt2')
{4}	not_present?(pdir(pt2'))
{5}	pde_pt?(pdir(pt2'))
{6}	access(pde_4m(pdir(pt2')))(ac1')

Applying decompose-equality,

This completes the proof of set_reference_accessible.26.

set_reference_accessible.27:

{-1}	not_present?(pdir(pt2'))
{-2}	Pdir(Pde_4m(pde_4m(pdir(pt1'))) WITH [accessed := TRUE])) = Pdir(Not_present)
{-3}	access(pde_4m(pdir(pt2')))(ac1')
{1}	Write?(access')
{2}	ptab?(pt1')
{3}	not_present?(pdir(pt1'))
{4}	pde_pt?(pdir(pt1'))
{5}	ptab?(pt2')
{6}	access(pde_4m(pdir(pt1')))(ac1')

Applying decompose-equality,

This completes the proof of set_reference_accessible.27.

C Proof scripts

`set_reference_accessible.28:`

{-1}	<code>pde_pt?(pdir(pt2'))</code>	
{-2}	<code>Pdir(Pde_4m(pde_4m(pdir(pt1'))) WITH [accessed := TRUE])) =</code>	
	<code>Pdir(Pde_pt(pde_pt(pdir(pt2'))) WITH [accessed := TRUE]))</code>	
{-3}	<code>access(pde_pt(pdir(pt2')))(ac1')</code>	
{1}	<code>Write?(access')</code>	
{2}	<code>ptab?(pt1')</code>	
{3}	<code>not_present?(pdir(pt1'))</code>	
{4}	<code>pde_pt?(pdir(pt1'))</code>	
{5}	<code>ptab?(pt2')</code>	
{6}	<code>access(pde_4m(pdir(pt1')))(ac1')</code>	

Applying `decompose-equality`,

This completes the proof of `set_reference_accessible.28`.

`set_reference_accessible.29:`

{-1}	<code>Pdir(Pde_4m(pde_4m(pdir(pt1'))) WITH [accessed := TRUE])) =</code>	
	<code>Pdir(Pde_4m(pde_4m(pdir(pt2'))) WITH [accessed := TRUE]))</code>	
{-2}	<code>access(pde_4m(pdir(pt2')))(ac1')</code>	
{1}	<code>Write?(access')</code>	
{2}	<code>ptab?(pt1')</code>	
{3}	<code>not_present?(pdir(pt1'))</code>	
{4}	<code>pde_pt?(pdir(pt1'))</code>	
{5}	<code>ptab?(pt2')</code>	
{6}	<code>not_present?(pdir(pt2'))</code>	
{7}	<code>pde_pt?(pdir(pt2'))</code>	
{8}	<code>access(pde_4m(pdir(pt1')))(ac1')</code>	

Applying `decompose-equality`,

Applying `decompose-equality`,

Applying `decompose-equality`,

This completes the proof of `set_reference_accessible.29`.

`set_reference_accessible.30:`

{-1}	<code>not_present?(pdir(pt2'))</code>	
{-2}	<code>Pdir(Pde_4m(pde_4m(pdir(pt1'))) WITH [accessed := TRUE])) = Pdir(Not_present)</code>	
{-3}	<code>access(pde_4m(pdir(pt1')))(ac1')</code>	
{1}	<code>Write?(access')</code>	
{2}	<code>ptab?(pt1')</code>	
{3}	<code>not_present?(pdir(pt1'))</code>	
{4}	<code>pde_pt?(pdir(pt1'))</code>	
{5}	<code>ptab?(pt2')</code>	
{6}	<code>access(pde_4m(pdir(pt2')))(ac1')</code>	

Applying `decompose-equality`,

This completes the proof of `set_reference_accessible.30`.

`set_reference_accessible.31:`

{-1}	<code>pde_pt?(pdir(pt2'))</code>
{-2}	<code>Pdir(Pde_4m(pde_4m(pdir(pt1'))) WITH [accessed := TRUE])) = Pdir(Pde_pt(pde_pt(pdir(pt2'))) WITH [accessed := TRUE]))</code>
{-3}	<code>access(pde_4m(pdir(pt1')))(ac1')</code>
{1}	<code>Write?(access')</code>
{2}	<code>ptab?(pt1')</code>
{3}	<code>not_present?(pdir(pt1'))</code>
{4}	<code>pde_pt?(pdir(pt1'))</code>
{5}	<code>ptab?(pt2')</code>
{6}	<code>access(pde_pt(pdir(pt2')))(ac1')</code>

Applying `decompose-equality`,

This completes the proof of `set_reference_accessible.31`.

`set_reference_accessible.32:`

{-1}	<code>Pdir(Pde_4m(pde_4m(pdir(pt1'))) WITH [accessed := TRUE])) = Pdir(Pde_4m(pde_4m(pdir(pt2'))) WITH [accessed := TRUE]))</code>
{-2}	<code>access(pde_4m(pdir(pt1')))(ac1')</code>
{1}	<code>Write?(access')</code>
{2}	<code>ptab?(pt1')</code>
{3}	<code>not_present?(pdir(pt1'))</code>
{4}	<code>pde_pt?(pdir(pt1'))</code>
{5}	<code>ptab?(pt2')</code>
{6}	<code>not_present?(pdir(pt2'))</code>
{7}	<code>pde_pt?(pdir(pt2'))</code>
{8}	<code>access(pde_4m(pdir(pt2')))(ac1')</code>

Applying `decompose-equality`,

Applying `decompose-equality`,

Applying `decompose-equality`,

This completes the proof of `set_reference_accessible.32`.

Q.E.D.

C.159.12 *Paging_type_helpers.set_reference_privileged*

Terse proof for `set_reference_privileged`.

`set_reference_privileged:`

{1}	$\forall (pt1, pt2: \text{Paging_type}, \text{access}: \text{Memory_access}, \text{priv}: \text{Memory_privilege}):$ $\text{set_reference}(pt1, \text{access}) = \text{set_reference}(pt2, \text{access}) \wedge \text{present?}(pt1) \supset$ $\text{privileged?}(pt1, \text{priv}) = \text{privileged?}(pt2, \text{priv})$
-----	---

Trying repeated skolemization, instantiation, and if-lifting,

we get 32 subgoals:

C Proof scripts

`set_reference_privileged.1:`

{-1}	Write?(access')
{-2}	ptab?(pt1')
{-3}	ptab?(pt2')
{-4}	not_present?(ptab(pt2'))
{-5}	Ptab(Pte(pte(ptab(pt1')) WITH [dirty := TRUE, accessed := TRUE])) = Ptab(Not_present)
{-6}	User?(privilege(pte(ptab(pt1'))))
{1}	not_present?(ptab(pt1'))
{2}	Supervisor?(priv')
{3}	User?(privilege(pte(ptab(pt2'))))

Applying decompose-equality,

This completes the proof of `set_reference_privileged.1`.

`set_reference_privileged.2:`

{-1}	Write?(access')
{-2}	ptab?(pt1')
{-3}	ptab?(pt2')
{-4}	Ptab(Pte(pte(ptab(pt1')) WITH [dirty := TRUE, accessed := TRUE])) = Ptab(Pte(pte(ptab(pt2')) WITH [dirty := TRUE, accessed := TRUE]))
{-5}	User?(privilege(pte(ptab(pt1'))))
{1}	not_present?(ptab(pt1'))
{2}	not_present?(ptab(pt2'))
{3}	Supervisor?(priv')
{4}	User?(privilege(pte(ptab(pt2'))))

Applying decompose-equality,

Applying decompose-equality,

Applying decompose-equality,

This completes the proof of `set_reference_privileged.2`.

`set_reference_privileged.3:`

{-1}	Write?(access')
{-2}	ptab?(pt1')
{-3}	ptab?(pt2')
{-4}	not_present?(ptab(pt2'))
{-5}	Ptab(Pte(pte(ptab(pt1')) WITH [dirty := TRUE, accessed := TRUE])) = Ptab(Not_present)
{-6}	User?(privilege(pte(ptab(pt2'))))
{1}	not_present?(ptab(pt1'))
{2}	User?(privilege(pte(ptab(pt1'))))
{3}	Supervisor?(priv')

Applying decompose-equality,

This completes the proof of `set_reference_privileged.3`.

set_reference_privileged.4:

{-1}	Write?(access')
{-2}	ptab?(pt1')
{-3}	ptab?(pt2')
{-4}	Ptab(Pte(pte(ptab(pt1'))) WITH [dirty := TRUE, accessed := TRUE])) = Ptab(Pte(pte(ptab(pt2'))) WITH [dirty := TRUE, accessed := TRUE]))
{-5}	User?(privilege(pte(ptab(pt2'))))
{1}	not_present?(ptab(pt1'))
{2}	not_present?(ptab(pt2'))
{3}	User?(privilege(pte(ptab(pt1'))))
{4}	Supervisor?(priv')

Applying decompose-equality,

Applying decompose-equality,

Applying decompose-equality,

This completes the proof of set_reference_privileged.4.

set_reference_privileged.5:

{-1}	Write?(access')
{-2}	pde_pt?(pdir(pt1'))
{-3}	not_present?(pdir(pt2'))
{-4}	Pdir(Pde_pt(pde_pt(pdir(pt1'))) WITH [accessed := TRUE])) = Pdir(Not_present)
{-5}	User?(privilege(pde_4m(pdir(pt2'))))
{1}	ptab?(pt1')
{2}	ptab?(pt2')
{3}	Supervisor?(priv')
{4}	User?(privilege(pde_pt(pdir(pt1'))))

Applying decompose-equality,

This completes the proof of set_reference_privileged.5.

set_reference_privileged.6:

{-1}	Write?(access')
{-2}	pde_pt?(pdir(pt1'))
{-3}	pde_pt?(pdir(pt2'))
{-4}	Pdir(Pde_pt(pde_pt(pdir(pt1'))) WITH [accessed := TRUE])) = Pdir(Pde_pt(pde_pt(pdir(pt2'))) WITH [accessed := TRUE]))
{-5}	User?(privilege(pde_pt(pdir(pt2'))))
{1}	ptab?(pt1')
{2}	ptab?(pt2')
{3}	Supervisor?(priv')
{4}	User?(privilege(pde_pt(pdir(pt1'))))

Applying decompose-equality,

Applying decompose-equality,

Applying decompose-equality,

This completes the proof of set_reference_privileged.6.

C Proof scripts

set_reference_privileged.7:

{-1}	Write?(access')
{-2}	pde_pt?(pdir(pt1'))
{-3}	Pdir(Pde_pt(pde_pt(pdir(pt1'))) WITH [accessed := TRUE])) = Pdir(Pde_4m(pde_4m(pdir(pt2'))) WITH [dirty := TRUE, accessed := TRUE]))
{-4}	User?(privilege(pde_4m(pdir(pt2'))))
{1}	ptab?(pt1')
{2}	ptab?(pt2')
{3}	not_present?(pdir(pt2'))
{4}	pde_pt?(pdir(pt2'))
{5}	Supervisor?(priv')
{6}	User?(privilege(pde_pt(pdir(pt1'))))

Applying decompose-equality,

This completes the proof of set_reference_privileged.7.

set_reference_privileged.8:

{-1}	Write?(access')
{-2}	pde_pt?(pdir(pt1'))
{-3}	not_present?(pdir(pt2'))
{-4}	Pdir(Pde_pt(pde_pt(pdir(pt1'))) WITH [accessed := TRUE])) = Pdir(Not_present)
{-5}	User?(privilege(pde_pt(pdir(pt1'))))
{1}	ptab?(pt1')
{2}	ptab?(pt2')
{3}	Supervisor?(priv')
{4}	User?(privilege(pde_4m(pdir(pt2'))))

Applying decompose-equality,

This completes the proof of set_reference_privileged.8.

set_reference_privileged.9:

{-1}	Write?(access')
{-2}	pde_pt?(pdir(pt1'))
{-3}	pde_pt?(pdir(pt2'))
{-4}	Pdir(Pde_pt(pde_pt(pdir(pt1'))) WITH [accessed := TRUE])) = Pdir(Pde_pt(pde_pt(pdir(pt2'))) WITH [accessed := TRUE]))
{-5}	User?(privilege(pde_pt(pdir(pt1'))))
{1}	ptab?(pt1')
{2}	ptab?(pt2')
{3}	Supervisor?(priv')
{4}	User?(privilege(pde_pt(pdir(pt2'))))

Applying decompose-equality,

Applying decompose-equality,

Applying decompose-equality,

This completes the proof of set_reference_privileged.9.

set_reference_privileged.10:

{-1}	Write?(access')
{-2}	pde_pt?(pdir(pt1'))
{-3}	Pdir(Pde_pt(pde_pt(pdir(pt1'))) WITH [accessed := TRUE])) = Pdir(Pde_4m(pde_4m(pdir(pt2'))) WITH [dirty := TRUE, accessed := TRUE]))
{-4}	User?(privilege(pde_pt(pdir(pt1'))))
{1}	ptab?(pt1')
{2}	ptab?(pt2')
{3}	not_present?(pdir(pt2'))
{4}	pde_pt?(pdir(pt2'))
{5}	Supervisor?(priv')
{6}	User?(privilege(pde_4m(pdir(pt2'))))

Applying decompose-equality,

This completes the proof of set_reference_privileged.10.

set_reference_privileged.11:

{-1}	Write?(access')
{-2}	not_present?(pdir(pt2'))
{-3}	Pdir(Pde_4m(pde_4m(pdir(pt1'))) WITH [dirty := TRUE, accessed := TRUE])) = Pdir(Not_present)
{-4}	User?(privilege(pde_4m(pdir(pt2'))))
{1}	ptab?(pt1')
{2}	not_present?(pdir(pt1'))
{3}	pde_pt?(pdir(pt1'))
{4}	ptab?(pt2')
{5}	Supervisor?(priv')
{6}	User?(privilege(pde_4m(pdir(pt1'))))

Applying decompose-equality,

This completes the proof of set_reference_privileged.11.

set_reference_privileged.12:

{-1}	Write?(access')
{-2}	pde_pt?(pdir(pt2'))
{-3}	Pdir(Pde_4m(pde_4m(pdir(pt1'))) WITH [dirty := TRUE, accessed := TRUE])) = Pdir(Pde_pt(pde_pt(pdir(pt2'))) WITH [accessed := TRUE]))
{-4}	User?(privilege(pde_pt(pdir(pt2'))))
{1}	ptab?(pt1')
{2}	not_present?(pdir(pt1'))
{3}	pde_pt?(pdir(pt1'))
{4}	ptab?(pt2')
{5}	Supervisor?(priv')
{6}	User?(privilege(pde_4m(pdir(pt1'))))

Applying decompose-equality,

This completes the proof of set_reference_privileged.12.

set_reference_privileged.13:

{-1}	Write?(access')
{-2}	Pdir(Pde_4m(pde_4m(pdir(pt1'))) WITH [dirty := TRUE, accessed := TRUE])) = Pdir(Pde_4m(pde_4m(pdir(pt2'))) WITH [dirty := TRUE, accessed := TRUE]))
{-3}	User?(privilege(pde_4m(pdir(pt2'))))
{1}	ptab?(pt1')
{2}	not_present?(pdir(pt1'))
{3}	pde_pt?(pdir(pt1'))
{4}	ptab?(pt2')
{5}	not_present?(pdir(pt2'))
{6}	pde_pt?(pdir(pt2'))
{7}	Supervisor?(priv')
{8}	User?(privilege(pde_4m(pdir(pt1'))))

Applying decompose-equality,

Applying decompose-equality,

Applying decompose-equality,

This completes the proof of set_reference_privileged.13.

set_reference_privileged.14:

{-1}	Write?(access')
{-2}	not_present?(pdir(pt2'))
{-3}	Pdir(Pde_4m(pde_4m(pdir(pt1'))) WITH [dirty := TRUE, accessed := TRUE])) = Pdir(Not_present)
{-4}	User?(privilege(pde_4m(pdir(pt1'))))
{1}	ptab?(pt1')
{2}	not_present?(pdir(pt1'))
{3}	pde_pt?(pdir(pt1'))
{4}	ptab?(pt2')
{5}	Supervisor?(priv')
{6}	User?(privilege(pde_4m(pdir(pt2'))))

Applying decompose-equality,

This completes the proof of set_reference_privileged.14.

set_reference_privileged.15:

{-1}	Write?(access')
{-2}	pde_pt?(pdir(pt2'))
{-3}	Pdir(Pde_4m(pde_4m(pdir(pt1'))) WITH [dirty := TRUE, accessed := TRUE])) = Pdir(Pde_pt(pde_pt(pdir(pt2'))) WITH [accessed := TRUE]))
{-4}	User?(privilege(pde_4m(pdir(pt1'))))
{1}	ptab?(pt1')
{2}	not_present?(pdir(pt1'))
{3}	pde_pt?(pdir(pt1'))
{4}	ptab?(pt2')
{5}	Supervisor?(priv')
{6}	User?(privilege(pde_pt(pdir(pt2'))))

Applying decompose-equality,

This completes the proof of set_reference_privileged.15.

set_reference_privileged.16:

{-1}	Write?(access')
{-2}	Pdir(Pde_4m(pde_4m(pdir(pt1'))) WITH [dirty := TRUE, accessed := TRUE])) = Pdir(Pde_4m(pde_4m(pdir(pt2'))) WITH [dirty := TRUE, accessed := TRUE]))
{-3}	User?(privilege(pde_4m(pdir(pt1'))))
{1}	ptab?(pt1')
{2}	not_present?(pdir(pt1'))
{3}	pde_pt?(pdir(pt1'))
{4}	ptab?(pt2')
{5}	not_present?(pdir(pt2'))
{6}	pde_pt?(pdir(pt2'))
{7}	Supervisor?(priv')
{8}	User?(privilege(pde_4m(pdir(pt2'))))

Applying decompose-equality,

Applying decompose-equality,

Applying decompose-equality,

This completes the proof of set_reference_privileged.16.

set_reference_privileged.17:

{-1}	ptab?(pt1')
{-2}	ptab?(pt2')
{-3}	not_present?(ptab(pt2'))
{-4}	Ptab(Pte(pte(ptab(pt1'))) WITH [accessed := TRUE])) = Ptab(Not_present)
{-5}	User?(privilege(pte(ptab(pt1'))))
{1}	Write?(access')
{2}	not_present?(ptab(pt1'))
{3}	Supervisor?(priv')
{4}	User?(privilege(pte(ptab(pt2'))))

Applying decompose-equality,

This completes the proof of set_reference_privileged.17.

set_reference_privileged.18:

{-1}	ptab?(pt1')
{-2}	ptab?(pt2')
{-3}	Ptab(Pte(pte(ptab(pt1'))) WITH [accessed := TRUE])) = Ptab(Pte(pte(ptab(pt2'))) WITH [accessed := TRUE]))
{-4}	User?(privilege(pte(ptab(pt1'))))
{1}	Write?(access')
{2}	not_present?(ptab(pt1'))
{3}	not_present?(ptab(pt2'))
{4}	Supervisor?(priv')
{5}	User?(privilege(pte(ptab(pt2'))))

Applying decompose-equality,

Applying decompose-equality,

Applying decompose-equality,

This completes the proof of set_reference_privileged.18.

C Proof scripts

`set_reference_privileged.19:`

{-1}	ptab?(pt1')
{-2}	ptab?(pt2')
{-3}	not_present?(ptab(pt2'))
{-4}	Ptab(Pte(pte(ptab(pt1'))) WITH [accessed := TRUE])) = Ptab(Not_present)
{-5}	User?(privilege(pte(ptab(pt2'))))
{1}	Write?(access')
{2}	not_present?(ptab(pt1'))
{3}	User?(privilege(pte(ptab(pt1'))))
{4}	Supervisor?(priv')

Applying decompose-equality,

This completes the proof of `set_reference_privileged.19`.

`set_reference_privileged.20:`

{-1}	ptab?(pt1')
{-2}	ptab?(pt2')
{-3}	Ptab(Pte(pte(ptab(pt1'))) WITH [accessed := TRUE])) = Ptab(Pte(pte(ptab(pt2'))) WITH [accessed := TRUE]))
{-4}	User?(privilege(pte(ptab(pt2'))))
{1}	Write?(access')
{2}	not_present?(ptab(pt1'))
{3}	not_present?(ptab(pt2'))
{4}	User?(privilege(pte(ptab(pt1'))))
{5}	Supervisor?(priv')

Applying decompose-equality,

Applying decompose-equality,

Applying decompose-equality,

This completes the proof of `set_reference_privileged.20`.

`set_reference_privileged.21:`

{-1}	pde_pt?(pdir(pt1'))
{-2}	not_present?(pdir(pt2'))
{-3}	Pdir(Pde_pt(pde_pt(pdir(pt1'))) WITH [accessed := TRUE])) = Pdir(Not_present)
{-4}	User?(privilege(pde_4m(pdir(pt2'))))
{1}	Write?(access')
{2}	ptab?(pt1')
{3}	ptab?(pt2')
{4}	Supervisor?(priv')
{5}	User?(privilege(pde_pt(pdir(pt1'))))

Applying decompose-equality,

This completes the proof of `set_reference_privileged.21`.

set_reference_privileged.22:

{-1}	pde_pt?(pdir(pt1'))
{-2}	pde_pt?(pdir(pt2'))
{-3}	Pdir(Pde_pt(pde_pt(pdir(pt1'))) WITH [accessed := TRUE])) = Pdir(Pde_pt(pde_pt(pdir(pt2'))) WITH [accessed := TRUE]))
{-4}	User?(privilege(pde_pt(pdir(pt2'))))
{1}	Write?(access')
{2}	ptab?(pt1')
{3}	ptab?(pt2')
{4}	Supervisor?(priv')
{5}	User?(privilege(pde_pt(pdir(pt1'))))

Applying decompose-equality,

Applying decompose-equality,

Applying decompose-equality,

This completes the proof of set_reference_privileged.22.

set_reference_privileged.23:

{-1}	pde_pt?(pdir(pt1'))
{-2}	Pdir(Pde_pt(pde_pt(pdir(pt1'))) WITH [accessed := TRUE])) = Pdir(Pde_4m(pde_4m(pdir(pt2'))) WITH [accessed := TRUE]))
{-3}	User?(privilege(pde_4m(pdir(pt2'))))
{1}	Write?(access')
{2}	ptab?(pt1')
{3}	ptab?(pt2')
{4}	not_present?(pdir(pt2'))
{5}	pde_pt?(pdir(pt2'))
{6}	Supervisor?(priv')
{7}	User?(privilege(pde_pt(pdir(pt1'))))

Applying decompose-equality,

This completes the proof of set_reference_privileged.23.

set_reference_privileged.24:

{-1}	pde_pt?(pdir(pt1'))
{-2}	not_present?(pdir(pt2'))
{-3}	Pdir(Pde_pt(pde_pt(pdir(pt1'))) WITH [accessed := TRUE])) = Pdir(Not_present)
{-4}	User?(privilege(pde_pt(pdir(pt1'))))
{1}	Write?(access')
{2}	ptab?(pt1')
{3}	ptab?(pt2')
{4}	Supervisor?(priv')
{5}	User?(privilege(pde_4m(pdir(pt2'))))

Applying decompose-equality,

This completes the proof of set_reference_privileged.24.

set_reference_privileged.25:

{-1}	pde_pt?(pdir(pt1'))
{-2}	pde_pt?(pdir(pt2'))
{-3}	Pdir(Pde_pt(pde_pt(pdir(pt1'))) WITH [accessed := TRUE])) = Pdir(Pde_pt(pde_pt(pdir(pt2'))) WITH [accessed := TRUE]))
{-4}	User?(privilege(pde_pt(pdir(pt1'))))
{1}	Write?(access')
{2}	ptab?(pt1')
{3}	ptab?(pt2')
{4}	Supervisor?(priv')
{5}	User?(privilege(pde_pt(pdir(pt2'))))

Applying decompose-equality,

Applying decompose-equality,

Applying decompose-equality,

This completes the proof of set_reference_privileged.25.

set_reference_privileged.26:

{-1}	pde_pt?(pdir(pt1'))
{-2}	Pdir(Pde_pt(pde_pt(pdir(pt1'))) WITH [accessed := TRUE])) = Pdir(Pde_4m(pde_4m(pdir(pt2'))) WITH [accessed := TRUE]))
{-3}	User?(privilege(pde_pt(pdir(pt1'))))
{1}	Write?(access')
{2}	ptab?(pt1')
{3}	ptab?(pt2')
{4}	not_present?(pdir(pt2'))
{5}	pde_pt?(pdir(pt2'))
{6}	Supervisor?(priv')
{7}	User?(privilege(pde_4m(pdir(pt2'))))

Applying decompose-equality,

This completes the proof of set_reference_privileged.26.

set_reference_privileged.27:

{-1}	not_present?(pdir(pt2'))
{-2}	Pdir(Pde_4m(pde_4m(pdir(pt1'))) WITH [accessed := TRUE])) = Pdir(Not_present)
{-3}	User?(privilege(pde_4m(pdir(pt2'))))
{1}	Write?(access')
{2}	ptab?(pt1')
{3}	not_present?(pdir(pt1'))
{4}	pde_pt?(pdir(pt1'))
{5}	ptab?(pt2')
{6}	Supervisor?(priv')
{7}	User?(privilege(pde_4m(pdir(pt1'))))

Applying decompose-equality,

This completes the proof of set_reference_privileged.27.

set_reference_privileged.28:

{-1}	pde_pt?(pdir(pt2'))
{-2}	Pdir(Pde_4m(pde_4m(pdir(pt1'))) WITH [accessed := TRUE])) = Pdir(Pde_pt(pde_pt(pdir(pt2'))) WITH [accessed := TRUE]))
{-3}	User?(privilege(pde_pt(pdir(pt2'))))
{1}	Write?(access')
{2}	ptab?(pt1')
{3}	not_present?(pdir(pt1'))
{4}	pde_pt?(pdir(pt1'))
{5}	ptab?(pt2')
{6}	Supervisor?(priv')
{7}	User?(privilege(pde_4m(pdir(pt1'))))

Applying decompose-equality,

This completes the proof of set_reference_privileged.28.

set_reference_privileged.29:

{-1}	Pdir(Pde_4m(pde_4m(pdir(pt1'))) WITH [accessed := TRUE])) = Pdir(Pde_4m(pde_4m(pdir(pt2'))) WITH [accessed := TRUE]))
{-2}	User?(privilege(pde_4m(pdir(pt2'))))
{1}	Write?(access')
{2}	ptab?(pt1')
{3}	not_present?(pdir(pt1'))
{4}	pde_pt?(pdir(pt1'))
{5}	ptab?(pt2')
{6}	not_present?(pdir(pt2'))
{7}	pde_pt?(pdir(pt2'))
{8}	Supervisor?(priv')
{9}	User?(privilege(pde_4m(pdir(pt1'))))

Applying decompose-equality,

Applying decompose-equality,

Applying decompose-equality,

This completes the proof of set_reference_privileged.29.

set_reference_privileged.30:

{-1}	not_present?(pdir(pt2'))
{-2}	Pdir(Pde_4m(pde_4m(pdir(pt1'))) WITH [accessed := TRUE])) = Pdir(Not_present)
{-3}	User?(privilege(pde_4m(pdir(pt1'))))
{1}	Write?(access')
{2}	ptab?(pt1')
{3}	not_present?(pdir(pt1'))
{4}	pde_pt?(pdir(pt1'))
{5}	ptab?(pt2')
{6}	Supervisor?(priv')
{7}	User?(privilege(pde_4m(pdir(pt2'))))

Applying decompose-equality,

This completes the proof of set_reference_privileged.30.

set_reference_privileged.31:

{-1}	pde_pt?(pdir(pt2'))
{-2}	Pdir(Pde_4m(pde_4m(pdir(pt1'))) WITH [accessed := TRUE])) = Pdir(Pde_pt(pde_pt(pdir(pt2'))) WITH [accessed := TRUE]))
{-3}	User?(privilege(pde_4m(pdir(pt1'))))
{1}	Write?(access')
{2}	ptab?(pt1')
{3}	not_present?(pdir(pt1'))
{4}	pde_pt?(pdir(pt1'))
{5}	ptab?(pt2')
{6}	Supervisor?(priv')
{7}	User?(privilege(pde_pt(pdir(pt2'))))

Applying decompose-equality,

This completes the proof of set_reference_privileged.31.

set_reference_privileged.32:

{-1}	Pdir(Pde_4m(pde_4m(pdir(pt1'))) WITH [accessed := TRUE])) = Pdir(Pde_4m(pde_4m(pdir(pt2'))) WITH [accessed := TRUE]))
{-2}	User?(privilege(pde_4m(pdir(pt1'))))
{1}	Write?(access')
{2}	ptab?(pt1')
{3}	not_present?(pdir(pt1'))
{4}	pde_pt?(pdir(pt1'))
{5}	ptab?(pt2')
{6}	not_present?(pdir(pt2'))
{7}	pde_pt?(pdir(pt2'))
{8}	Supervisor?(priv')
{9}	User?(privilege(pde_4m(pdir(pt2'))))

Applying decompose-equality,

Applying decompose-equality,

Applying decompose-equality,

This completes the proof of set_reference_privileged.32.

Q.E.D.

C.159.13 Paging_type_helpers.set_reference_leaf

Terse proof for set_reference_leaf.

set_reference_leaf:

{1}	$\forall (pt1, pt2: \text{Paging_type}, \text{access}: \text{Memory_access}):$ set_reference(pt1, access) = set_reference(pt2, access) \wedge present?(pt1) \supset is_leaf?(pt1) = is_leaf?(pt2)
-----	---

Trying repeated skolemization, instantiation, and if-lifting,

we get 6 subgoals:

set_reference_leaf.1:

{-1}	Write?(access')
{-2}	pdir?(pt1')
{-3}	pde_pt?(pdir(pt1'))
{-4}	not_present?(pdir(pt2'))
{-5}	Pdir(Pde_pt(pde_pt(pdir(pt1'))) WITH [accessed := TRUE])) = Pdir(Not_present)
{1}	ptab?(pt2')

Applying decompose-equality,

This completes the proof of **set_reference_leaf.1**.

set_reference_leaf.2:

{-1}	Write?(access')
{-2}	pdir?(pt1')
{-3}	pde_pt?(pdir(pt1'))
{-4}	Pdir(Pde_pt(pde_pt(pdir(pt1'))) WITH [accessed := TRUE])) = Pdir(Pde_4m(pde_4m(pdir(pt2'))) WITH [dirty := TRUE, accessed := TRUE]))
{1}	ptab?(pt2')
{2}	not_present?(pdir(pt2'))
{3}	pde_pt?(pdir(pt2'))

Applying decompose-equality,

This completes the proof of **set_reference_leaf.2**.

set_reference_leaf.3:

{-1}	Write?(access')
{-2}	pdir?(pt1')
{-3}	Pdir(Pde_4m(pde_4m(pdir(pt1'))) WITH [dirty := TRUE, accessed := TRUE])) = Pdir(Pde_pt(pde_pt(pdir(pt2'))) WITH [accessed := TRUE]))
{-4}	pdir?(pt2')
{-5}	pde_pt?(pdir(pt2'))
{1}	not_present?(pdir(pt1'))
{2}	pde_pt?(pdir(pt1'))

Applying decompose-equality,

This completes the proof of **set_reference_leaf.3**.

set_reference_leaf.4:

{-1}	pdir?(pt1')
{-2}	pde_pt?(pdir(pt1'))
{-3}	not_present?(pdir(pt2'))
{-4}	Pdir(Pde_pt(pde_pt(pdir(pt1'))) WITH [accessed := TRUE])) = Pdir(Not_present)
{1}	Write?(access')
{2}	ptab?(pt2')

Applying decompose-equality,

This completes the proof of **set_reference_leaf.4**.

set_reference_leaf.5:

{-1}	pdir?(pt1')	=
{-2}	pde_pt?(pdir(pt1'))	
{-3}	Pdir(Pde_pt(pde_pt(pdir(pt1'))) WITH [accessed := TRUE])) = Pdir(Pde_4m(pde_4m(pdir(pt2'))) WITH [accessed := TRUE]))	
{1}	Write?(access')	
{2}	ptab?(pt2')	
{3}	not_present?(pdir(pt2'))	
{4}	pde_pt?(pdir(pt2'))	

Applying decompose-equality,

This completes the proof of set_reference_leaf.5.

set_reference_leaf.6:

{-1}	pdir?(pt1')	=
{-2}	Pdir(Pde_4m(pde_4m(pdir(pt1'))) WITH [accessed := TRUE])) = Pdir(Pde_pt(pde_pt(pdir(pt2'))) WITH [accessed := TRUE]))	
{-3}	pdir?(pt2')	
{-4}	pde_pt?(pdir(pt2'))	
{1}	Write?(access')	
{2}	not_present?(pdir(pt1'))	
{3}	pde_pt?(pdir(pt1'))	

Applying decompose-equality,

This completes the proof of set_reference_leaf.6.

Q.E.D.

C.160 Proofs for Pde_type (paging-data.pvs)

This theory contains no provable formal statements.

C.161 Proofs for Pde_type_adt (Pde_type_adt.pvs)

C.161.1 Pde_type_adt.Pde_4m_TCC1

Terse proof for Pde_4m_TCC1.

Pde_4m_TCC1:

{1}	$\forall (a: \text{Memory_Address_4G}): \text{offset}(a) \geq 0$
-----	--

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of Pde_4m_TCC1.

Q.E.D.

C.162 Proofs for Pde_type_adt_reduce (Pde_type_adt.pvs)

This theory contains no provable formal statements.

C.163 Proofs for Phy_Mem_Blessing (challenge-phymem.pvs)

C.163.1 Phy_Mem_Blessing.phy_mem_plain_memory

Terse proof for phy_mem_plain_memory.

phy_mem_plain_memory:

$$\frac{\{1\} \quad \forall (\text{oact}: \text{PRED}[[\text{Physical_memory} \rightarrow \text{SuperResult}[\text{Physical_memory}]]]):$$

$$\quad \text{unchanged_memory_invariant?}(\text{phy_mem}, \text{fullset}[\text{Physical_memory}], \text{oact}, \text{in_memory}(\text{min}, \text{max}))$$

$$\quad \supset \text{plain_memory?}(\text{phy_pm}(\text{oact}))$$

Repeatedly Skolemizing and flattening,

Expanding the definition of plain_memory?,

Expanding the definition of phy_pm,

Rewriting using union_empty_left, matching in *,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 7 subgoals:

phy_mem_plain_memory.1:

$$\frac{\{-1\} \quad \text{unchanged_memory_invariant?}(\text{phy_mem}, \text{fullset}[\text{Physical_memory}], \text{oact}', \text{in_memory}(\text{min}, \text{max}))$$

$$\{1\} \quad \text{side_effect_content_unchanged}(\text{in_memory}(\text{min}, \text{max}), \text{fullset}[\text{Physical_memory}],$$

$$\quad \text{memory_write_side_effect}(\text{phy_mem}))$$

Hiding formulas: (-1),

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of phy_mem_plain_memory.1.

phy_mem_plain_memory.2:

$$\frac{\{-1\} \quad \text{unchanged_memory_invariant?}(\text{phy_mem}, \text{fullset}[\text{Physical_memory}], \text{oact}', \text{in_memory}(\text{min}, \text{max}))$$

$$\{1\} \quad \text{side_effect_content_unchanged}(\text{in_memory}(\text{min}, \text{max}), \text{fullset}[\text{Physical_memory}],$$

$$\quad \text{memory_read_side_effect}(\text{phy_mem}))$$

Hiding formulas: (-1),

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of phy_mem_plain_memory.2.

phy_mem_plain_memory.3:

$$\frac{\{-1\} \quad \text{unchanged_memory_invariant?}(\text{phy_mem}, \text{fullset}[\text{Physical_memory}], \text{oact}', \text{in_memory}(\text{min}, \text{max}))$$

$$\{1\} \quad \text{transformers_ok?}(\text{fullset}[\text{Physical_memory}],$$

$$\quad ((\text{memory_read_transformers}(\text{phy_mem}, \text{in_memory}(\text{min}, \text{max})) \cup \text{memory_write_transf}))$$

Hiding formulas: -1,

Using lemma phy_mem_read_transformers_ok,

Using lemma phy_mem_write_transformers_ok,

Using lemma phy_mem_read_side_effects_transformers_ok,

Using lemma phy_mem_write_side_effects_transformers_ok,

Rewriting using transformers_ok_union_transformers, matching in *,

we get 2 subgoals:

C Proof scripts

phy_mem_plain_memory.3.1:

{-1}	transformers_ok?(fullset[Physical_memory[min, max]], memory_write_side_effect_super_transformers(phy_mem, in_memory(min, ma
{-2}	transformers_ok?(fullset[Physical_memory[min, max]], memory_read_side_effect_super_transformers(phy_mem, in_memory(min, ma
{-3}	transformers_ok?(fullset[Physical_memory[min, max]], memory_write_transformers(phy_mem, in_memory(min, max)))
{-4}	transformers_ok?(fullset[Physical_memory[min, max]], memory_read_transformers(phy_mem, in_memory(min, max)))
{1}	transformers_ok?(fullset[Physical_memory], (memory_read_transformers(phy_mem, in_memory(min, max)) ∪ memory_w
{2}	transformers_ok?(fullset[Physical_memory], ((memory_read_transformers(phy_mem, in_memory(min, max)) ∪ memory_w

Rewriting using transformers_ok_union_transformers, matching in *,

This completes the proof of phy_mem_plain_memory.3.1.

phy_mem_plain_memory.3.2:

{-1}	transformers_ok?(fullset[Physical_memory[min, max]], memory_write_side_effect_super_transformers(phy_mem, in_memory(min, m
{-2}	transformers_ok?(fullset[Physical_memory[min, max]], memory_read_side_effect_super_transformers(phy_mem, in_memory(min, ma
{-3}	transformers_ok?(fullset[Physical_memory[min, max]], memory_write_transformers(phy_mem, in_memory(min, max)))
{-4}	transformers_ok?(fullset[Physical_memory[min, max]], memory_read_transformers(phy_mem, in_memory(min, max)))
{1}	transformers_ok?(fullset[Physical_memory], (memory_read_side_effect_super_transformers(phy_mem, in_memory(min, m
{2}	transformers_ok?(fullset[Physical_memory], ((memory_read_transformers(phy_mem, in_memory(min, max)) ∪ memory_w

Rewriting using transformers_ok_union_transformers, matching in *,

This completes the proof of phy_mem_plain_memory.3.2.

phy_mem_plain_memory.4:

{-1}	unchanged_memory_invariant?(phy_mem, fullset[Physical_memory], oact', in_memory(min, ma
{1}	changed_memory_invariant?(phy_mem, fullset[Physical_memory], in_memory(min, max))

Rewriting using phy_mem_changed_memory_invariant, matching in *,

This completes the proof of phy_mem_plain_memory.4.

phy_mem_plain_memory.5:

{-1}	unchanged_memory_invariant?(phy_mem, fullset[Physical_memory], oact', in_memory(min, ma
{1}	unchanged_memory_write_invariant?(phy_mem, fullset[Physical_memory], in_memory(min, max)

Rewriting using phy_mem_unchanged_memory_write_invariant, matching in *,

This completes the proof of phy_mem_plain_memory.5.

phy_mem_plain_memory.6:

{-1}	unchanged_memory_invariant?(phy_mem, fullset[Physical_memory], oact', in_memory(min, ma
{1}	unchanged_memory_invariant?(phy_mem, fullset[Physical_memory], memory_write_transformers(phy_mem, in_memory(min, max)) emptyset[Address])

Hiding formulas: -1,

Expanding the definition of unchanged_memory_invariant?,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 2 subgoals:

`phy_mem_plain_memory.6.1:`

$$\begin{array}{|l}
 \hline
 \{1\} \quad \forall (s: \text{Physical_memory}[\text{min}, \text{max}], \\
 \quad q: [\text{Physical_memory}[\text{min}, \text{max}] \rightarrow \text{SuperResult}[\text{Physical_memory}[\text{min}, \text{max}]]], \\
 \quad a: \text{Address}): \\
 \text{fullset}[\text{Physical_memory}](s) \wedge \\
 \text{memory_write_transformers}(\text{phy_mem}, \text{in_memory}(\text{min}, \text{max}))(q) \wedge \\
 \text{emptyset}[\text{Address}](a) \wedge \\
 \text{OK?}(q(s)) \wedge \\
 \quad \text{OK?}(\text{memory_read}(\text{phy_mem})(a)(s)) \wedge \\
 \quad \text{OK?}(\text{memory_read}(\text{phy_mem})(a)(\text{state}(q(s)))) \\
 \supset \\
 \text{data}(\text{memory_read}(\text{phy_mem})(a)(\text{state}(q(s)))) = \\
 \text{data}(\text{memory_read}(\text{phy_mem})(a)(s))
 \end{array}$$

Repeatedly Skolemizing and flattening,

Expanding the definition of `emptyset`,

which is trivially true.

This completes the proof of `phy_mem_plain_memory.6.1`.

`phy_mem_plain_memory.6.2:`

$$\begin{array}{|l}
 \hline
 \{1\} \quad \text{transformer_invariant?}(\text{fullset}[\text{Physical_memory}], \\
 \quad \text{memory_write_transformers}(\text{phy_mem}, \text{in_memory}(\text{min}, \text{max})))
 \end{array}$$

Rewriting using `transformer_invariant_truth`, matching in `*`,

This completes the proof of `phy_mem_plain_memory.6.2`.

`phy_mem_plain_memory.7:`

$$\begin{array}{|l}
 \hline
 \{-1\} \quad \text{unchanged_memory_invariant?}(\text{phy_mem}, \text{fullset}[\text{Physical_memory}], \text{oact}', \text{in_memory}(\text{min}, \text{max})) \\
 \{1\} \quad \text{unchanged_memory_invariant?}(\text{phy_mem}, \text{fullset}[\text{Physical_memory}], \\
 \quad ((\text{oact}' \cup \text{memory_read_transformers}(\text{phy_mem}, \text{in_memory}(\text{min}, \text{max}))) \cup \\
 \quad \text{in_memory}(\text{min}, \text{max}))
 \end{array}$$

Using lemma `phy_mem_unchanged_memory_invariant_read_transformers`,

Using lemma `phy_mem_unchanged_memory_invariant_read_side_effect_transformers`,

Using lemma `phy_mem_unchanged_memory_invariant_write_side_effect_transformers`,

Rewriting using `unchanged_memory_invariant_union_transformers`, matching in `*`,

we get 2 subgoals:

C Proof scripts

phy_mem_plain_memory.7.1:

{-1}	unchanged_memory_invariant?(phy_mem, fullset[Physical_memory[min, max]], memory_write_side_effect_super_transformers(phy_mem, in_memory(min, max)))
{-2}	unchanged_memory_invariant?(phy_mem, fullset[Physical_memory[min, max]], memory_read_side_effect_super_transformers(phy_mem, in_memory(min, max)))
{-3}	unchanged_memory_invariant?(phy_mem, fullset[Physical_memory[min, max]], memory_read_transformers(phy_mem, in_memory(min, max)), in_memory(min, max))
{-4}	unchanged_memory_invariant?(phy_mem, fullset[Physical_memory], oact', in_memory(min, max))
{1}	unchanged_memory_invariant?(phy_mem, fullset[Physical_memory], (oact' ∪ memory_read_transformers(phy_mem, in_memory(min, max))))
{2}	unchanged_memory_invariant?(phy_mem, fullset[Physical_memory], ((oact' ∪ memory_read_transformers(phy_mem, in_memory(min, max))))

Rewriting using unchanged_memory_invariant_union_transformers, matching in *,

This completes the proof of phy_mem_plain_memory.7.1.

phy_mem_plain_memory.7.2:

{-1}	unchanged_memory_invariant?(phy_mem, fullset[Physical_memory[min, max]], memory_write_side_effect_super_transformers(phy_mem, in_memory(min, max)))
{-2}	unchanged_memory_invariant?(phy_mem, fullset[Physical_memory[min, max]], memory_read_side_effect_super_transformers(phy_mem, in_memory(min, max)))
{-3}	unchanged_memory_invariant?(phy_mem, fullset[Physical_memory[min, max]], memory_read_transformers(phy_mem, in_memory(min, max)), in_memory(min, max))
{-4}	unchanged_memory_invariant?(phy_mem, fullset[Physical_memory], oact', in_memory(min, max))
{1}	unchanged_memory_invariant?(phy_mem, fullset[Physical_memory], (memory_read_side_effect_super_transformers(phy_mem, in_memory(min, max))))
{2}	unchanged_memory_invariant?(phy_mem, fullset[Physical_memory], ((oact' ∪ memory_read_transformers(phy_mem, in_memory(min, max))))

Rewriting using unchanged_memory_invariant_union_transformers, matching in *,

This completes the proof of phy_mem_plain_memory.7.2.

Q.E.D.

C.164 Proofs for Phy_Mem_Blessing_Data (challenge-phymem.pvs)

C.164.1 Phy_Mem_Blessing_Data.in_memory_blessed_memory

Terse proof for `in_memory_blessed_memory`.

`in_memory_blessed_memory`:

$$\{1\} \quad \forall (\text{addr}: \text{Address}, \text{idt}: (\text{interpreted_data_type?}[\text{Data}])): \\ \text{in_memory?}(\text{idt}, \text{addr}, \text{min}, \text{max}) \supset \text{in_blessed_memory?}(\text{idt}, \text{addr}, \text{in_memory}(\text{min}, \text{max}))$$

Installing automatic rewrites from: `in_memory?` `in_blessed_memory?` `subset?` `member` `address_block`
`in_memory` + `<` `≤`

Repeatedly Skolemizing and flattening,

Simplifying, rewriting, and recording with decision procedures,

Repeatedly Skolemizing and flattening,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `in_memory_blessed_memory`.

Q.E.D.

C.165 Proofs for Phy_Mem_Challenge_Read_Other (challenge-phymem.pvs)

C.165.1 Phy_Mem_Challenge_Read_Other.read_write_other_ok

Terse proof for `read_write_other_ok`.

`read_write_other_ok`:

$$\{1\} \quad \forall (\text{addr1}, \text{addr2}: \text{Address}, \text{data1}: \text{Data1}, \text{dt1}: (\text{interpreted_data_type?}[\text{Data1}]), \\ \text{dt2}: (\text{interpreted_data_type?}[\text{Data2}]), \text{s}: \text{Physical_memory}[\text{min}, \text{max}]): \\ \text{in_memory?}(\text{dt1}, \text{addr1}, \text{min}, \text{max}) \wedge \\ \text{in_memory?}(\text{dt2}, \text{addr2}, \text{min}, \text{max}) \wedge \\ \text{blocks_disjoint?}(\text{addr1}, \text{size}(\text{uidt}(\text{dt1})), \text{addr2}, \text{size}(\text{uidt}(\text{dt2}))) \wedge \\ \text{valid_in_mem}(\text{phy_mem}, \text{dt2})(\text{addr2})(\text{s}) \\ \supset \\ \text{OK?}((\text{write_data}(\text{phy_mem}, \text{dt1})(\text{addr1}, \text{data1}) \#\# \text{read_data}(\text{phy_mem}, \text{dt2})(\text{addr2}))(\text{s}))$$

Repeatedly Skolemizing and flattening,

Installing automatic rewrites from: `unchanged_memory_invariant_empty` `emptyset_is_empty?`

Using lemma `phy_mem_plain_memory`,

Simplifying, rewriting, and recording with decision procedures,

Using lemma `plain_memory_read_write_other_ok`[`Physical_memory`, `Data1`, `Data2`],

Installing automatic rewrites from: `fullset!` `union_empty_left` `phy_pm` `read_data` `write_data`
`valid_in_mem`

Simplifying, rewriting, and recording with decision procedures,

Keeping (-4 -5 1) and hiding *,

Using lemma `in_memory_blessed_memory`[`Data1`, `min`, `max`],

Using lemma `in_memory_blessed_memory`[`Data2`, `min`, `max`],

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `read_write_other_ok`.

Q.E.D.

C.165.2

Phy_Mem_Challenge_Read_Other.read_write_other_res_TCC1

Terse proof for read_write_other_res_TCC1.

read_write_other_res_TCC1:

$$\begin{array}{l}
 \{1\} \quad \forall (\text{addr1}, \text{addr2}: \text{Address}, \text{data1}: \text{Data1}, \text{dt1}: (\text{interpreted_data_type?}[\text{Data1}]), \\
 \quad \text{dt2}: (\text{interpreted_data_type?}[\text{Data2}]), s: \text{Physical_memory}[\text{min}, \text{max}]): \\
 \text{valid_in_mem}(\text{phy_mem}, \text{dt2})(\text{addr2})(s) \wedge \\
 \text{blocks_disjoint?}(\text{addr1}, \text{size}(\text{uidt}(\text{dt1})), \text{addr2}, \text{size}(\text{uidt}(\text{dt2}))) \wedge \\
 \text{in_memory?}(\text{dt2}, \text{addr2}, \text{min}, \text{max}) \wedge \text{in_memory?}(\text{dt1}, \text{addr1}, \text{min}, \text{max}) \\
 \supset \\
 \text{OK?}[\text{Physical_memory}[\text{min}, \text{max}], \text{Data2}] \\
 \quad ((\#\#[\text{Physical_memory}[\text{min}, \text{max}], \text{Unit}, \text{Data2}] \\
 \quad \quad (\text{write_data}[\text{Physical_memory}[\text{min}, \text{max}], \text{Data1}] \\
 \quad \quad \quad (\text{phy_mem}[\text{min}, \text{max}], \text{dt1})(\text{addr1}, \text{data1}), \\
 \quad \quad \quad \text{read_data}[\text{Physical_memory}[\text{min}, \text{max}], \text{Data2}] \\
 \quad \quad \quad (\text{phy_mem}[\text{min}, \text{max}], \text{dt2})(\text{addr2}))) \\
 \quad (s))
 \end{array}$$

Repeatedly Skolemizing and flattening,

Rewriting using read_write_other_ok, matching in *,

This completes the proof of read_write_other_res_TCC1.

Q.E.D.

C.165.3

Phy_Mem_Challenge_Read_Other.read_write_other_res_TCC2

Terse proof for read_write_other_res_TCC2.

read_write_other_res_TCC2:

$$\begin{array}{l}
 \{1\} \quad \forall (\text{addr1}, \text{addr2}: \text{Address}, \text{dt1}: (\text{interpreted_data_type?}[\text{Data1}]), \\
 \quad \text{dt2}: (\text{interpreted_data_type?}[\text{Data2}]), s: \text{Physical_memory}[\text{min}, \text{max}]): \\
 \text{valid_in_mem}(\text{phy_mem}, \text{dt2})(\text{addr2})(s) \wedge \\
 \text{blocks_disjoint?}(\text{addr1}, \text{size}(\text{uidt}(\text{dt1})), \text{addr2}, \text{size}(\text{uidt}(\text{dt2}))) \wedge \\
 \text{in_memory?}(\text{dt2}, \text{addr2}, \text{min}, \text{max}) \wedge \text{in_memory?}(\text{dt1}, \text{addr1}, \text{min}, \text{max}) \\
 \supset \\
 \text{OK?}[\text{Physical_memory}[\text{min}, \text{max}], \text{Data2}] \\
 \quad (\text{read_data}[\text{Physical_memory}[\text{min}, \text{max}], \text{Data2}] \\
 \quad \quad (\text{phy_mem}[\text{min}, \text{max}], \text{dt2})(\text{addr2})(s))
 \end{array}$$

Repeatedly Skolemizing and flattening,

Installing automatic rewrites from: unchanged_memory_invariant_empty emptyset_is_empty?
union_empty_left phy_pm fullset in_memory_blessed_memory

Using lemma phy_mem_plain_memory,

Simplifying, rewriting, and recording with decision procedures,

Using lemma plain_memory_read_data_ok[Physical_memory, Data2],

Installing automatic rewrites from: (read_data! write_data! valid_in_mem!)

Simplifying, rewriting, and recording with decision procedures,
 Rewriting using `in_memory_blessed_memory[Data2, min, max]`, matching in *,
 This completes the proof of `read_write_other_res_TCC2`.
 Q.E.D.

C.165.4 *Phy_Mem_Challenge_Read_Other.read_write_other_res*

Terse proof for `read_write_other_res`.

`read_write_other_res`:

$\{1\} \quad \forall (\text{addr1}, \text{addr2}: \text{Address}, \text{data1}: \text{Data1}, \text{dt1}: (\text{interpreted_data_type?}[\text{Data1}]),$ $\quad \text{dt2}: (\text{interpreted_data_type?}[\text{Data2}]), s: \text{Physical_memory}[\text{min}, \text{max}]):$ $\text{in_memory?}(\text{dt1}, \text{addr1}, \text{min}, \text{max}) \wedge$ $\text{in_memory?}(\text{dt2}, \text{addr2}, \text{min}, \text{max}) \wedge$ $\text{blocks_disjoint?}(\text{addr1}, \text{size}(\text{uidt}(\text{dt1})), \text{addr2}, \text{size}(\text{uidt}(\text{dt2}))) \wedge$ $\text{valid_in_mem}(\text{phy_mem}, \text{dt2})(\text{addr2})(s)$ \supset $\text{data}(\text{write_data}(\text{phy_mem}, \text{dt1})(\text{addr1}, \text{data1}) \#\# \text{read_data}(\text{phy_mem}, \text{dt2})(\text{addr2}))$ $\quad (s)$ $= \text{data}(\text{read_data}(\text{phy_mem}, \text{dt2})(\text{addr2})(s))$

Repeatedly Skolemizing and flattening,
 Installing automatic rewrites from: `unchanged_memory_invariant_empty` `emptyset_is_empty?`
`union_empty_left` `fullset` `phy_pm`
 Using lemma `phy_mem_plain_memory`,
 Simplifying, rewriting, and recording with decision procedures,
 Using lemma `plain_memory_read_write_other_res[Physical_memory, Data1, Data2]`,
 Installing automatic rewrites from: `(read_data! write_data! valid_in_mem!)`
 Simplifying, rewriting, and recording with decision procedures,
 Rewriting using `in_memory_blessed_memory[Data2, min, max]`, matching in *,
 Rewriting using `in_memory_blessed_memory[Data1, min, max]`, matching in *,
 This completes the proof of `read_write_other_res`.
 Q.E.D.

C.165.5 *Phy_Mem_Challenge_Read_Other.read_read_ok*

Terse proof for `read_read_ok`.

`read_read_ok`:

$\{1\} \quad \forall (\text{addr1}, \text{addr2}: \text{Address}, \text{dt1}: (\text{interpreted_data_type?}[\text{Data1}]),$ $\quad \text{dt2}: (\text{interpreted_data_type?}[\text{Data2}]), s: \text{Physical_memory}[\text{min}, \text{max}]):$ $\text{in_memory?}(\text{dt1}, \text{addr1}, \text{min}, \text{max}) \wedge$ $\text{in_memory?}(\text{dt2}, \text{addr2}, \text{min}, \text{max}) \wedge$ $\text{valid_in_mem}(\text{phy_mem}, \text{dt1})(\text{addr1})(s) \wedge \text{valid_in_mem}(\text{phy_mem}, \text{dt2})(\text{addr2})(s)$ $\supset \text{OK?}((\text{read_data}(\text{phy_mem}, \text{dt1})(\text{addr1}) \#\# \text{read_data}(\text{phy_mem}, \text{dt2})(\text{addr2}))(s))$
--

Repeatedly Skolemizing and flattening,
 Installing automatic rewrites from: `unchanged_memory_invariant_empty` `emptyset_is_empty?`
`union_empty_left` `fullset` `phy_pm`
 Using lemma `phy_mem_plain_memory`,
 Simplifying, rewriting, and recording with decision procedures,
 Using lemma `plain_memory_read_read_ok[Physical_memory, Data1, Data2]`,

Simplifying, rewriting, and recording with decision procedures,
 Using lemma `in_memory_blessed_memory [Data2, min, max]`,
 Using lemma `in_memory_blessed_memory [Data1, min, max]`,
 Installing automatic rewrites from: (`read_data!` `write_data!` `valid_in_mem!`)
 Simplifying, rewriting, and recording with decision procedures,
 This completes the proof of `read_read_ok`.
 Q.E.D.

C.165.6 Phy_Mem_Challenge_Read_Other.read_read_TCC1

Terse proof for `read_read_TCC1`.

`read_read_TCC1`:

<pre>{1} ∃ (addr1, addr2: Address, dt1: (interpreted_data_type? [Data1]), dt2: (interpreted_data_type? [Data2]), s: Physical_memory [min, max]): valid_in_mem(phy_mem, dt2)(addr2)(s) ∧ valid_in_mem(phy_mem, dt1)(addr1)(s) ∧ in_memory?(dt2, addr2, min, max) ∧ in_memory?(dt1, addr1, min, max) ⊃ OK?[Physical_memory [min, max], Data2] ((##[Physical_memory [min, max], Data1, Data2] (read_data[Physical_memory [min, max], Data1] (phy_mem [min, max], dt1)(addr1), read_data[Physical_memory [min, max], Data2] (phy_mem [min, max], dt2)(addr2))) (s))</pre>
--

Repeatedly Skolemizing and flattening,
 Rewriting using `read_read_ok`, matching in *,
 This completes the proof of `read_read_TCC1`.
 Q.E.D.

C.165.7 Phy_Mem_Challenge_Read_Other.read_read_TCC2

Terse proof for `read_read_TCC2`.

`read_read_TCC2`:

<pre>{1} ∃ (addr1, addr2: Address, dt1: (interpreted_data_type? [Data1]), dt2: (interpreted_data_type? [Data2]), s: Physical_memory [min, max]): valid_in_mem(phy_mem, dt2)(addr2)(s) ∧ valid_in_mem(phy_mem, dt1)(addr1)(s) ∧ in_memory?(dt2, addr2, min, max) ∧ in_memory?(dt1, addr1, min, max) ⊃ OK?[Physical_memory [min, max], Data2] (read_data[Physical_memory [min, max], Data2] (phy_mem [min, max], dt2)(addr2)(s))</pre>

Repeatedly Skolemizing and flattening,
 Installing automatic rewrites from: `unchanged_memory_invariant_empty` `emptyset_is_empty?`
`union_empty_left` `fullset` `phy_pm`
 Using lemma `phy_mem_plain_memory`,

Simplifying, rewriting, and recording with decision procedures,
 Using lemma `plain_memory_read_data_ok`[`Physical_memory`, `Data2`],
 Installing automatic rewrites from: (`read_data!` `write_data!` `valid_in_mem!`)
 Simplifying, rewriting, and recording with decision procedures,
 Rewriting using `in_memory_blessed_memory`[`Data2`, `min`, `max`], matching in *,
 This completes the proof of `read_read_TCC2`.
 Q.E.D.

C.165.8 *Phy_Mem_Challenge_Read_Other.read_read*

Terse proof for `read_read`.

`read_read`:

$\{1\} \quad \forall (\text{addr1}, \text{addr2}: \text{Address}, \text{dt1}: (\text{interpreted_data_type?}[\text{Data1}]), \\ \text{dt2}: (\text{interpreted_data_type?}[\text{Data2}]), s: \text{Physical_memory}[\text{min}, \text{max}]): \\ \text{in_memory?}(\text{dt1}, \text{addr1}, \text{min}, \text{max}) \wedge \\ \text{in_memory?}(\text{dt2}, \text{addr2}, \text{min}, \text{max}) \wedge \\ \text{valid_in_mem}(\text{phy_mem}, \text{dt1})(\text{addr1})(s) \wedge \text{valid_in_mem}(\text{phy_mem}, \text{dt2})(\text{addr2})(s) \\ \supset \\ \text{data}(\text{read_data}(\text{phy_mem}, \text{dt1})(\text{addr1}) \text{##} \text{read_data}(\text{phy_mem}, \text{dt2})(\text{addr2}))(s) = \\ \text{data}(\text{read_data}(\text{phy_mem}, \text{dt2})(\text{addr2}))(s))$

Repeatedly Skolemizing and flattening,
 Installing automatic rewrites from: `unchanged_memory_invariant_empty` `emptyset_is_empty?`
`union_empty_left` `fullset` `phy_pm`
 Using lemma `phy_mem_plain_memory`,
 Simplifying, rewriting, and recording with decision procedures,
 Using lemma `plain_memory_read_read`[`Physical_memory`, `Data1`, `Data2`],
 Installing automatic rewrites from: (`read_data!` `write_data!` `valid_in_mem!`)
 Simplifying, rewriting, and recording with decision procedures,
 Rewriting using `in_memory_blessed_memory`[`Data1`, `min`, `max`], matching in *,
 Rewriting using `in_memory_blessed_memory`[`Data2`, `min`, `max`], matching in *,
 This completes the proof of `read_read`.
 Q.E.D.

C.166 Proofs for *Phy_Mem_Challenge_Read_Same* (*challenge-phymem.pvs*)

C.166.1 *Phy_Mem_Challenge_Read_Same.read_write_ok*

Terse proof for `read_write_ok`.

`read_write_ok`:

$\{1\} \quad \forall (\text{addr}: \text{Address}, \text{data}: \text{Data}, \text{dt}: (\text{interpreted_data_type?}[\text{Data}]), \\ s: \text{Physical_memory}[\text{min}, \text{max}]): \\ \text{in_memory?}(\text{dt}, \text{addr}, \text{min}, \text{max}) \supset \\ \text{OK?}(\text{write_data}(\text{phy_mem}, \text{dt})(\text{addr}, \text{data}) \text{##} \text{read_data}(\text{phy_mem}, \text{dt})(\text{addr}))(s))$

Repeatedly Skolemizing and flattening,
 Installing automatic rewrites from: `unchanged_memory_invariant_empty` `emptyset_is_empty?`

Using lemma `phy_mem_plain_memory`,
 Simplifying, rewriting, and recording with decision procedures,
 Using lemma `plain_memory_read_write_ok` [`Physical_memory`, `Data`],
 Simplifying, rewriting, and recording with decision procedures,
 Hiding formulas: -2,
 Expanding the definition of `phy_pm`,
 Using lemma `in_memory_blessed_memory`,
 Simplifying, rewriting, and recording with decision procedures,
 Expanding the definition of `fullset`,
 Expanding the definition of `write_data`,
 Expanding the definition of `read_data`,
 which is trivially true.
 This completes the proof of `read_write_ok`.
 Q.E.D.

C.166.2 `Phy_Mem_Challenge_Read_Same.read_write_res_TCC1`

Terse proof for `read_write_res_TCC1`.

`read_write_res_TCC1`:

```
{1}  ∃ (addr: Address, data: Data, dt: (interpreted_data_type?[Data]),
      s: Physical_memory [min, max]):
  in_memory?(dt, addr, min, max) ⊃
  OK?[Physical_memory [min, max], Data]
  ((##[Physical_memory [min, max], Unit, Data]
    (write_data[Physical_memory [min, max], Data]
      (phy_mem [min, max], dt)(addr, data),
      read_data[Physical_memory [min, max], Data]
        (phy_mem [min, max], dt)(addr))))
  (s)
```

Repeatedly Skolemizing and flattening,
 Rewriting using `read_write_ok`, matching in *,
 This completes the proof of `read_write_res_TCC1`.
 Q.E.D.

C.166.3 `Phy_Mem_Challenge_Read_Same.read_write_res`

Terse proof for `read_write_res`.

`read_write_res`:

```
{1}  ∃ (addr: Address, data1: Data, dt: (interpreted_data_type?[Data]),
      s: Physical_memory [min, max]):
  in_memory?(dt, addr, min, max) ⊃
  data((write_data(phy_mem, dt)(addr, data1) ## read_data(phy_mem, dt)(addr))(s))
  = data1
```

Repeatedly Skolemizing and flattening,
 Installing automatic rewrites from: `unchanged_memory_invariant_empty` `emptyset_is_empty?`
 Using lemma `phy_mem_plain_memory`,
 Simplifying, rewriting, and recording with decision procedures,
 Using lemma `plain_memory_read_write_res` [`Physical_memory`, `Data`],

Installing automatic rewrites from: phy_pm fullset in_memory_blessed_memory read_data write_data
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of read_write_res.
Q.E.D.

C.167 Proofs for Phy_Mem_Properties (challenge-phymem.pvs)

C.167.1 Phy_Mem_Properties.min_TCC1

Terse proof for min_TCC1.

min_TCC1:

{1} $\exists (x: \text{Memory_Address}): \text{TRUE}$

Instantiating the top quantifier in 1 with the terms: Mem(0),
Expanding the definition of Mem,
which is trivially true.
This completes the proof of min_TCC1.
Q.E.D.

C.167.2 Phy_Mem_Properties.phy_mem_write_transformers_ok

Terse proof for phy_mem_write_transformers_ok.

phy_mem_write_transformers_ok:

{1} transformers_ok?(fullset[Physical_memory],
memory_write_transformers(phy_mem, in_memory(min, max)))

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of phy_mem_write_transformers_ok.
Q.E.D.

C.167.3 Phy_Mem_Properties.phy_mem_read_transformers_ok

Terse proof for phy_mem_read_transformers_ok.

phy_mem_read_transformers_ok:

{1} transformers_ok?(fullset[Physical_memory],
memory_read_transformers(phy_mem, in_memory(min, max)))

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of phy_mem_read_transformers_ok.
Q.E.D.

C.167.4

Phy_Mem_Properties.phy_mem_read_side_effects_transformers_ok

Terse proof for phy_mem_read_side_effects_transformers_ok.

phy_mem_read_side_effects_transformers_ok:

{1}	transformers_ok?(fullset [Physical_memory], memory_read_side_effect_super_transformers(phy_mem, in_memory(min, ma
-----	--

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of phy_mem_read_side_effects_transformers_ok.

Q.E.D.

C.167.5

Phy_Mem_Properties.phy_mem_write_side_effects_transformers_ok

Terse proof for phy_mem_write_side_effects_transformers_ok.

phy_mem_write_side_effects_transformers_ok:

{1}	transformers_ok?(fullset [Physical_memory], memory_write_side_effect_super_transformers(phy_mem, in_memory(min, ma
-----	---

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of phy_mem_write_side_effects_transformers_ok.

Q.E.D.

C.167.6

Phy_Mem_Properties.phy_mem_unchanged_memory_invariant_read_transformers

Terse proof for phy_mem_unchanged_memory_invariant_read_transformers.

phy_mem_unchanged_memory_invariant_read_transformers:

{1}	unchanged_memory_invariant?(phy_mem, fullset [Physical_memory], memory_read_transformers(phy_mem, in_memory(min, max)), in_memory(min, max))
-----	--

Expanding the definition of unchanged_memory_invariant?,

Applying propositional simplification,

we get 2 subgoals:

phy_mem_unchanged_memory_invariant_read_transformers.1:

{1}	transformer_invariant?(fullset [Physical_memory], memory_read_transformers(phy_mem, in_memory(min, max)))
-----	--

Rewriting using transformer_invariant_truth, matching in *,

This completes the proof of phy_mem_unchanged_memory_invariant_read_transformers.1.

phy_mem_unchanged_memory_invariant_read_transformers.2:

$\{1\} \quad \forall (s: \text{Physical_memory}[\text{min}, \text{max}],$ $q: [\text{Physical_memory}[\text{min}, \text{max}] \rightarrow \text{SuperResult}[\text{Physical_memory}[\text{min}, \text{max}]]) ,$ $a: \text{Address}):$ $\text{fullset}[\text{Physical_memory}](s) \wedge$ $\text{memory_read_transformers}(\text{phy_mem}, \text{in_memory}(\text{min}, \text{max}))(q) \wedge$ $\text{in_memory}(\text{min}, \text{max})(a) \wedge$ $\text{OK?}(q(s)) \wedge$ $\text{OK?}(\text{memory_read}(\text{phy_mem})(a)(s)) \wedge$ $\text{OK?}(\text{memory_read}(\text{phy_mem})(a)(\text{state}(q(s))))$ \supset $\text{data}(\text{memory_read}(\text{phy_mem})(a)(\text{state}(q(s)))) =$ $\text{data}(\text{memory_read}(\text{phy_mem})(a)(s))$
--

Expanding the definition of memory_read_transformers,

Repeatedly Skolemizing and flattening,

Replacing using formula -3,

Keeping (-2 -4 1) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of phy_mem_unchanged_memory_invariant_read_transformers.2.

Q.E.D.

C.167.7

Phy_Mem_Properties.phy_mem_unchanged_memory_invariant_read_side_effect_trans

Terse proof for phy_mem_unchanged_memory_invariant_read_side_effect_transformers.

phy_mem_unchanged_memory_invariant_read_side_effect_transformers:

$\{1\} \quad \text{unchanged_memory_invariant?}(\text{phy_mem}, \text{fullset}[\text{Physical_memory}],$ $\text{memory_read_side_effect_super_transformers}(\text{phy_mem},$ $\text{in_memory}(\text{min}, \text{max}))$ $\text{in_memory}(\text{min}, \text{max}))$
--

Expanding the definition of unchanged_memory_invariant?,

Applying propositional simplification,

we get 2 subgoals:

phy_mem_unchanged_memory_invariant_read_side_effect_transformers.1:

$\{1\} \quad \text{transformer_invariant?}(\text{fullset}[\text{Physical_memory}],$ $\text{memory_read_side_effect_super_transformers}(\text{phy_mem},$ $\text{in_memory}(\text{min}, \text{max}))$
--

Rewriting using transformer_invariant_truth, matching in *,

This completes the proof of phy_mem_unchanged_memory_invariant_read_side_effect_transformers.1.

phy_mem_unchanged_memory_invariant_read_side_effect_transformers.2:

$\{1\} \quad \forall (s: \text{Physical_memory}[\text{min}, \text{max}],$ $q: [\text{Physical_memory}[\text{min}, \text{max}] \rightarrow \text{SuperResult}[\text{Physical_memory}[\text{min}, \text{max}]]],$ $a: \text{Address}):$ $\text{fullset}[\text{Physical_memory}](s) \wedge$ $\text{memory_read_side_effect_super_transformers}(\text{phy_mem}, \text{in_memory}(\text{min}, \text{max}))(q) \wedge$ $\text{in_memory}(\text{min}, \text{max})(a) \wedge$ $\text{OK?}(q(s)) \wedge$ $\text{OK?}(\text{memory_read}(\text{phy_mem})(a)(s)) \wedge$ $\text{OK?}(\text{memory_read}(\text{phy_mem})(a)(\text{state}(q(s))))$ \supset $\text{data}(\text{memory_read}(\text{phy_mem})(a)(\text{state}(q(s)))) =$ $\text{data}(\text{memory_read}(\text{phy_mem})(a)(s))$
--

Expanding the definition of memory_read_side_effect_super_transformers,

Repeatedly Skolemizing and flattening,

Replacing using formula -4,

Keeping (-2 -3 1) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of phy_mem_unchanged_memory_invariant_read_side_effect_transformers.2.

Q.E.D.

C.167.8

Phy_Mem_Properties.phy_mem_unchanged_memory_invariant_write_side_eff

Terse proof for phy_mem_unchanged_memory_invariant_write_side_effect_transformers.

phy_mem_unchanged_memory_invariant_write_side_effect_transformers:

$\{1\} \quad \text{unchanged_memory_invariant?}(\text{phy_mem}, \text{fullset}[\text{Physical_memory}],$ $\text{memory_write_side_effect_super_transformers}(\text{phy_mem},$ $\text{in_memory}(\text{min}, \text{max}))$
--

Expanding the definition of unchanged_memory_invariant?,

Applying propositional simplification,

we get 2 subgoals:

phy_mem_unchanged_memory_invariant_write_side_effect_transformers.1:

$\{1\} \quad \text{transformer_invariant?}(\text{fullset}[\text{Physical_memory}],$ $\text{memory_write_side_effect_super_transformers}(\text{phy_mem},$ $\text{in_memory}(\text{min},$
--

Rewriting using transformer_invariant_truth, matching in *,

This completes the proof of phy_mem_unchanged_memory_invariant_write_side_effect_transformers.1.

phy_mem_unchanged_memory_invariant_write_side_effect_transformers.2:

$\{1\} \quad \forall (s: \text{Physical_memory}[\text{min}, \text{max}],$ $q: [\text{Physical_memory}[\text{min}, \text{max}] \rightarrow \text{SuperResult}[\text{Physical_memory}[\text{min}, \text{max}]]],$ $a: \text{Address}):$ $\text{fullset}[\text{Physical_memory}](s) \wedge$ $\text{memory_write_side_effect_super_transformers}(\text{phy_mem}, \text{in_memory}(\text{min}, \text{max}))(q) \wedge$ $\text{in_memory}(\text{min}, \text{max})(a) \wedge$ $\text{OK?}(q(s)) \wedge$ $\text{OK?}(\text{memory_read}(\text{phy_mem})(a)(s)) \wedge$ $\text{OK?}(\text{memory_read}(\text{phy_mem})(a)(\text{state}(q(s))))$ \supset $\text{data}(\text{memory_read}(\text{phy_mem})(a)(\text{state}(q(s)))) =$ $\text{data}(\text{memory_read}(\text{phy_mem})(a)(s))$

Expanding the definition of memory_write_side_effect_super_transformers,

Repeatedly Skolemizing and flattening,

Replacing using formula -4,

Keeping (-2 -3 1) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of phy_mem_unchanged_memory_invariant_write_side_effect_transformers.2.

Q.E.D.

C.167.9

Phy_Mem_Properties.phy_mem_unchanged_memory_write_invariant

Terse proof for phy_mem_unchanged_memory_write_invariant.

phy_mem_unchanged_memory_write_invariant:

$\{1\} \quad \text{unchanged_memory_write_invariant?}(\text{phy_mem}, \text{fullset}[\text{Physical_memory}], \text{in_memory}(\text{min}, \text{max}))$
--

Expanding the definition of unchanged_memory_write_invariant?,

Expanding the definition of unchanged_memory_invariant?,

Expanding the definition of singleton,

Repeatedly Skolemizing and flattening,

Applying propositional simplification,

we get 2 subgoals:

phy_mem_unchanged_memory_write_invariant.1:

$\{-1\} \quad b' < \text{max_byte}$ $\{-2\} \quad \text{in_memory}(\text{min}, \text{max})(\text{waddr}')$ <hr/> $\{1\} \quad \text{transformer_invariant?}(\text{fullset}[\text{Physical_memory}],$ $\{y \mid y = \text{expr_2_super}(\text{memory_write}(\text{phy_mem})(\text{waddr}', b'))\})$
--

Rewriting using transformer_invariant_truth, matching in *,

This completes the proof of phy_mem_unchanged_memory_write_invariant.1.

phy_mem_unchanged_memory_write_invariant.2:

<pre> {-1} b' < max_byte {-2} in_memory(min, max)(waddr') </pre>
<pre> {1} ∀ (s: Physical_memory[min, max], q: [Physical_memory[min, max] → SuperResult[Physical_memory[min, max]]], a: Address): fullset[Physical_memory](s) ∧ q = expr_2_super(memory_write(phy_mem)(waddr', b')) ∧ remove(waddr', in_memory(min, max))(a) ∧ OK?(q(s)) ∧ OK?(memory_read(phy_mem)(a)(s)) ∧ OK?(memory_read(phy_mem)(a)(state(q(s)))) ⊃ data(memory_read(phy_mem)(a)(state(q(s)))) = data(memory_read(phy_mem)(a)(s)) </pre>

Repeatedly Skolemizing and flattening,

Replacing using formula -2,

Keeping (-3 -8 1) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of phy_mem_unchanged_memory_write_invariant.2.

Q.E.D.

C.167.10

Phy_Mem_Properties.phy_mem_changed_memory_invariant

Terse proof for phy_mem_changed_memory_invariant.

phy_mem_changed_memory_invariant:

<pre> {1} changed_memory_invariant?(phy_mem, fullset[Physical_memory], in_memory(min, max)) </pre>

Expanding the definition of changed_memory_invariant?,

Rewriting using transformer_invariant_truth, matching in *,

Rewriting using transformer_invariant_truth, matching in *,

Repeatedly Skolemizing and flattening,

Keeping (-3 1) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of phy_mem_changed_memory_invariant.

Q.E.D.

C.168 Proofs for Physical_Memory (physical_memory.pvs)

This theory contains no provable formal statements.

C.169 Proofs for Physical_Memory_Corollaries (physical_memory.pvs)

C.169.1

Physical_Memory_Corollaries.physical_read_ok_next_state_TCC1

Terse proof for physical_read_ok_next_state_TCC1.

physical_read_ok_next_state_TCC1:

$\{1\} \quad \forall (a: \text{Address}, s: \text{Physical_memory}[\text{min}, \text{max}]):$ $\text{OK?}(\text{physical_read}(a)(s)) \supset$ $\text{OK?}[\text{Physical_memory}[\text{min}, \text{max}], \text{Byte}](\text{physical_read}[\text{min}, \text{max}](a)(s)) \vee$ $\text{Exception?}[\text{Physical_memory}[\text{min}, \text{max}], \text{Byte}](\text{physical_read}[\text{min}, \text{max}](a)(s))$
--

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of physical_read_ok_next_state_TCC1.

Q.E.D.

C.169.2 Physical_Memory_Corollaries.physical_read_ok_next_state

Terse proof for physical_read_ok_next_state.

physical_read_ok_next_state:

$\{1\} \quad \forall (a: \text{Address}, s: \text{Physical_memory}[\text{min}, \text{max}]):$ $\text{OK?}(\text{physical_read}(a)(s)) \supset \text{state}(\text{physical_read}(a)(s)) = s$
--

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of physical_read_ok_next_state.

Q.E.D.

C.169.3

Physical_Memory_Corollaries.physical_write_ok_next_state_TCC1

Terse proof for physical_write_ok_next_state_TCC1.

physical_write_ok_next_state_TCC1:

$\{1\} \quad \forall (a: \text{Address}, b: \text{Byte}, s: \text{Physical_memory}[\text{min}, \text{max}]):$ $\text{OK?}(\text{physical_write}(a, b)(s)) \supset$ $\text{OK?}[\text{Physical_memory}[\text{min}, \text{max}], \text{Unit}](\text{physical_write}[\text{min}, \text{max}](a, b)(s)) \vee$ $\text{Exception?}[\text{Physical_memory}[\text{min}, \text{max}], \text{Unit}](\text{physical_write}[\text{min}, \text{max}](a, b)(s))$
--

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of physical_write_ok_next_state_TCC1.

Q.E.D.

C.169.4 Physical_Memory_Corollaries.physical_write_ok_next_state

Terse proof for physical_write_ok_next_state.

physical_write_ok_next_state:

$$\{1\} \quad \forall (a: \text{Address}, b: \text{Byte}, s: \text{Physical_memory}[\text{min}, \text{max}]):$$
$$\text{OK?}(\text{physical_write}(a, b)(s)) \supset$$
$$\text{state}(\text{physical_write}(a, b)(s)) = s \text{ WITH } [(a) := b]$$

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `physical_write_ok_next_state`.
Q.E.D.

C.169.5 Physical_Memory_Corollaries.physical_read_ok_get_data

Terse proof for `physical_read_ok_get_data`.

physical_read_ok_get_data:

$$\{1\} \quad \forall (a: \text{Address}, s: \text{Physical_memory}[\text{min}, \text{max}]):$$
$$\text{OK?}(\text{physical_read}(a)(s)) \supset \text{data}(\text{physical_read}(a)(s)) = s(a)$$

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `physical_read_ok_get_data`.
Q.E.D.

C.170 Proofs for Physical_Memory_Properties (physical_memory.pvs)

C.170.1 Physical_Memory_Properties.physical_read_ok

Terse proof for `physical_read_ok`.

physical_read_ok:

$$\{1\} \quad \forall (a: \text{Address}, s: \text{Physical_memory}[\text{min}, \text{max}]):$$
$$\text{in_memory}(\text{min}, \text{max})(a) \supset \text{OK?}(\text{physical_read}(a)(s))$$

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `physical_read_ok`.
Q.E.D.

C.170.2 Physical_Memory_Properties.physical_write_ok

Terse proof for `physical_write_ok`.

physical_write_ok:

$$\{1\} \quad \forall (a: \text{Address}, b: \text{Byte}, s: \text{Physical_memory}[\text{min}, \text{max}]):$$
$$\text{in_memory}(\text{min}, \text{max})(a) \supset \text{OK?}(\text{physical_write}(a, b)(s))$$

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `physical_write_ok`.
Q.E.D.

C.171 Proofs for Plain_Mem_Properties (plain_memory.pvs)

C.171.1 Plain_Mem_Properties.plain_mem_write_list_read_list_TCC1

Terse proof for plain_mem_write_list_read_list_TCC1.

plain_mem_write_list_read_list_TCC1:

$\{1\} \quad \forall (\text{addr: Address, pm: Plain_Memory}[\text{State}], s: \text{State, size: nat, bl: list}[\text{Byte}]):$ $\text{OK?}(\text{memory_write_list}(\text{pm}'\text{mem})(\text{addr}, \text{bl})(s)) \wedge$ $(\text{address_block}(\text{addr}, \text{size}) \subseteq \text{pm}'\text{rw_addr}) \wedge$ $\text{pm}'\text{states}(s) \wedge \text{plain_memory?}(\text{pm}) \wedge \text{size} = \text{length}(\text{bl})$ \supset $\text{OK?}[\text{State}, \text{Unit}](\text{memory_write_list}[\text{State}](\text{pm}'\text{mem})(\text{addr}, \text{bl})(s)) \vee$ $\text{Exception?}[\text{State}, \text{Unit}](\text{memory_write_list}[\text{State}](\text{pm}'\text{mem})(\text{addr}, \text{bl})(s))$
--

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of plain_mem_write_list_read_list_TCC1.

Q.E.D.

C.171.2 Plain_Mem_Properties.plain_mem_write_list_read_list

Terse proof for plain_mem_write_list_read_list.

plain_mem_write_list_read_list:

$\{1\} \quad \forall (\text{addr: Address, pm: Plain_Memory}[\text{State}], s: \text{State, size: nat, bl: list}[\text{Byte}]):$ $\text{size} = \text{length}(\text{bl}) \wedge$ $\text{plain_memory?}(\text{pm}) \wedge$ $\text{pm}'\text{states}(s) \wedge$ $(\text{address_block}(\text{addr}, \text{size}) \subseteq \text{pm}'\text{rw_addr}) \wedge$ $\text{OK?}(\text{memory_write_list}(\text{pm}'\text{mem})(\text{addr}, \text{bl})(s)) \wedge$ $\text{OK?}(\text{memory_read_list}(\text{pm}'\text{mem})(\text{addr}, \text{size})$ $\quad \quad \quad (\text{state}(\text{memory_write_list}(\text{pm}'\text{mem})(\text{addr}, \text{bl})(s))))$ \supset $\text{data}(\text{memory_read_list}(\text{pm}'\text{mem})(\text{addr}, \text{size})$ $\quad \quad \quad (\text{state}(\text{memory_write_list}(\text{pm}'\text{mem})(\text{addr}, \text{bl})(s))))$ $= \text{bl}$
--

Repeatedly Skolemizing and flattening,

Installing automatic rewrites from: subset_bigger_union_right

Using lemma unchanged_memory_read_list_write_list,

Replacing using formula -2,

Installing automatic rewrites from: plain_memory_transformers_ok_write_block transformers_ok_union_transformers plain_memory_transformers_ok_write_side_effects_block plain_memory_transformers_ok_read_block plain_memory_transformers_ok_read_side_effects_block plain_memory_side_effect_content_unchanged_read_block plain_memory_side_effect_content_unchanged_write_block plain_memory_changed_block plain_memory_unchanged_block plain_memory_unchanged_invariant

Installing automatic rewrites from: unchanged_memory_invariant_union_transformers plain_memory_unchanged_read_block plain_memory_unchanged_read_side_effect_block plain_memory_unchanged_write_side_effect_block

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of plain_mem_write_list_read_list.

Q.E.D.

C.171.3 Plain_Mem_Properties.plain_mem_q_read_list_TCC1

Terse proof for `plain_mem_q_read_list_TCC1`.

`plain_mem_q_read_list_TCC1`:

$$\{1\} \quad \forall (\text{addr}: \text{Address}, \text{pm}: \text{Plain_Memory}[\text{State}], s: \text{State}, \text{size}: \text{nat}, \\ q: [\text{State} \rightarrow \text{SuperResult}[\text{State}]]) : \\ \text{OK?}(q(s)) \wedge \text{OK?}(\text{memory_read_list}(\text{pm}'\text{mem})(\text{addr}, \text{size})(s)) \supset \\ \text{OK?}[\text{State}](q(s)) \vee \text{abnormal?}[\text{State}](q(s))$$

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `plain_mem_q_read_list_TCC1`.
Q.E.D.

C.171.4 Plain_Mem_Properties.plain_mem_q_read_list

Terse proof for `plain_mem_q_read_list`.

`plain_mem_q_read_list`:

$$\{1\} \quad \forall (\text{addr}: \text{Address}, \text{pm}: \text{Plain_Memory}[\text{State}], s: \text{State}, \text{size}: \text{nat}, \\ q: [\text{State} \rightarrow \text{SuperResult}[\text{State}]]) : \\ \text{OK?}(\text{memory_read_list}(\text{pm}'\text{mem})(\text{addr}, \text{size})(s)) \wedge \\ \text{OK?}(q(s)) \wedge \\ \text{OK?}(\text{memory_read_list}(\text{pm}'\text{mem})(\text{addr}, \text{size})(\text{state}(q(s)))) \wedge \\ \text{plain_memory?}(\text{pm}) \wedge \\ \text{unchanged_memory_invariant?}(\text{pm}'\text{mem}, \text{pm}'\text{states}, \text{singleton}(q), \\ \text{address_block}(\text{addr}, \text{size})) \\ \wedge \text{pm}'\text{states}(s) \wedge (\text{address_block}(\text{addr}, \text{size}) \subseteq (\text{pm}'\text{ro_addr} \cup \text{pm}'\text{rw_addr})) \\ \supset \\ \text{data}(\text{memory_read_list}(\text{pm}'\text{mem})(\text{addr}, \text{size})(\text{state}(q(s)))) = \\ \text{data}(\text{memory_read_list}(\text{pm}'\text{mem})(\text{addr}, \text{size})(s))$$

Repeatedly Skolemizing and flattening,
Installing automatic rewrites from: `union_subset1 subset_reflexive`
Using lemma `unchanged_memory_read_list[State]`,
Installing automatic rewrites from: `transformers_ok_union_transformers plain_memory_side_effect_content_uncha`
`plain_memory_transformers_ok_read_block plain_memory_transformers_ok_read_side_effects_block un-`
`changed_memory_invariant_union_transformers plain_memory_unchanged_read_block plain_memory_unchanged_rea`
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `plain_mem_q_read_list`.
Q.E.D.

C.172 Proofs for Plain_Mem_Properties_2 (plain_memory.pvs)

C.172.1

Plain_Mem_Properties_2.plain_memory_inv_pred_write_data_TCC1

Terse proof for `plain_memory_inv_pred_write_data_TCC1`.

plain_memory_inv_pred_write_data_TCC1:

<pre>{1} ∃ (addr: Address, data: Data, dt: (interpreted_data_type?[Data]), pm: Plain_Memory[State], s: State): OK?(write_data(pm, dt)(addr, data)(s)) ∧ (address_block(addr, size(uidt(dt))) ⊆ pm'rw_addr) ∧ pm'states(s) ∧ plain_memory?(pm) ⊃ OK?[State, Unit](write_data[State, Data](pm, dt)(addr, data)(s)) ∨ Exception?[State, Unit](write_data[State, Data](pm, dt)(addr, data)(s))</pre>

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of plain_memory_inv_pred_write_data_TCC1.
Q.E.D.

C.172.2 Plain_Mem_Properties_2.plain_memory_inv_pred_write_data

The L^AT_EX code for this proof is broken.

C.172.3 Plain_Mem_Properties_2.plain_memory_read_data_ok

Terse proof for plain_memory_read_data_ok.

plain_memory_read_data_ok:

<pre>{1} ∃ (addr: Address, dt: (interpreted_data_type?[Data]), pm: Plain_Memory[State], s: State): plain_memory?(pm) ∧ pm'states(s) ∧ in_blessed_memory?(dt, addr, (pm'ro_addr ∪ pm'rw_addr)) ∧ valid_in_mem(pm, dt)(addr)(s) ⊃ OK?(read_data(pm, dt)(addr)(s))</pre>
--

Installing automatic rewrites from: (## read_data)
Repeatedly Skolemizing and flattening,
Simplifying, rewriting, and recording with decision procedures,
Case splitting on OK?(memory_read_list(pm!1'mem)(addr!1, size(uidt(dt!1)))(s!1)),
we get 2 subgoals:

plain_memory_read_data_ok.1:

<pre>{-1} OK?(memory_read_list(pm'mem)(addr', size(uidt(dt')))(s')) {-2} interpreted_data_type?[Data](dt') {-3} plain_memory?(pm') {-4} pm'states(s') {-5} in_blessed_memory?(dt', addr', (pm'ro_addr ∪ pm'rw_addr)) {-6} valid_in_mem(pm', dt')(addr')(s')</pre>
<pre>{1} OK?(CASES memory_read_list(pm'mem)(addr', size(uidt(dt')))(s') OF OK(state, data): ok_lift(from_byte(dt')(data, addr'))(state), Exception(ex_type, state): Exception(ex_type, state), Fatal: Fatal, Hang: Hang ENDCASES)</pre>

Simplifying, rewriting, and recording with decision procedures,
Rewriting using ok_lift_ok, matching in *,

C Proof scripts

we get 2 subgoals:

`plain_memory_read_data_ok.1.1:`

{-1}	OK?(memory_read_list(pm' mem)(addr', size(uidt(dt')))(s'))
{-2}	interpreted_data_type?[Data](dt')
{-3}	plain_memory?(pm')
{-4}	pm' states(s')
{-5}	in_blessed_memory?(dt', addr', (pm' ro_addr ∪ pm' rw_addr))
{-6}	valid_in_mem(pm', dt')(addr')(s')
{1}	OK?(OK(state(memory_read_list(pm' mem)(addr', size(uidt(dt')))(s')), down(from_byte(dt') (data(memory_read_list(pm' mem)(addr', size(uidt(dt')))(s')), addr'))))

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `plain_memory_read_data_ok.1.1`.

`plain_memory_read_data_ok.1.2:`

{-1}	OK?(memory_read_list(pm' mem)(addr', size(uidt(dt')))(s'))
{-2}	interpreted_data_type?[Data](dt')
{-3}	plain_memory?(pm')
{-4}	pm' states(s')
{-5}	in_blessed_memory?(dt', addr', (pm' ro_addr ∪ pm' rw_addr))
{-6}	valid_in_mem(pm', dt')(addr')(s')
{1}	up?(from_byte(dt') (data(memory_read_list(pm' mem)(addr', size(uidt(dt')))(s')), addr'))
{2}	OK?(ok_lift(from_byte(dt') (data(memory_read_list(pm' mem)(addr', size(uidt(dt')))(s')), addr')) (state(memory_read_list(pm' mem)(addr', size(uidt(dt')))(s'))))

Rewriting using `valid_from_byte`, matching in `*`,

Expanding the definition of `valid_in_mem`,

Expanding the definition of `valid_in_mem`,

which is trivially true.

This completes the proof of `plain_memory_read_data_ok.1.2`.

`plain_memory_read_data_ok.2:`

{-1}	interpreted_data_type?[Data](dt')
{-2}	plain_memory?(pm')
{-3}	pm' states(s')
{-4}	in_blessed_memory?(dt', addr', (pm' ro_addr ∪ pm' rw_addr))
{-5}	valid_in_mem(pm', dt')(addr')(s')
{1}	OK?(memory_read_list(pm' mem)(addr', size(uidt(dt')))(s'))
{2}	OK?(CASES memory_read_list(pm' mem)(addr', size(uidt(dt')))(s') OF OK(state, data): ok_lift(from_byte(dt')(data, addr'))(state), Exception(ex_type, state): Exception(ex_type, state), Fatal: Fatal, Hang: Hang ENDCASES)

Expanding the definition of `in_blessed_memory?`,

Rewriting using `memory_read_list_ok`, matching in `*` where states gets `pm!1'states`,

we get 2 subgoals:

plain_memory_read_data_ok.2.1:

<pre> {-1} interpreted_data_type?[Data](dt') {-2} plain_memory?(pm') {-3} pm'`states(s') {-4} (address_block(addr', size(uidt(dt')))) ⊆ (pm'`ro_addr ∪ pm'`rw_addr) {-5} valid_in_mem(pm', dt')(addr')(s') </pre>	<pre> {1} transformer_invariant?(pm'`states, (memory_read_transformers(pm'`mem, address_block(addr', size(uidt(dt')))) ∪ m {2} OK?(memory_read_list(pm'`mem)(addr', size(uidt(dt')))(s')) {3} OK?(CASES memory_read_list(pm'`mem)(addr', size(uidt(dt')))(s') OF OK(state, data): ok_lift(from_byte(dt')(data, addr'))(state), Exception(ex_type, state): Exception(ex_type, state), Fatal: Fatal, Hang: Hang ENDCASES) </pre>
--	---

Rewriting using plain_memory_transformer_invariant_read_list_block, matching in * where pm gets pm!1,

This completes the proof of plain_memory_read_data_ok.2.1.

plain_memory_read_data_ok.2.2:

<pre> {-1} interpreted_data_type?[Data](dt') {-2} plain_memory?(pm') {-3} pm'`states(s') {-4} (address_block(addr', size(uidt(dt')))) ⊆ (pm'`ro_addr ∪ pm'`rw_addr) {-5} valid_in_mem(pm', dt')(addr')(s') </pre>	<pre> {1} transformers_ok?(pm'`states, (memory_read_transformers(pm'`mem, address_block(addr', size(uidt(dt')))) ∪ memory {2} OK?(memory_read_list(pm'`mem)(addr', size(uidt(dt')))(s')) {3} OK?(CASES memory_read_list(pm'`mem)(addr', size(uidt(dt')))(s') OF OK(state, data): ok_lift(from_byte(dt')(data, addr'))(state), Exception(ex_type, state): Exception(ex_type, state), Fatal: Fatal, Hang: Hang ENDCASES) </pre>
--	--

Rewriting using plain_memory_transformers_ok_read_list_block, matching in * where pm gets pm!1,

This completes the proof of plain_memory_read_data_ok.2.2.

Q.E.D.

C.172.4 Plain_Mem_Properties_2.plain_memory_write_data_ok

Terse proof for plain_memory_write_data_ok.

plain_memory_write_data_ok:

<pre> {1} ∃ (addr: Address, data: Data, dt: (interpreted_data_type?[Data]), pm: Plain_Memory[State], s: State): plain_memory?(pm) ∧ pm'`states(s) ∧ in_blessed_memory?(dt, addr, pm'`rw_addr) ⊃ OK?(write_data(pm, dt)(addr, data)(s)) </pre>	
--	--

Repeatedly Skolemizing and flattening,

Expanding the definition of write_data,

Expanding the definition of write_data,

C Proof scripts

Installing automatic rewrites from: `length_to_byte`

Expanding the definition of `in_blessed_memory?`,

Rewriting using `memory_write_list_ok`, matching in `*` where `pm` gets `pm!1'mem`, `states` gets `pm!1'states`,

we get 3 subgoals:

`plain_memory_write_data_ok.1`:

{-1}	<code>interpreted_data_type?[Data](dt')</code>
{-2}	<code>plain_memory?(pm')</code>
{-3}	<code>pm!'states(s')</code>
{-4}	<code>(address_block(addr', size(uidt(dt')))) ⊆ pm!'rw_addr</code>
<hr/>	
{1}	<code>transformer_invariant?(pm!'states,</code> <code>(memory_write_transformers(pm!'mem, address_block(addr', size(uidt(dt'))))</code>
{2}	<code>OK?(memory_write_list(pm!'mem)(addr', to_byte(dt')(data', addr'))(s'))</code>

Using lemma `plain_memory_transformer_invariant_write_list_block[State]`,

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `plain_memory_write_data_ok.1`.

`plain_memory_write_data_ok.2`:

{-1}	<code>interpreted_data_type?[Data](dt')</code>
{-2}	<code>plain_memory?(pm')</code>
{-3}	<code>pm!'states(s')</code>
{-4}	<code>(address_block(addr', size(uidt(dt')))) ⊆ pm!'rw_addr</code>
<hr/>	
{1}	<code>transformers_ok?(pm!'states,</code> <code>(memory_write_transformers(pm!'mem, address_block(addr', size(uidt(dt'))))</code>
{2}	<code>OK?(memory_write_list(pm!'mem)(addr', to_byte(dt')(data', addr'))(s'))</code>

Using lemma `plain_memory_transformers_ok_write_list_block[State]`,

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `plain_memory_write_data_ok.2`.

`plain_memory_write_data_ok.3`:

{-1}	<code>interpreted_data_type?[Data](dt')</code>
{-2}	<code>plain_memory?(pm')</code>
{-3}	<code>pm!'states(s')</code>
{-4}	<code>(address_block(addr', size(uidt(dt')))) ⊆ pm!'rw_addr</code>
<hr/>	
{1}	<code>side_effect_content_unchanged(address_block(addr', size(uidt(dt'))), pm!'states,</code> <code>memory_write_side_effect(pm!'mem))</code>
{2}	<code>OK?(memory_write_list(pm!'mem)(addr', to_byte(dt')(data', addr'))(s'))</code>

Using lemma `plain_memory_side_effect_content_unchanged_write_block[State]`,

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `plain_memory_write_data_ok.3`.

Q.E.D.

C.172.5

Plain_Mem_Properties_2.plain_memory_write_data_valid_TCC1

Terse proof for `plain_memory_write_data_valid_TCC1`.

plain_memory_write_data_valid_TCC1:

$$\frac{\{1\} \quad \forall (\text{addr}: \text{Address}, \text{data}: \text{Data}, \text{dt}: (\text{interpreted_data_type?}[\text{Data}]), \text{pm}: \text{Plain_Memory}[\text{State}], \text{s}: \text{State}):}{\text{in_blessed_memory?}(\text{dt}, \text{addr}, \text{pm}'\text{rw_addr}) \wedge \text{pm}'\text{states}(\text{s}) \wedge \text{plain_memory?}(\text{pm}) \supset \text{OK?}[\text{State}, \text{Unit}](\text{write_data}[\text{State}, \text{Data}](\text{pm}, \text{dt})(\text{addr}, \text{data})(\text{s})) \vee \text{Exception?}[\text{State}, \text{Unit}](\text{write_data}[\text{State}, \text{Data}](\text{pm}, \text{dt})(\text{addr}, \text{data})(\text{s}))}$$

Repeatedly Skolemizing and flattening,

Using lemma plain_memory_write_data_ok,

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of plain_memory_write_data_valid_TCC1.

Q.E.D.

C.172.6 Plain_Mem_Properties_2.plain_memory_write_data_valid

Terse proof for plain_memory_write_data_valid.

plain_memory_write_data_valid:

$$\frac{\{1\} \quad \forall (\text{addr}: \text{Address}, \text{data}: \text{Data}, \text{dt}: (\text{interpreted_data_type?}[\text{Data}]), \text{pm}: \text{Plain_Memory}[\text{State}], \text{s}: \text{State}):}{\text{plain_memory?}(\text{pm}) \wedge \text{pm}'\text{states}(\text{s}) \wedge \text{in_blessed_memory?}(\text{dt}, \text{addr}, \text{pm}'\text{rw_addr}) \supset \text{valid_in_mem}(\text{pm}, \text{dt})(\text{addr})(\text{state}(\text{write_data}(\text{pm}, \text{dt})(\text{addr}, \text{data})(\text{s})))}$$

Repeatedly Skolemizing and flattening,

Using lemma plain_memory_write_data_ok,

Simplifying, rewriting, and recording with decision procedures,

Expanding the definition of valid_in_mem,

Expanding the definition of valid_in_mem,

Expanding the definition of write_data,

Expanding the definition of write_data,

Installing automatic rewrites from: length_to_byte valid_to_byte subset_bigger_union_right

Using lemma plain_mem_write_list_read_list[State],

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 2 subgoals:

plain_memory_write_data_valid.1:

{-1}	plain_memory?(pm')
{-2}	pm' states(s')
{-3}	OK?(memory_write_list(pm' mem)(addr', to_byte(dt')(data', addr'))(s'))
{-4}	data(memory_read_list(pm' mem)(addr', size(uidt(dt')) <div style="text-align: right; margin-right: 20px;">(state(memory_write_list(pm' mem) <div style="text-align: right; margin-right: 20px;">(addr', to_byte(dt')(data', addr')) <div style="text-align: right; margin-right: 20px;">(s'))))</div> </div> </div>
	= to_byte(dt')(data', addr')
{-5}	interpreted_data_type?[Data](dt')
{-6}	in_blessed_memory?(dt', addr', pm' rw_addr)
{-7}	OK?(memory_read_list(pm' mem)(addr', size(uidt(dt')) <div style="text-align: right; margin-right: 20px;">(state(memory_write_list(pm' mem) <div style="text-align: right; margin-right: 20px;">(addr', to_byte(dt')(data', addr')) <div style="text-align: right; margin-right: 20px;">(s'))))</div> </div> </div>
{1} dt' uidt' valid? <div style="text-align: right; margin-right: 20px;">(data(memory_read_list(pm' mem)(addr', size(uidt(dt')) <div style="text-align: right; margin-right: 20px;">(state(memory_write_list(pm' mem) <div style="text-align: right; margin-right: 20px;">(addr', to_byte(dt')(data', addr')) <div style="text-align: right; margin-right: 20px;">(s')))),</div> </div> </div> </div>	
addr')	

Replacing using formula -4,

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of plain_memory_write_data_valid.1.

plain_memory_write_data_valid.2:

{-1}	plain_memory?(pm')
{-2}	pm' states(s')
{-3}	OK?(memory_write_list(pm' mem)(addr', to_byte(dt')(data', addr'))(s'))
{-4}	interpreted_data_type?[Data](dt')
{-5}	in_blessed_memory?(dt', addr', pm' rw_addr)
{-6}	OK?(memory_read_list(pm' mem)(addr', size(uidt(dt')) <div style="text-align: right; margin-right: 20px;">(state(memory_write_list(pm' mem) <div style="text-align: right; margin-right: 20px;">(addr', to_byte(dt')(data', addr')) <div style="text-align: right; margin-right: 20px;">(s'))))</div> </div> </div>
{1}	(address_block(addr', size(uidt(dt')))) \subseteq pm' rw_addr
{2}	dt' uidt' valid? <div style="text-align: right; margin-right: 20px;">(data(memory_read_list(pm' mem)(addr', size(uidt(dt')) <div style="text-align: right; margin-right: 20px;">(state(memory_write_list(pm' mem) <div style="text-align: right; margin-right: 20px;">(addr', to_byte(dt')(data', addr')) <div style="text-align: right; margin-right: 20px;">(s')))),</div> </div> </div> </div>
addr')	

Expanding the definition of in_blessed_memory?,

which is trivially true.

This completes the proof of plain_memory_write_data_valid.2.

Q.E.D.

C.172.7

Plain_Mem_Properties_2.plain_memory_transformers_ok_write_data

Terse proof for plain_memory_transformers_ok_write_data.

plain_memory_transformers_ok_write_data:

$\{1\} \quad \forall (\text{addr: Address, data: Data, dt: (interpreted_data_type? [Data]),}$ $\quad \text{pm: Plain_Memory [State]}):$ $\text{plain_memory? (pm) } \wedge \text{ in_blessed_memory? (dt, addr, pm'rw_addr) } \supset$ $\text{transformers_ok? (pm' states, singleton(expr_2_super(write_data(pm, dt)(addr, data))))}$

Expanding the definition of transformers_ok?,

Repeatedly Skolemizing and flattening,

Installing automatic rewrites from: length_to_byte singleton write_data plain_memory_side_effect_content_unchanged_write_list
 expr_2_super expr_2_super_res

Simplifying, rewriting, and recording with decision procedures,

Replacing using formula -5,

Using lemma memory_write_list_ok,

Using lemma plain_memory_transformer_invariant_write_list_block [State],

Using lemma plain_memory_transformers_ok_write_list_block [State],

Expanding the definition of in_blessed_memory?,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of plain_memory_transformers_ok_write_data.

Q.E.D.

C.172.8

Plain_Mem_Properties_2.plain_memory_transformer_invariant_write_data

Terse proof for plain_memory_transformer_invariant_write_data.

plain_memory_transformer_invariant_write_data:

$\{1\} \quad \forall (\text{addr: Address, data: Data, dt: (interpreted_data_type? [Data]),}$ $\quad \text{pm: Plain_Memory [State]}):$ $\text{plain_memory? (pm) } \wedge \text{ in_blessed_memory? (dt, addr, pm'rw_addr) } \supset$ $\text{transformer_invariant? (pm' states,}$ $\quad \text{singleton(expr_2_super(write_data(pm, dt)(addr, data))))}$
--

Expanding the definition of transformer_invariant?,

Repeatedly Skolemizing and flattening,

Expanding the definition of singleton,

Expanding the definition of write_data,

Expanding the definition of write_data,

Using lemma transformer_invariant_write_list,

Using lemma plain_memory_transformer_invariant_write_list_block [State],

Installing automatic rewrites from: length_to_byte

Expanding the definition of in_blessed_memory?,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 2 subgoals:

C Proof scripts

plain_memory_transformer_invariant_write_data.1:

{-1}	plain_memory?(pm')
{-2}	transformer_invariant?(pm' states, (memory_write_transformers(pm' mem, address_block(addr', size(uidt(dt')))))
{-3}	pm' states(s')
{-4}	interpreted_data_type?[Data](dt')
{-5}	(address_block(addr', size(uidt(dt')))) \subseteq pm' rw_addr
{-6}	$q' = \text{expr_2_super}(\text{memory_write_list}(\text{pm' mem})(\text{addr'}, \text{to_byte}(\text{dt}')(\text{data'}, \text{addr'})))$
{1}	side_effect_content_unchanged(address_block(addr', size(uidt(dt'))), pm' states, memory_write_side_effect(pm' mem))
{2}	result_pred(pm' states)(q'(s'))

Rewriting using plain_memory_side_effect_content_unchanged_write_block, matching in *,

This completes the proof of plain_memory_transformer_invariant_write_data.1.

plain_memory_transformer_invariant_write_data.2:

{-1}	plain_memory?(pm')
{-2}	transformer_invariant?(pm' states, (memory_write_transformers(pm' mem, address_block(addr', size(uidt(dt')))))
{-3}	pm' states(s')
{-4}	interpreted_data_type?[Data](dt')
{-5}	(address_block(addr', size(uidt(dt')))) \subseteq pm' rw_addr
{-6}	$q' = \text{expr_2_super}(\text{memory_write_list}(\text{pm' mem})(\text{addr'}, \text{to_byte}(\text{dt}')(\text{data'}, \text{addr'})))$
{1}	transformer_invariant?(pm' states, (memory_write_transformers(pm' mem, address_block(addr', size(uidt(dt')))))
{2}	result_pred(pm' states)(q'(s'))

Keeping (-2 -3 -5 1) and hiding *,

Rewriting using add_as_union, matching in *,

Using lemma transformer_invariant_mono_transformers,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Hiding formulas: (-1 2),

Expanding the definition of subset?,

Expanding the definition of union,

Expanding the definition of member,

Repeatedly Skolemizing and flattening,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Expanding the definition of memory_write_side_effect_super_transformers,

Instantiating quantified variables,

Simplifying, rewriting, and recording with decision procedures,

Rewriting using subset_reflexive, matching in *,

Expanding the definition of singleton,

which is trivially true.

This completes the proof of plain_memory_transformer_invariant_write_data.2.

Q.E.D.

C.172.9

Plain_Mem_Properties_2.plain_memory_transformer_invariant_read_data

Terse proof for plain_memory_transformer_invariant_read_data.

plain_memory_transformer_invariant_read_data:

$\{1\} \quad \forall (\text{addr: Address, dt: (interpreted_data_type? [Data])}, \text{pm: Plain_Memory [State]}):$ $\text{plain_memory?}(\text{pm}) \wedge \text{in_blessed_memory?}(\text{dt}, \text{addr}, (\text{pm}'\text{ro_addr} \cup \text{pm}'\text{rw_addr})) \supset$ $\text{transformer_invariant?}(\text{pm}'\text{states}, \text{singleton}(\text{expr_2_super}(\text{read_data}(\text{pm}, \text{dt})(\text{addr}))))$

Expanding the definition of transformer_invariant?,

Installing automatic rewrites from: result_pred has_next_state

Repeatedly Skolemizing and flattening,

Simplifying, rewriting, and recording with decision procedures,

Expanding the definition of singleton,

Replacing using formula -5,

Installing automatic rewrites from: (expr_2_super! expr_2_super_res! ok_lift! read_data! ##!)

Simplifying, rewriting, and recording with decision procedures,

Using lemma memory_read_list_next_ok [State],

Expanding the definition of in_blessed_memory?,

Rewriting using plain_memory_transformers_ok_read_list_block [State], matching in *,

Rewriting using plain_memory_transformer_invariant_read_list_block [State], matching in *,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of plain_memory_transformer_invariant_read_data.

Q.E.D.

C.172.10

Plain_Mem_Properties_2.plain_memory_unchanged_memory_invariant_write_data

Terse proof for plain_memory_unchanged_memory_invariant_write_data.

plain_memory_unchanged_memory_invariant_write_data:

$\{1\} \quad \forall (\text{addr: Address, data: Data, dt: (interpreted_data_type? [Data])},$ $\text{pm: Plain_Memory [State]}):$ $\text{plain_memory?}(\text{pm}) \wedge \text{in_blessed_memory?}(\text{dt}, \text{addr}, \text{pm}'\text{rw_addr}) \supset$ $\text{unchanged_memory_invariant?}(\text{pm}'\text{mem}, \text{pm}'\text{states},$ $\text{singleton}(\text{expr_2_super}(\text{write_data}(\text{pm}, \text{dt})(\text{addr}, \text{data}))),$ $((\text{pm}'\text{ro_addr} \cup \text{pm}'\text{rw_addr}) \setminus \text{address_block}(\text{addr}, \text{size}(\text{uidt}(\text{dt}))))))$

Expanding the definition of unchanged_memory_invariant?,

Repeatedly Skolemizing and flattening,

Applying propositional simplification,

we get 2 subgoals:

plain_memory_unchanged_memory_invariant_write_data.1:

$\{-1\} \quad \text{interpreted_data_type? [Data]}(\text{dt}')$ $\{-2\} \quad \text{plain_memory?}(\text{pm}')$ $\{-3\} \quad \text{in_blessed_memory?}(\text{dt}', \text{addr}', \text{pm}'\text{rw_addr})$ <hr/> $\{1\} \quad \text{transformer_invariant?}(\text{pm}'\text{states},$ $\text{singleton}(\text{expr_2_super}(\text{write_data}(\text{pm}', \text{dt}')(\text{addr}', \text{data}'))))$

Using lemma plain_memory_transformer_invariant_write_data,

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of plain_memory_unchanged_memory_invariant_write_data.1.

plain_memory_unchanged_memory_invariant_write_data.2:

{-1}	interpreted_data_type?[Data](dt')
{-2}	plain_memory?(pm')
{-3}	in_blessed_memory?(dt', addr', pm'rw_addr)
{1}	$\forall (s: \text{State}, q: [\text{State} \rightarrow \text{SuperResult}[\text{State}]], a: \text{Address}):$ $\text{pm}'\text{states}(s) \wedge$ $\text{singleton}(\text{expr_2_super}(\text{write_data}(\text{pm}', \text{dt}')(\text{addr}', \text{data'}))(q) \wedge$ $\text{difference}((\text{pm}'\text{ro_addr} \cup \text{pm}'\text{rw_addr}), \text{address_block}(\text{addr}', \text{size}(\text{uidt}(\text{dt'}))))$ (a) \wedge $\text{OK?}(q(s)) \wedge$ $\text{OK?}(\text{memory_read}(\text{pm}'\text{mem})(a)(s)) \wedge$ $\text{OK?}(\text{memory_read}(\text{pm}'\text{mem})(a)(\text{state}(q(s))))$ \supset $\text{data}(\text{memory_read}(\text{pm}'\text{mem})(a)(\text{state}(q(s)))) =$ $\text{data}(\text{memory_read}(\text{pm}'\text{mem})(a)(s))$

Repeatedly Skolemizing and flattening,
Expanding the definition of write_data,
Expanding the definition of write_data,
Installing automatic rewrites from: length_to_byte ok_expr_2_super state_expr_2_super has_next_state
subset_singleton

Expanding the definition of singleton,
Replacing using formula -2,
Simplifying, rewriting, and recording with decision procedures,
Expanding the definition of in_blessed_memory?,
Using lemma unchanged_memory_invariant_unchanged_write_list [State],
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
we get 4 subgoals:

plain_memory_unchanged_memory_invariant_write_data.2.1:

{-1}	pm' states(s')
{-2}	q' = expr_2_super(memory_write_list(pm' mem)(addr', to_byte(dt')(data', addr')))
{-3}	difference((pm'ro_addr ∪ pm'rw_addr), address_block(addr', size(uidt(dt'))))
	(a')
{-4}	OK?(memory_write_list(pm' mem)(addr', to_byte(dt')(data', addr'))(s'))
{-5}	OK?(memory_read(pm' mem)(a')(s'))
{-6}	OK?(memory_read(pm' mem)
	(a')
	(state(memory_write_list(pm' mem)(addr', to_byte(dt')(data', addr'))
	(s'))))
{-7}	interpreted_data_type?[Data](dt')
{-8}	plain_memory?(pm')
{-9}	(address_block(addr', size(uidt(dt'))) ⊆ pm'rw_addr)
{1}	side_effect_content_unchanged(address_block(addr', size(uidt(dt'))), pm' states,
	memory_write_side_effect(pm' mem))
{2}	data(memory_read(pm' mem)
	(a')
	(state(memory_write_list(pm' mem)(addr', to_byte(dt')(data', addr'))
	(s'))))
	= data(memory_read(pm' mem)(a')(s'))

Rewriting using plain_memory_side_effect_content_unchanged_write_block, matching in *,
This completes the proof of plain_memory_unchanged_memory_invariant_write_data.2.1.

plain_memory_unchanged_memory_invariant_write_data.2.2:

{-1}	$pm' \text{ states}(s')$
{-2}	$q' = \text{expr_2_super}(\text{memory_write_list}(pm' \text{ mem})(\text{addr}', \text{to_byte}(dt')(data', \text{addr}')))$
{-3}	$\text{difference}((pm' \text{ ro_addr} \cup pm' \text{ rw_addr}), \text{address_block}(\text{addr}', \text{size}(\text{uidt}(dt'))))$ (a')
{-4}	$\text{OK?}(\text{memory_write_list}(pm' \text{ mem})(\text{addr}', \text{to_byte}(dt')(data', \text{addr}'))(s'))$
{-5}	$\text{OK?}(\text{memory_read}(pm' \text{ mem})(a')(s'))$
{-6}	$\text{OK?}(\text{memory_read}(pm' \text{ mem})$ (a') $(\text{state}(\text{memory_write_list}(pm' \text{ mem})(\text{addr}', \text{to_byte}(dt')(data', \text{addr}'))$ $(s'))))$
{-7}	$\text{interpreted_data_type?}[\text{Data}](dt')$
{-8}	$\text{plain_memory?}(pm')$
{-9}	$(\text{address_block}(\text{addr}', \text{size}(\text{uidt}(dt')))) \subseteq pm' \text{ rw_addr}$
{1}	$\text{transformers_ok?}(pm' \text{ states}, \text{singleton}(\text{expr_2_super}(\text{memory_read}(pm' \text{ mem})(a'))))$
{2}	$\text{data}(\text{memory_read}(pm' \text{ mem})$ (a') $(\text{state}(\text{memory_write_list}(pm' \text{ mem})(\text{addr}', \text{to_byte}(dt')(data', \text{addr}'))$ $(s'))))$ $= \text{data}(\text{memory_read}(pm' \text{ mem})(a')(s'))$

Keeping (-3 -8 1) and hiding *,

Using lemma plain_memory_transformers_ok_read_ro_rw,

Simplifying, rewriting, and recording with decision procedures,

Using lemma transformers_ok_mono_transformers[State],

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Rewriting using memory_read_transformers_memory_read, matching in *,

Keeping (-2 1) and hiding *,

Expanding the definition(s) of (difference member),

This completes the proof of plain_memory_unchanged_memory_invariant_write_data.2.2.

plain_memory_unchanged_memory_invariant_write_data.2.3:

{-1}	$pm' \text{ states}(s')$
{-2}	$q' = \text{expr_2_super}(\text{memory_write_list}(pm' \text{ mem})(\text{addr}', \text{to_byte}(dt')(data', \text{addr}')))$
{-3}	$\text{difference}((pm' \text{ ro_addr} \cup pm' \text{ rw_addr}), \text{address_block}(\text{addr}', \text{size}(\text{uidt}(dt'))))$ (a')
{-4}	$\text{OK?}(\text{memory_write_list}(pm' \text{ mem})(\text{addr}', \text{to_byte}(dt')(data', \text{addr}'))(s'))$
{-5}	$\text{OK?}(\text{memory_read}(pm' \text{ mem})(a')(s'))$
{-6}	$\text{OK?}(\text{memory_read}(pm' \text{ mem})$ (a') $(\text{state}(\text{memory_write_list}(pm' \text{ mem})(\text{addr}', \text{to_byte}(dt')(data', \text{addr}'))$ $(s'))))$
{-7}	$\text{interpreted_data_type?}[\text{Data}](dt')$
{-8}	$\text{plain_memory?}(pm')$
{-9}	$(\text{address_block}(\text{addr}', \text{size}(\text{uidt}(dt')))) \subseteq pm' \text{ rw_addr}$
{1}	$\text{transformers_ok?}(pm' \text{ states},$ $(\text{memory_write_transformers}(pm' \text{ mem}, \text{address_block}(\text{addr}', \text{size}(\text{uidt}(dt')))) \cup \text{memory_read_transformers}(pm' \text{ mem}, \text{address_block}(\text{addr}', \text{size}(\text{uidt}(dt')))))$
{2}	$\text{data}(\text{memory_read}(pm' \text{ mem})$ (a') $(\text{state}(\text{memory_write_list}(pm' \text{ mem})(\text{addr}', \text{to_byte}(dt')(data', \text{addr}'))$ $(s'))))$ $= \text{data}(\text{memory_read}(pm' \text{ mem})(a')(s'))$

Rewriting using plain_memory_transformers_ok_write_list_block, matching in *,

C Proof scripts

This completes the proof of `plain_memory_unchanged_memory_invariant_write_data.2.3`.

`plain_memory_unchanged_memory_invariant_write_data.2.4`:

<pre> {-1} pm' 'states(s') {-2} q' = expr_2_super(memory_write_list(pm' 'mem)(addr', to_byte(dt')(data', addr'))) {-3} difference((pm' 'ro_addr ∪ pm' 'rw_addr), address_block(addr', size(uidt(dt')))) (a') {-4} OK?(memory_write_list(pm' 'mem)(addr', to_byte(dt')(data', addr'))(s')) {-5} OK?(memory_read(pm' 'mem)(a')(s')) {-6} OK?(memory_read(pm' 'mem) (a') (state(memory_write_list(pm' 'mem)(addr', to_byte(dt')(data', addr')) (s')))) {-7} interpreted_data_type?[Data](dt') {-8} plain_memory?(pm') {-9} (address_block(addr', size(uidt(dt')))) ⊆ pm' 'rw_addr </pre>	<pre> {1} unchanged_memory_invariant?(pm' 'mem, pm' 'states, (memory_write_transformers(pm' 'mem, address_block(addr', si singleton(a')) {2} data(memory_read(pm' 'mem) (a') (state(memory_write_list(pm' 'mem)(addr', to_byte(dt')(data', addr')) (s')))) = data(memory_read(pm' 'mem)(a')(s')) </pre>
---	---

Keeping (-3 -8 -10 1) and hiding *,

Rewriting using `unchanged_memory_invariant_union_transformers[State]`, matching in *,

we get 2 subgoals:

`plain_memory_unchanged_memory_invariant_write_data.2.4.1`:

<pre> {-1} difference((pm' 'ro_addr ∪ pm' 'rw_addr), address_block(addr', size(uidt(dt')))) (a') {-2} plain_memory?(pm') </pre>	<pre> {1} unchanged_memory_invariant?(pm' 'mem, pm' 'states, memory_write_transformers(pm' 'mem, address_block(addr', size(uidt(dt')) singleton(a')) {2} unchanged_memory_invariant?(pm' 'mem, pm' 'states, (memory_write_transformers(pm' 'mem, address_block(addr', si singleton(a')) </pre>
---	--

Hiding formulas: 2,

Using lemma `unchanged_memory_invariant_union[State]`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 3 subgoals:

plain_memory_unchanged_memory_invariant_write_data.2.4.1.1:

{-1}	unchanged_memory_invariant?(pm' mem, pm' states,	memory_write_transformers(pm' mem,	address_block(addr',	size(uidt(dt'))),
{-2}	difference((pm' ro_addr \cup pm' rw_addr), address_block(addr', size(uidt(dt'))))			
{-3}	plain_memory?(pm')			
{1}	unchanged_memory_invariant?(pm' mem, pm' states,	memory_write_transformers(pm' mem,	address_block(addr',	size(uidt(dt'))),
				singleton(a')

Using lemma unchanged_memory_invariant_mono[State],

Installing automatic rewrites from: subset_equal

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of plain_memory_unchanged_memory_invariant_write_data.2.4.1.1.

plain_memory_unchanged_memory_invariant_write_data.2.4.1.2:

{-1}	difference((pm' ro_addr \cup pm' rw_addr), address_block(addr', size(uidt(dt'))))			
{-2}	plain_memory?(pm')			
{1}	unchanged_memory_write_invariant?(pm' mem, pm' states, pm' rw_addr)			
{2}	unchanged_memory_invariant?(pm' mem, pm' states,	memory_write_transformers(pm' mem,	address_block(addr',	size(uidt(dt'))),
				singleton(a')

Rewriting using plain_memory_unchanged_memory_write_invariant, matching in *,

This completes the proof of plain_memory_unchanged_memory_invariant_write_data.2.4.1.2.

plain_memory_unchanged_memory_invariant_write_data.2.4.1.3:

{-1}	difference((pm' ro_addr \cup pm' rw_addr), address_block(addr', size(uidt(dt'))))			
{-2}	plain_memory?(pm')			
{1}	unchanged_memory_invariant?(pm' mem, pm' states,	memory_write_transformers(pm' mem, pm' rw_addr), pm' ro_addr)		
{2}	unchanged_memory_invariant?(pm' mem, pm' states,	memory_write_transformers(pm' mem,	address_block(addr',	size(uidt(dt'))),
				singleton(a')

Rewriting using plain_memory_unchanged_memory_invariant_write, matching in *,

This completes the proof of plain_memory_unchanged_memory_invariant_write_data.2.4.1.3.

C Proof scripts

plain_memory_unchanged_memory_invariant_write_data.2.4.2:

{-1}	difference((pm'ro_addr \cup pm'rw_addr), address_block(addr', size(uidt(dt'))))
{-2}	plain_memory?(pm')
{1}	unchanged_memory_invariant?(pm'mem, pm'states, memory_write_side_effect_super_transformers(pm'mem, ad- dress_block (addr', size (uidt(dt
{2}	unchanged_memory_invariant?(pm'mem, pm'states, memory_write_transformers(pm'mem, address_block(addr', si singleton(a'))

Hiding formulas: 2,

Using lemma plain_memory_unchanged_invariant,

Using lemma unchanged_memory_invariant_mono[State],

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 2 subgoals:

plain_memory_unchanged_memory_invariant_write_data.2.4.2.1:

{-1}	unchanged_memory_invariant?(pm'mem, pm'states, ((pm'other_actions \cup memory_read_transformers(pm'mem, (pm'ro_addr \cup pm'rw_addr))
{-2}	plain_memory?(pm')
{-3}	difference((pm'ro_addr \cup pm'rw_addr), address_block(addr', size(uidt(dt'))))
{1}	union(pm'ro_addr, pm'rw_addr)(a')
{2}	unchanged_memory_invariant?(pm'mem, pm'states, memory_write_side_effect_super_transformers(pm'mem, ad- dress_block (addr', size (uidt(dt
	singleton(a'))

Keeping (-3 1) and hiding *,

Expanding the definition(s) of (difference member),

This completes the proof of plain_memory_unchanged_memory_invariant_write_data.2.4.2.1.

plain_memory_unchanged_memory_invariant_write_data.2.4.2.2:

{-1}	unchanged_memory_invariant?(pm' mem, pm' states, ((pm' other_actions \cup memory_read_transformers(pm' mem, (pm' ro_addr \cup (pm' ro_addr \cup pm' rw_addr)))
{-2}	plain_memory?(pm')
{-3}	difference((pm' ro_addr \cup pm' rw_addr), address_block(addr', size(uidt(dt')))) (a')
{1}	(memory_write_side_effect_super_transformers(pm' mem, address_block(addr', size(uidt(dt')))) \subseteq ((pm' other_actions \cup memory_read_transformers(pm' mem, (pm' ro_addr \cup pm' rw_addr))))
{2}	unchanged_memory_invariant?(pm' mem, pm' states, memory_write_side_effect_super_transformers(pm' mem, address_block(addr', size(uidt(dt')))) address_block(addr', size(uidt(dt')))) singleton(a'))

Keeping (-4 1) and hiding *,

Installing automatic rewrites from: memory_write_side_effect_super_transformers_mono

Rewriting using subset_bigger_union_right, matching in *,

Rewriting using subset_bigger_union_right, matching in *,

This completes the proof of plain_memory_unchanged_memory_invariant_write_data.2.4.2.2.

Q.E.D.

C.172.11

Plain_Mem_Properties_2.plain_memory_unchanged_memory_invariant_read_data

Terse proof for plain_memory_unchanged_memory_invariant_read_data.

plain_memory_unchanged_memory_invariant_read_data:

{1}	\forall (addr: Address, dt: (interpreted_data_type?[Data]), pm: Plain_Memory[State]): plain_memory?(pm) \wedge in_blessed_memory?(dt, addr, (pm' ro_addr \cup pm' rw_addr)) \supset unchanged_memory_invariant?(pm' mem, pm' states, singleton(expr_2_super(read_data(pm, dt)(addr))), (pm' ro_addr \cup pm' rw_addr))
-----	--

Installing automatic rewrites from: ok_expr_2_super state_expr_2_super singleton ## ok_lift result_pred has_next_state has_next_state_expr_2_super in_blessed_memory_subset transformers_ok_union_transformers plain_memory_transformers_ok_read_side_effects_ro_rw plain_memory_transformers_ok_read_ro_rw plain_memory_transformers_ok_read_side_effects_block plain_memory_transformers_ok_read_block

Repeatedly Skolemizing and flattening,

Using lemma unchanged_memory_invariant_read_list[State],

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 2 subgoals:

plain_memory_unchanged_memory_invariant_read_data.1:

{-1}	unchanged_memory_invariant?(pm' mem, pm' states, singleton(expr_2_super(memory_read_list(pm' mem) (addr', size(uidt(dt') (pm'ro_addr ∪ pm'rw_addr))
{-2}	interpreted_data_type?[Data](dt')
{-3}	plain_memory?(pm')
{-4}	in_blessed_memory?(dt', addr', (pm'ro_addr ∪ pm'rw_addr))
{1}	unchanged_memory_invariant?(pm' mem, pm' states, singleton(expr_2_super(read_data(pm', dt')(addr'))), (pm'ro_addr ∪ pm'rw_addr))

Expanding the definition of unchanged_memory_invariant?,

Applying propositional simplification,

we get 2 subgoals:

plain_memory_unchanged_memory_invariant_read_data.1.1:

{-1}	unchanged_memory_invariant?(pm' mem, pm' states, singleton(expr_2_super(memory_read_list(pm' mem) (addr', size(uidt(dt') (pm'ro_addr ∪ pm'rw_addr))
{-2}	interpreted_data_type?[Data](dt')
{-3}	plain_memory?(pm')
{-4}	in_blessed_memory?(dt', addr', (pm'ro_addr ∪ pm'rw_addr))
{1}	transformer_invariant?(pm' states, singleton(expr_2_super(read_data(pm', dt')(addr'))))

Expanding the definition of transformer_invariant?,

Repeatedly Skolemizing and flattening,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Using lemma memory_read_list_ok[State],

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 2 subgoals:

plain_memory_unchanged_memory_invariant_read_data.1.1.1:

{-1}	pm' states(s')
{-2}	OK?(memory_read_list(pm' mem)(addr', size(uidt(dt')))(s'))
{-3}	q' = expr_2_super(read_data(pm', dt')(addr'))
{-4}	has_next_state(q'(s'))
{-5}	unchanged_memory_invariant?(pm' mem, pm' states, singleton(expr_2_super(memory_read_list(pm' mem) (addr', size(uidt(dt') (pm'ro_addr ∪ pm'rw_addr))
{-6}	interpreted_data_type?[Data](dt')
{-7}	plain_memory?(pm')
{-8}	in_blessed_memory?(dt', addr', (pm'ro_addr ∪ pm'rw_addr))
{1}	pm' states(state(q'(s')))

Replacing using formula -3,

Hiding formulas: -3,

Simplifying, rewriting, and recording with decision procedures,

Installing automatic rewrites from: read_data ok_lift!

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Using lemma expr_transformer_invariant_next_ok,

Forward chaining on unchanged_memory_invariant_invariant,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of plain_memory_unchanged_memory_invariant_read_data.1.1.1.

plain_memory_unchanged_memory_invariant_read_data.1.1.2:

{-1}	pm' states(s')
{-2}	q' = expr_2_super(read_data(pm', dt')(addr'))
{-3}	has_next_state(q'(s'))
{-4}	unchanged_memory_invariant?(pm' mem, pm' states, singleton(expr_2_super(memory_read_list(pm' mem) (addr', size(uidt(dt'))))), (pm' ro_addr ∪ pm' rw_addr))
{-5}	interpreted_data_type?[Data](dt')
{-6}	plain_memory?(pm')
{-7}	in_blessed_memory?(dt', addr', (pm' ro_addr ∪ pm' rw_addr))
{1}	transformer_invariant?(pm' states, (memory_read_transformers(pm' mem, address_block(addr', size(uidt(dt')))) ∪ m
{2}	pm' states(state(q'(s')))

Rewriting using plain_memory_transformer_invariant_read_list_block, matching in *,

This completes the proof of plain_memory_unchanged_memory_invariant_read_data.1.1.2.

plain_memory_unchanged_memory_invariant_read_data.1.2:

{-1}	unchanged_memory_invariant?(pm' mem, pm' states, singleton(expr_2_super(memory_read_list(pm' mem) (addr', size(uidt(dt'))))), (pm' ro_addr ∪ pm' rw_addr))
{-2}	interpreted_data_type?[Data](dt')
{-3}	plain_memory?(pm')
{-4}	in_blessed_memory?(dt', addr', (pm' ro_addr ∪ pm' rw_addr))
{1}	$\forall (s: \text{State}, q: [\text{State} \rightarrow \text{SuperResult}[\text{State}]], a: \text{Address}):$ $\text{pm}' \text{ states}(s) \wedge$ $\text{singleton}(\text{expr_2_super}(\text{read_data}(\text{pm}', \text{dt}')(\text{addr}')))(q) \wedge$ $\text{union}(\text{pm}' \text{ ro_addr}, \text{pm}' \text{ rw_addr})(a) \wedge$ $\text{OK?}(q(s)) \wedge$ $\text{OK?}(\text{memory_read}(\text{pm}' \text{ mem})(a)(s)) \wedge$ $\text{OK?}(\text{memory_read}(\text{pm}' \text{ mem})(a)(\text{state}(q(s))))$ \supset $\text{data}(\text{memory_read}(\text{pm}' \text{ mem})(a)(\text{state}(q(s)))) =$ $\text{data}(\text{memory_read}(\text{pm}' \text{ mem})(a)(s))$

Repeatedly Skolemizing and flattening,

Simplifying, rewriting, and recording with decision procedures,

Replacing using formula -2,

Hiding formulas: -2,

Installing automatic rewrites from: read_data ok_lift! ##!

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Using lemma expr_unchanged_memory_invariant_unchanged[State, list[Byte]],

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `plain_memory_unchanged_memory_invariant_read_data.1.2`.

`plain_memory_unchanged_memory_invariant_read_data.2`:

{-1}	<code>interpreted_data_type?[Data](dt')</code>
{-2}	<code>plain_memory?(pm')</code>
{-3}	<code>in_blessed_memory?(dt', addr', (pm'ro_addr ∪ pm'rw_addr))</code>
{1}	$\text{unchanged_memory_invariant?}(pm'\text{mem}, pm'\text{states},$ $\text{memory_read_transformers}(pm'\text{mem}, \text{address_block}(addr', \text{size}(\text{uidt}(dt'))),$ $(pm'\text{ro_addr} \cup pm'\text{rw_addr}))$
{2}	$\text{unchanged_memory_invariant?}(pm'\text{mem}, pm'\text{states},$ $\text{singleton}(\text{expr_2_super}(\text{read_data}(pm', dt')(addr'))),$ $(pm'\text{ro_addr} \cup pm'\text{rw_addr}))$

Hiding formulas: 2,

Installing automatic rewrites from: `plain_memory_unchanged_read_block_ro_rw_addr plain_memory_unchanged_`

Rewriting using `unchanged_memory_invariant_union_transformers`, matching in `*`,

This completes the proof of `plain_memory_unchanged_memory_invariant_read_data.2`.

Q.E.D.

C.172.12

Plain_Mem_Properties_2.plain_memory_read_data_q_ok_TCC1

Terse proof for `plain_memory_read_data_q_ok_TCC1`.

`plain_memory_read_data_q_ok_TCC1`:

{1}	$\forall (\text{addr}: \text{Address}, \text{dt}: (\text{interpreted_data_type?}[\text{Data}])), \text{pm}: \text{Plain_Memory}[\text{State}], \text{s}: \text{State},$ $q: [\text{State} \rightarrow \text{SuperResult}[\text{State}]]:$ $\text{plain_memory?}(\text{pm}) \wedge$ $\text{in_blessed_memory?}(\text{dt}, \text{addr}, (\text{pm'ro_addr} \cup \text{pm'rw_addr})) \wedge$ $\text{unchanged_memory_invariant?}(\text{pm' mem}, \text{pm' states}, \text{singleton}(q),$ $\text{address_block}(\text{addr}, \text{size}(\text{uidt}(\text{dt}))))$ $\wedge \text{pm' states}(\text{s}) \wedge \text{OK?}(q(\text{s})) \wedge \text{valid_in_mem}(\text{pm}, \text{dt})(\text{addr})(\text{s})$ $\supset \text{OK?}[\text{State}](q(\text{s})) \vee \text{abnormal?}[\text{State}](q(\text{s}))$
-----	--

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `plain_memory_read_data_q_ok_TCC1`.

Q.E.D.

C.172.13 Plain_Mem_Properties_2.plain_memory_read_data_q_ok

Terse proof for `plain_memory_read_data_q_ok`.

`plain_memory_read_data_q_ok`:

{1}	$\forall (\text{addr}: \text{Address}, \text{dt}: (\text{interpreted_data_type?}[\text{Data}])), \text{pm}: \text{Plain_Memory}[\text{State}], \text{s}: \text{State},$ $q: [\text{State} \rightarrow \text{SuperResult}[\text{State}]]:$ $\text{plain_memory?}(\text{pm}) \wedge$ $\text{in_blessed_memory?}(\text{dt}, \text{addr}, (\text{pm'ro_addr} \cup \text{pm'rw_addr})) \wedge$ $\text{unchanged_memory_invariant?}(\text{pm' mem}, \text{pm' states}, \text{singleton}(q),$ $\text{address_block}(\text{addr}, \text{size}(\text{uidt}(\text{dt}))))$ $\wedge \text{pm' states}(\text{s}) \wedge \text{OK?}(q(\text{s})) \wedge \text{valid_in_mem}(\text{pm}, \text{dt})(\text{addr})(\text{s})$ $\supset \text{OK?}(\text{read_data}(\text{pm}, \text{dt})(\text{addr})(\text{state}(q(\text{s}))))$
-----	---

C.172 Proofs for Plain_Mem_Properties_2 (plain_memory.pvs)

Installing automatic rewrites from: (valid_in_mem read_data ## expr_2_super expr_2_super_res ok_lift has_next_state)

Repeatedly Skolemizing and flattening,

Using lemma plain_mem_q_read_list[State],

Using lemma memory_read_list_ok[State],

Using lemma memory_read_list_ok[State],

Expanding the definition of in_blessed_memory?,

Simplifying, rewriting, and recording with decision procedures,

Using lemma plain_memory_transformers_ok_read_list_block[State],

Using lemma plain_memory_transformer_invariant_read_list_block[State],

Case splitting on OK?(memory_read_list(pm!1'mem)(addr!1, size(uidt(dt!1))) (state(q!1(s!1))))),

we get 2 subgoals:

plain_memory_read_data_q_ok.1:

```

{-1} OK?(memory_read_list(pm' mem)(addr', size(uidt(dt')))(state(q'(s'))))
{-2} plain_memory?(pm') ∧
      (address_block(addr', size(uidt(dt'))) ⊆ (pm'ro_addr ∪ pm'rw_addr))
      ⊃
      transformer_invariant?(pm' states,
                              (memory_read_transformers(pm' mem, address_block(addr', size(uidt(dt')))))
{-3} plain_memory?(pm') ∧
      (address_block(addr', size(uidt(dt'))) ⊆ (pm'ro_addr ∪ pm'rw_addr))
      ⊃
      transformers_ok?(pm' states,
                       (memory_read_transformers(pm' mem, address_block(addr', size(uidt(dt')))))
{-4} transformer_invariant?(pm' states,
                              (memory_read_transformers(pm' mem, address_block(addr', size(uidt(dt')))))
      ∧
      transformers_ok?(pm' states,
                       (memory_read_transformers(pm' mem, address_block(addr', size(uidt(dt')))))
      ∧ pm' states(state(q'(s')))
      ⊃ OK?(memory_read_list(pm' mem)(addr', size(uidt(dt')))(state(q'(s'))))
{-5} transformer_invariant?(pm' states,
                              (memory_read_transformers(pm' mem, address_block(addr', size(uidt(dt')))))
      ∧
      transformers_ok?(pm' states,
                       (memory_read_transformers(pm' mem, address_block(addr', size(uidt(dt')))))
      ⊃ OK?(memory_read_list(pm' mem)(addr', size(uidt(dt')))(s'))
{-6} OK?(memory_read_list(pm' mem)(addr', size(uidt(dt')))(s')) ∧
      OK?(memory_read_list(pm' mem)(addr', size(uidt(dt')))(state(q'(s'))))
      ⊃
      data(memory_read_list(pm' mem)(addr', size(uidt(dt')))(state(q'(s')))) =
      data(memory_read_list(pm' mem)(addr', size(uidt(dt')))(s'))
{-7} interpreted_data_type?[Data](dt')
{-8} plain_memory?(pm')
{-9} (address_block(addr', size(uidt(dt'))) ⊆ (pm'ro_addr ∪ pm'rw_addr))
{-10} unchanged_memory_invariant?(pm' mem, pm' states, singleton(q'),
                                   address_block(addr', size(uidt(dt'))))
{-11} pm' states(s')
{-12} OK?(q'(s'))
{-13} CASES memory_read_list(pm' mem)(addr', size(uidt(dt')))(s') OF
      OK(sn, bl): dt' uidt' valid?(bl, addr')
      ELSE TRUE
      ENDCASES
-----
{1} OK?(CASES memory_read_list(pm' mem)(addr', size(uidt(dt')))(state(q'(s')))) OF
      OK(state_1, data): ok_lift(from_byte(dt')(data, addr'))(state_1),
      Exception(ex_type, state_1): Exception(ex_type, state_1),
      Fatal: Fatal,
      Hang: Hang
      ENDCASES)

```

Simplifying, rewriting, and recording with decision procedures,

Replacing using formula -5,

Using lemma valid_from_byte[Data],

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of plain_memory_read_data_q_ok.1.

plain_memory_read_data_q_ok.2:

{-1}	plain_memory?(pm') \wedge (address_block(addr', size(uidt(dt')))) \subseteq (pm'ro_addr \cup pm'rw_addr) \supset transformer_invariant?(pm' states, (memory_read_transformers(pm'mem, address_block(addr', size(uidt(dt'))))) \cup memo
{-2}	plain_memory?(pm') \wedge (address_block(addr', size(uidt(dt')))) \subseteq (pm'ro_addr \cup pm'rw_addr) \supset transformers_ok?(pm' states, (memory_read_transformers(pm'mem, address_block(addr', size(uidt(dt'))))) \cup memo
{-3}	transformer_invariant?(pm' states, (memory_read_transformers(pm'mem, address_block(addr', size(uidt(dt'))))) \cup memo \wedge transformers_ok?(pm' states, (memory_read_transformers(pm'mem, address_block(addr', size(uidt(dt'))))) \cup memo \wedge pm' states(state(q'(s'))) \supset OK?(memory_read_list(pm'mem)(addr', size(uidt(dt')))(state(q'(s'))))
{-4}	transformer_invariant?(pm' states, (memory_read_transformers(pm'mem, address_block(addr', size(uidt(dt'))))) \cup memo \wedge transformers_ok?(pm' states, (memory_read_transformers(pm'mem, address_block(addr', size(uidt(dt'))))) \cup memo \supset OK?(memory_read_list(pm'mem)(addr', size(uidt(dt')))(s'))
{-5}	OK?(memory_read_list(pm'mem)(addr', size(uidt(dt')))(s')) \wedge OK?(memory_read_list(pm'mem)(addr', size(uidt(dt')))(state(q'(s')))) \supset data(memory_read_list(pm'mem)(addr', size(uidt(dt')))(state(q'(s')))) = data(memory_read_list(pm'mem)(addr', size(uidt(dt')))(s'))
{-6}	interpreted_data_type?[Data](dt')
{-7}	plain_memory?(pm')
{-8}	(address_block(addr', size(uidt(dt')))) \subseteq (pm'ro_addr \cup pm'rw_addr)
{-9}	unchanged_memory_invariant?(pm'mem, pm' states, singleton(q'), address_block(addr', size(uidt(dt'))))
{-10}	pm' states(s')
{-11}	OK?(q'(s'))
{-12}	CASES memory_read_list(pm'mem)(addr', size(uidt(dt')))(s') OF OK(sn, bl): dt'uidt'valid?(bl, addr') ELSE TRUE ENDCASES
{1}	OK?(memory_read_list(pm'mem)(addr', size(uidt(dt')))(state(q'(s'))))
{2}	OK?(CASES memory_read_list(pm'mem)(addr', size(uidt(dt')))(state(q'(s')))) OF OK(state_1, data): ok_lift(from_byte(dt')(data, addr'))(state_1), Exception(ex_type, state_1): Exception(ex_type, state_1), Fatal: Fatal, Hang: Hang ENDCASES)

Using lemma unchanged_memory_invariant_invariant[State],

Using lemma super_transformer_invariant_next_ok[State],

Installing automatic rewrites from: (singleton!)

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of plain_memory_read_data_q_ok.2.

Q.E.D.

C.172.14

Plain_Mem_Properties_2.plain_memory_read_data_q_same_TCC1

Terse proof for plain_memory_read_data_q_same_TCC1.

plain_memory_read_data_q_same_TCC1:

$\{1\} \quad \forall (\text{addr}: \text{Address}, \text{dt}: (\text{interpreted_data_type?}[\text{Data}]), \text{pm}: \text{Plain_Memory}[\text{State}], s: \text{State}, \\ q: [\text{State} \rightarrow \text{SuperResult}[\text{State}]]): \\ \text{plain_memory?}(\text{pm}) \wedge \\ \text{in_blessed_memory?}(\text{dt}, \text{addr}, (\text{pm}'\text{ro_addr} \cup \text{pm}'\text{rw_addr})) \wedge \\ \text{unchanged_memory_invariant?}(\text{pm}'\text{mem}, \text{pm}'\text{states}, \text{singleton}(q), \\ \text{address_block}(\text{addr}, \text{size}(\text{uidt}(\text{dt})))) \\ \wedge \text{pm}'\text{states}(s) \wedge \text{OK?}(q(s)) \wedge \text{valid_in_mem}(\text{pm}, \text{dt})(\text{addr})(s) \\ \supset \text{OK?}[\text{State}, \text{Data}](\text{read_data}[\text{State}, \text{Data}](\text{pm}, \text{dt})(\text{addr})(\text{state}[\text{State}](q(s))))$
--

Repeatedly Skolemizing and flattening,

Using lemma plain_memory_read_data_q_ok[State, Data],

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of plain_memory_read_data_q_same_TCC1.

Q.E.D.

C.172.15

Plain_Mem_Properties_2.plain_memory_read_data_q_same_TCC2

Terse proof for plain_memory_read_data_q_same_TCC2.

plain_memory_read_data_q_same_TCC2:

$\{1\} \quad \forall (\text{addr}: \text{Address}, \text{dt}: (\text{interpreted_data_type?}[\text{Data}]), \text{pm}: \text{Plain_Memory}[\text{State}], s: \text{State}, \\ q: [\text{State} \rightarrow \text{SuperResult}[\text{State}]]): \\ \text{plain_memory?}(\text{pm}) \wedge \\ \text{in_blessed_memory?}(\text{dt}, \text{addr}, (\text{pm}'\text{ro_addr} \cup \text{pm}'\text{rw_addr})) \wedge \\ \text{unchanged_memory_invariant?}(\text{pm}'\text{mem}, \text{pm}'\text{states}, \text{singleton}(q), \\ \text{address_block}(\text{addr}, \text{size}(\text{uidt}(\text{dt})))) \\ \wedge \text{pm}'\text{states}(s) \wedge \text{OK?}(q(s)) \wedge \text{valid_in_mem}(\text{pm}, \text{dt})(\text{addr})(s) \\ \supset \text{OK?}[\text{State}, \text{Data}](\text{read_data}[\text{State}, \text{Data}](\text{pm}, \text{dt})(\text{addr})(s))$

Repeatedly Skolemizing and flattening,

Using lemma plain_memory_read_data_ok,

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of plain_memory_read_data_q_same_TCC2.

Q.E.D.

C.172.16 Plain_Mem_Properties_2.plain_memory_read_data_q_same

Terse proof for plain_memory_read_data_q_same.

plain_memory_read_data_q_same:

$\{1\} \quad \forall (\text{addr}: \text{Address}, \text{dt}: (\text{interpreted_data_type?}[\text{Data}]), \text{pm}: \text{Plain_Memory}[\text{State}], \text{s}: \text{State}, \\ \text{q}: [\text{State} \rightarrow \text{SuperResult}[\text{State}]]):$ $\text{plain_memory?}(\text{pm}) \wedge$ $\text{in_blessed_memory?}(\text{dt}, \text{addr}, (\text{pm}'\text{ro_addr} \cup \text{pm}'\text{rw_addr})) \wedge$ $\text{unchanged_memory_invariant?}(\text{pm}'\text{mem}, \text{pm}'\text{states}, \text{singleton}(\text{q}), \\ \text{address_block}(\text{addr}, \text{size}(\text{uidt}(\text{dt}))))$ $\wedge \text{pm}'\text{states}(\text{s}) \wedge \text{OK?}(\text{q}(\text{s})) \wedge \text{valid_in_mem}(\text{pm}, \text{dt})(\text{addr})(\text{s})$ \supset $\text{data}(\text{read_data}(\text{pm}, \text{dt})(\text{addr})(\text{state}(\text{q}(\text{s})))) =$ $\text{data}(\text{read_data}(\text{pm}, \text{dt})(\text{addr})(\text{s}))$
--

Repeatedly Skolemizing and flattening,

Using lemma plain_memory_read_data_q_ok,

Simplifying, rewriting, and recording with decision procedures,

Using lemma plain_memory_read_data_ok,

Simplifying, rewriting, and recording with decision procedures,

Installing automatic rewrites from: in_blessed_memory_subset read_data ## expr_2_super
expr_2_super_res ok_lift singleton comp_expr_forget_expr

Using lemma super_transformers_ok_ok,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Expanding the definition of ok_lift,

Expanding the definition of ##,

Using lemma plain_mem_q_read_list[State],

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of plain_memory_read_data_q_same.

Q.E.D.

C.172.17 Plain_Mem_Properties_2.plain_memory_read_write_ok

Terse proof for plain_memory_read_write_ok.

plain_memory_read_write_ok:

$\{1\} \quad \forall (\text{addr}: \text{Address}, \text{data}: \text{Data}, \text{dt}: (\text{interpreted_data_type?}[\text{Data}]), \text{pm}: \text{Plain_Memory}[\text{State}], \\ \text{s}: \text{State}):$ $\text{plain_memory?}(\text{pm}) \wedge \text{pm}'\text{states}(\text{s}) \wedge \text{in_blessed_memory?}(\text{dt}, \text{addr}, \text{pm}'\text{rw_addr}) \supset$ $\text{OK?}((\text{write_data}(\text{pm}, \text{dt})(\text{addr}, \text{data}) \## \text{read_data}(\text{pm}, \text{dt})(\text{addr}))(\text{s}))$

Repeatedly Skolemizing and flattening,

Installing automatic rewrites from: comp_expr_forget_expr ## plain_memory_write_data_ok
plain_memory_read_data_ok subset_bigger_union_right

Simplifying, rewriting, and recording with decision procedures,

Expanding the definition of ##,

Case splitting on OK?(write_data(pm!1'mem, dt!1)(addr!1, data!1)(s!1)),

we get 2 subgoals:

plain_memory_read_write_ok.1:

{-1}	OK?(write_data(pm' mem, dt')(addr', data')(s'))
{-2}	interpreted_data_type?[Data](dt')
{-3}	plain_memory?(pm')
{-4}	pm' states(s')
{-5}	in_blessed_memory?(dt', addr', pm' rw_addr)
{1}	OK?(CASES write_data(pm', dt')(addr', data')(s') OF OK(state, data): read_data(pm', dt')(addr')(state), Exception(ex_type, state): Exception(ex_type, state), Fatal: Fatal, Hang: Hang ENDCASES)

Using lemma plain_memory_read_data_ok,
 Using lemma plain_memory_write_data_valid[State, Data],
 Expanding the definition of in_blessed_memory?,
 Using lemma plain_memory_inv_pred_write_data[State, Data],
 Simplifying, rewriting, and recording with decision procedures,
 Lifting IF-conditions to the top level,
 Simplifying, rewriting, and recording with decision procedures,
 This completes the proof of plain_memory_read_write_ok.1.

plain_memory_read_write_ok.2:

{-1}	interpreted_data_type?[Data](dt')
{-2}	plain_memory?(pm')
{-3}	pm' states(s')
{-4}	in_blessed_memory?(dt', addr', pm' rw_addr)
{1}	OK?(write_data(pm' mem, dt')(addr', data')(s'))
{2}	OK?(CASES write_data(pm', dt')(addr', data')(s') OF OK(state, data): read_data(pm', dt')(addr')(state), Exception(ex_type, state): Exception(ex_type, state), Fatal: Fatal, Hang: Hang ENDCASES)

Hiding formulas: 2,
 Using lemma plain_memory_write_data_ok,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 Expanding the definition of write_data,
 Expanding the definition of write_data,
 which is trivially true.
 This completes the proof of plain_memory_read_write_ok.2.
 Q.E.D.

C.172.18

Plain_Mem_Properties_2.plain_memory_read_write_res_TCC1

Terse proof for plain_memory_read_write_res_TCC1.

plain_memory_read_write_res_TCC1:

$\{1\} \quad \forall (\text{addr: Address, data: Data, dt: (interpreted_data_type? [Data]), pm: Plain_Memory [State], s: State):$ $\text{in_blessed_memory?}(dt, \text{addr}, \text{pm}'\text{rw_addr}) \wedge \text{pm}'\text{states}(s) \wedge \text{plain_memory?}(\text{pm}) \supset$ $\text{OK?}[State, Data]$ $((\#\#[State, Unit, Data]$ $\quad (\text{write_data}[State, Data](\text{pm}, dt)(\text{addr}, \text{data}),$ $\quad \text{read_data}[State, Data](\text{pm}, dt)(\text{addr}))$ $(\text{s}))$

Repeatedly Skolemizing and flattening,

Applying plain_memory_read_write_ok

Repeatedly simplifying with decision procedures, rewriting, propositional reasoning, quantifier instantiation, skolemization, if-lifting and equality replacement,

This completes the proof of plain_memory_read_write_res_TCC1.

Q.E.D.

C.172.19 Plain_Mem_Properties_2.plain_memory_read_write_res

Terse proof for plain_memory_read_write_res.

plain_memory_read_write_res:

$\{1\} \quad \forall (\text{addr: Address, data1: Data, dt: (interpreted_data_type? [Data]), pm: Plain_Memory [State], s: State):$ $\text{plain_memory?}(\text{pm}) \wedge \text{pm}'\text{states}(s) \wedge \text{in_blessed_memory?}(dt, \text{addr}, \text{pm}'\text{rw_addr}) \supset$ $\text{data}((\text{write_data}(\text{pm}, dt)(\text{addr}, \text{data1}) \#\# \text{read_data}(\text{pm}, dt)(\text{addr}))(\text{s})) = \text{data1}$

Repeatedly Skolemizing and flattening,

Using lemma plain_memory_read_write_ok,

Simplifying, rewriting, and recording with decision procedures,

Installing automatic rewrites from: `\#\# write_data read_data ok_lift length_to_byte from_byte_to_byte memory_read_transformers_mono subset_bigger_union_right`

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Expanding the definition of `\#\#`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Expanding the definition of `ok_lift`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Rewriting using `plain_mem_write_list_read_list`, matching in `*` where `pm` gets `pm!1`,

we get 2 subgoals:

C Proof scripts

`plain_memory_read_write_res.1:`

{-1}	OK?(memory_write_list(pm' 'mem)(addr', to_byte(dt')(data1', addr'))(s'))
{-2}	OK?(memory_read_list(pm' 'mem)(addr', size(uidt(dt')) (state(memory_write_list(pm' 'mem) (addr', to_byte(dt')(data1', addr')) (s'))))
{-3}	OK?(OK(state(memory_read_list(pm' 'mem)(addr', size(uidt(dt')) (state(memory_write_list(pm' 'mem) (addr', to_byte(dt')(data1', addr')) (s')))), down(from_byte(dt')(to_byte(dt')(data1', addr'), addr'))))
{-4}	interpreted_data_type?[Data](dt')
{-5}	plain_memory?(pm')
{-6}	pm' 'states(s')
{-7}	in_blessed_memory?(dt', addr', pm' 'rw_addr)
{1}	bottom?(from_byte(dt')(to_byte(dt')(data1', addr'), addr'))
{2}	down(from_byte(dt')(to_byte(dt')(data1', addr'), addr')) = data1'

Rewriting using `from_byte_to_byte`, matching in `*`,

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `plain_memory_read_write_res.1`.

plain_memory_read_write_res.2:

{-1}	OK?(memory_write_list(pm' 'mem)(addr', to_byte(dt')(data1', addr'))(s'))
{-2}	OK?(memory_read_list(pm' 'mem)(addr', size(uidt(dt')) (state(memory_write_list(pm' 'mem) (addr', to_byte(dt')(data1', addr')) (s')))))
{-3}	OK?(OK(state(memory_read_list(pm' 'mem)(addr', size(uidt(dt')) (state(memory_write_list(pm' 'mem) (addr', to_byte(dt')(data1', addr')) (s')))), down(from_byte(dt') (data(memory_read_list(pm' 'mem)(addr', size(uidt(dt')) (state(memory_write_list(pm' 'mem) (addr', to_byte(dt') (data1', addr')) (s')))), addr'))))
{-4}	interpreted_data_type?[Data](dt')
{-5}	plain_memory?(pm')
{-6}	pm' 'states(s')
{-7}	in_blessed_memory?(dt', addr', pm' 'rw_addr)
<hr/>	
{1}	(address_block(addr', size(uidt(dt'))) \subseteq pm' 'rw_addr)
{2}	bottom?(from_byte(dt') (data(memory_read_list(pm' 'mem)(addr', size(uidt(dt')) (state(memory_write_list(pm' 'mem) (addr', to_byte(dt') (data1', addr')) (s')))), addr'))
{3}	down(from_byte(dt') (data(memory_read_list(pm' 'mem)(addr', size(uidt(dt')) (state(memory_write_list(pm' 'mem) (addr', to_byte(dt')(data1', addr')) (s')))), addr')) = data1'

Expanding the definition of in_blessed_memory?,
which is trivially true.

This completes the proof of plain_memory_read_write_res.2.

Q.E.D.

C.172.20

Plain_Mem_Properties_2.plain_memory_unchanged_memory_ok_result

Terse proof for plain_memory_unchanged_memory_ok_result.

plain_memory_unchanged_memory_ok_result:

{1}	\forall (addresses: PRED[Address], pm: Plain_Memory[State], d: Data): plain_memory?(pm) \wedge (addresses \subseteq (pm'ro_addr \cup pm'rw_addr)) \supset unchanged_memory_invariant?(pm'mem, pm'states, singleton(expr_2_super[State, Data](ok_result(d))), addresses)
-----	---

Expanding the definition of unchanged_memory_invariant?,

Repeatedly Skolemizing and flattening,

Installing automatic rewrites from: (expr_2_super! expr_2_super_res! ok_result!)

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 2 subgoals:

plain_memory_unchanged_memory_ok_result.1:

{-1}	plain_memory?(pm')
{-2}	(addresses' \subseteq (pm'ro_addr \cup pm'rw_addr))
{1}	\forall (s: State, q: [State \rightarrow SuperResult[State]], a: Address): pm'states(s) \wedge singleton(expr_2_super[State, Data](ok_result(d'))(q) \wedge addresses'(a) \wedge OK?(q(s)) \wedge OK?(memory_read(pm'mem)(a)(s)) \wedge OK?(memory_read(pm'mem)(a)(state(q(s)))) \supset data(memory_read(pm'mem)(a)(state(q(s)))) = data(memory_read(pm'mem)(a)(s))

Repeatedly Skolemizing and flattening,

Expanding the definition of singleton,

Replacing using formula -4,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of plain_memory_unchanged_memory_ok_result.1.

plain_memory_unchanged_memory_ok_result.2:

{-1}	plain_memory?(pm')
{-2}	(addresses' \subseteq (pm'ro_addr \cup pm'rw_addr))
{1}	transformer_invariant?(pm'states, singleton(expr_2_super[State, Data](ok_result(d'))))

Expanding the definition of transformer_invariant?,

Repeatedly Skolemizing and flattening,

Expanding the definition of singleton,

Replacing using formula -4,

Expanding the definition of result_pred,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of plain_memory_unchanged_memory_ok_result.2.

Q.E.D.

C.173 Proofs for Plain_Mem_Properties_3 (plain_memory.pvs)

C.173.1 Plain_Mem_Properties_3.plain_memory_read_write_other_ok

Terse proof for plain_memory_read_write_other_ok.

plain_memory_read_write_other_ok:

<pre>{1} ∃ (addr1, addr2: Address, data1: Data1, dt1: (interpreted_data_type?[Data1]), dt2: (interpreted_data_type?[Data2]), pm: Plain_Memory[State], s: State): plain_memory?(pm) ∧ pm' states(s) ∧ in_blessed_memory?(dt1, addr1, pm'rw_addr) ∧ in_blessed_memory?(dt2, addr2, (pm'ro_addr ∪ pm'rw_addr)) ∧ blocks_disjoint?(addr1, size(uidt(dt1)), addr2, size(uidt(dt2))) ∧ valid_in_mem(pm, dt2)(addr2)(s) ⊃ OK?(write_data(pm, dt1)(addr1, data1) ## read_data(pm, dt2)(addr2))(s)</pre>
--

Repeatedly Skolemizing and flattening,

Expanding the definition of ##,

Expanding the definition of ##,

Case splitting on OK?(write_data(pm!1, dt1!1)(addr1!1, data1!1)(s!1)),

we get 2 subgoals:

plain_memory_read_write_other_ok.1:

<pre>{-1} OK?(write_data(pm', dt1')(addr1', data1')(s')) {-2} interpreted_data_type?[Data1](dt1') {-3} interpreted_data_type?[Data2](dt2') {-4} plain_memory?(pm') {-5} pm' states(s') {-6} in_blessed_memory?(dt1', addr1', pm'rw_addr) {-7} in_blessed_memory?(dt2', addr2', (pm'ro_addr ∪ pm'rw_addr)) {-8} blocks_disjoint?(addr1', size(uidt(dt1')), addr2', size(uidt(dt2'))) {-9} valid_in_mem(pm', dt2')(addr2')(s')</pre>
<pre>{1} OK?(CASES write_data(pm', dt1')(addr1', data1')(s') OF OK(state, data): read_data(pm', dt2')(addr2')(state), Exception(ex_type, state): Exception(ex_type, state), Fatal: Fatal, Hang: Hang ENDCASES)</pre>

Simplifying, rewriting, and recording with decision procedures,

Using lemma plain_memory_read_data_q_ok[State, Data2],

Installing automatic rewrites from: (expr_2_super! expr_2_super_res!)

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Using lemma plain_memory_unchanged_memory_invariant_write_data[State, Data1],

Simplifying, rewriting, and recording with decision procedures,

Using lemma unchanged_memory_invariant_mono,

Simplifying, rewriting, and recording with decision procedures,

Rewriting using subset_reflexive, matching in *,

Expanding the definition of in_blessed_memory?,

Keeping (-3 -9 -10 1) and hiding *,

Using lemma blocks_in_larger_set,

C Proof scripts

Simplifying, rewriting, and recording with decision procedures,

Rewriting using `subset_bigger_union_right`, matching in `*`,

This completes the proof of `plain_memory_read_write_other_ok.1`.

`plain_memory_read_write_other_ok.2`:

{-1}	<code>interpreted_data_type?[Data1](dt1')</code>
{-2}	<code>interpreted_data_type?[Data2](dt2')</code>
{-3}	<code>plain_memory?(pm')</code>
{-4}	<code>pm' states(s')</code>
{-5}	<code>in_blessed_memory?(dt1', addr1', pm' rw_addr)</code>
{-6}	<code>in_blessed_memory?(dt2', addr2', (pm' ro_addr ∪ pm' rw_addr))</code>
{-7}	<code>blocks_disjoint?(addr1', size(uidt(dt1')), addr2', size(uidt(dt2')))</code>
{-8}	<code>valid_in_mem(pm', dt2')(addr2')(s')</code>
{1}	<code>OK?(write_data(pm', dt1')(addr1', data1')(s'))</code>
{2}	<code>OK?(CASES write_data(pm', dt1')(addr1', data1')(s') OF</code> <code> OK(state, data): read_data(pm', dt2')(addr2')(state),</code> <code> Exception(ex_type, state): Exception(ex_type, state),</code> <code> Fatal: Fatal,</code> <code> Hang: Hang</code> <code>ENDCASES)</code>

Rewriting using `plain_memory_write_data_ok`, matching in `*`,

This completes the proof of `plain_memory_read_write_other_ok.2`.

Q.E.D.

C.173.2

Plain_Mem_Properties_3.plain_memory_read_write_other_res_TCC1

Terse proof for `plain_memory_read_write_other_res_TCC1`.

`plain_memory_read_write_other_res_TCC1`:

{1}	\forall (<code>addr1, addr2: Address, data1: Data1, dt1: (interpreted_data_type?[Data1]),</code> <code>dt2: (interpreted_data_type?[Data2]), pm: Plain_Memory[State], s: State):</code> <code>valid_in_mem(pm, dt2)(addr2)(s) ∧</code> <code>blocks_disjoint?(addr1, size(uidt(dt1)), addr2, size(uidt(dt2))) ∧</code> <code>in_blessed_memory?(dt2, addr2, (pm' ro_addr ∪ pm' rw_addr)) ∧</code> <code>in_blessed_memory?(dt1, addr1, pm' rw_addr) ∧ pm' states(s) ∧ plain_memory?(pm)</code> <code>⊃</code> <code>OK?[State, Data2]</code> <code>((##[State, Unit, Data2]</code> <code> (write_data[State, Data1](pm, dt1)(addr1, data1),</code> <code> read_data[State, Data2](pm, dt2)(addr2)))</code> <code>(s))</code>
-----	--

Repeatedly Skolemizing and flattening,

Using lemma `plain_memory_read_write_other_ok`,

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `plain_memory_read_write_other_res_TCC1`.

Q.E.D.

C.173.3**Plain_Mem_Properties_3.plain_memory_read_write_other_res_TCC2**

Terse proof for plain_memory_read_write_other_res_TCC2.

plain_memory_read_write_other_res_TCC2:

$$\{1\} \quad \forall (\text{addr1}, \text{addr2}: \text{Address}, \text{dt1}: (\text{interpreted_data_type?}[\text{Data1}]), \\ \text{dt2}: (\text{interpreted_data_type?}[\text{Data2}]), \text{pm}: \text{Plain_Memory}[\text{State}], \text{s}: \text{State}): \\ \text{valid_in_mem}(\text{pm}, \text{dt2})(\text{addr2})(\text{s}) \wedge \\ \text{blocks_disjoint?}(\text{addr1}, \text{size}(\text{uidt}(\text{dt1})), \text{addr2}, \text{size}(\text{uidt}(\text{dt2}))) \wedge \\ \text{in_blessed_memory?}(\text{dt2}, \text{addr2}, (\text{pm}'\text{ro_addr} \cup \text{pm}'\text{rw_addr})) \wedge \\ \text{in_blessed_memory?}(\text{dt1}, \text{addr1}, \text{pm}'\text{rw_addr}) \wedge \text{pm}'\text{states}(\text{s}) \wedge \text{plain_memory?}(\text{pm}) \\ \supset \text{OK?}[\text{State}, \text{Data2}](\text{read_data}[\text{State}, \text{Data2}](\text{pm}, \text{dt2})(\text{addr2})(\text{s}))$$

Repeatedly Skolemizing and flattening,

Using lemma plain_memory_read_data_ok[State, Data2],

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of plain_memory_read_write_other_res_TCC2.

Q.E.D.

C.173.4**Plain_Mem_Properties_3.plain_memory_read_write_other_res**

Terse proof for plain_memory_read_write_other_res.

plain_memory_read_write_other_res:

$$\{1\} \quad \forall (\text{addr1}, \text{addr2}: \text{Address}, \text{data1}: \text{Data1}, \text{dt1}: (\text{interpreted_data_type?}[\text{Data1}]), \\ \text{dt2}: (\text{interpreted_data_type?}[\text{Data2}]), \text{pm}: \text{Plain_Memory}[\text{State}], \text{s}: \text{State}): \\ \text{plain_memory?}(\text{pm}) \wedge \\ \text{pm}'\text{states}(\text{s}) \wedge \\ \text{in_blessed_memory?}(\text{dt1}, \text{addr1}, \text{pm}'\text{rw_addr}) \wedge \\ \text{in_blessed_memory?}(\text{dt2}, \text{addr2}, (\text{pm}'\text{ro_addr} \cup \text{pm}'\text{rw_addr})) \wedge \\ \text{blocks_disjoint?}(\text{addr1}, \text{size}(\text{uidt}(\text{dt1})), \text{addr2}, \text{size}(\text{uidt}(\text{dt2}))) \wedge \\ \text{valid_in_mem}(\text{pm}, \text{dt2})(\text{addr2})(\text{s}) \\ \supset \\ \text{data}(\text{write_data}(\text{pm}, \text{dt1})(\text{addr1}, \text{data1}) \text{##} \text{read_data}(\text{pm}, \text{dt2})(\text{addr2}))(\text{s}) = \\ \text{data}(\text{read_data}(\text{pm}, \text{dt2})(\text{addr2})(\text{s}))$$

Installing automatic rewrites from: (## expr_2_super expr_2_super_res)

Repeatedly Skolemizing and flattening,

Case splitting on OK?(write_data(pm!1, dt1!1)(addr1!1, data1!1)(s!1)),

we get 2 subgoals:

plain_memory_read_write_other_res.1:

{-1}	OK?(write_data(pm', dt1')(addr1', data1')(s'))
{-2}	interpreted_data_type?[Data1](dt1')
{-3}	interpreted_data_type?[Data2](dt2')
{-4}	plain_memory?(pm')
{-5}	pm' 'states(s')
{-6}	in_blessed_memory?(dt1', addr1', pm' 'rw_addr)
{-7}	in_blessed_memory?(dt2', addr2', (pm' 'ro_addr \cup pm' 'rw_addr))
{-8}	blocks_disjoint?(addr1', size(uidt(dt1')), addr2', size(uidt(dt2')))
{-9}	valid_in_mem(pm', dt2')(addr2')(s')
{1}	data((write_data(pm', dt1')(addr1', data1') ## read_data(pm', dt2')(addr2')) (s')) = data(read_data(pm', dt2')(addr2')(s'))

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Using lemma plain_memory_read_data_q_same[State, Data2],
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Using lemma plain_memory_unchanged_memory_invariant_write_data[State, Data1],
Simplifying, rewriting, and recording with decision procedures,
Using lemma unchanged_memory_invariant_mono,
Simplifying, rewriting, and recording with decision procedures,
Rewriting using subset_reflexive, matching in *,
Expanding the definition of in_blessed_memory?,
Keeping (-3 -9 -10 1) and hiding *,
Using lemma blocks_in_larger_set,
Simplifying, rewriting, and recording with decision procedures,
Rewriting using subset_bigger_union_right, matching in *,
This completes the proof of plain_memory_read_write_other_res.1.

plain_memory_read_write_other_res.2:

{-1}	interpreted_data_type?[Data1](dt1')
{-2}	interpreted_data_type?[Data2](dt2')
{-3}	plain_memory?(pm')
{-4}	pm' 'states(s')
{-5}	in_blessed_memory?(dt1', addr1', pm' 'rw_addr)
{-6}	in_blessed_memory?(dt2', addr2', (pm' 'ro_addr \cup pm' 'rw_addr))
{-7}	blocks_disjoint?(addr1', size(uidt(dt1')), addr2', size(uidt(dt2')))
{-8}	valid_in_mem(pm', dt2')(addr2')(s')
{1}	OK?(write_data(pm', dt1')(addr1', data1')(s'))
{2}	data((write_data(pm', dt1')(addr1', data1') ## read_data(pm', dt2')(addr2')) (s')) = data(read_data(pm', dt2')(addr2')(s'))

Rewriting using plain_memory_write_data_ok[State, Data1], matching in *,
This completes the proof of plain_memory_read_write_other_res.2.
Q.E.D.

C.173.5 Plain_Mem_Properties_3.plain_memory_read_read_ok

Terse proof for plain_memory_read_read_ok.

plain_memory_read_read_ok:

{1}	$\forall (\text{addr1}, \text{addr2}: \text{Address}, \text{dt1}: (\text{interpreted_data_type?}[\text{Data1}]),$ $\text{dt2}: (\text{interpreted_data_type?}[\text{Data2}]), \text{pm}: \text{Plain_Memory}[\text{State}], \text{s}: \text{State}):$ $\text{plain_memory?}(\text{pm}) \wedge$ $\text{pm}'\text{states}(\text{s}) \wedge$ $\text{in_blessed_memory?}(\text{dt1}, \text{addr1}, (\text{pm}'\text{ro_addr} \cup \text{pm}'\text{rw_addr})) \wedge$ $\text{in_blessed_memory?}(\text{dt2}, \text{addr2}, (\text{pm}'\text{ro_addr} \cup \text{pm}'\text{rw_addr})) \wedge$ $\text{valid_in_mem}(\text{pm}, \text{dt1})(\text{addr1})(\text{s}) \wedge \text{valid_in_mem}(\text{pm}, \text{dt2})(\text{addr2})(\text{s})$ $\supset \text{OK?}((\text{read_data}(\text{pm}, \text{dt1})(\text{addr1}) \text{ ## } \text{read_data}(\text{pm}, \text{dt2})(\text{addr2}))(\text{s}))$
-----	--

Repeatedly Skolemizing and flattening,

Case splitting on $\text{OK?}(\text{read_data}(\text{pm!1}, \text{dt1!1})(\text{addr1!1})(\text{s!1}))$,

we get 2 subgoals:

plain_memory_read_read_ok.1:

{-1}	$\text{OK?}(\text{read_data}(\text{pm}', \text{dt1}')(\text{addr1}')(\text{s}'))$
{-2}	$\text{interpreted_data_type?}[\text{Data1}](\text{dt1}')$
{-3}	$\text{interpreted_data_type?}[\text{Data2}](\text{dt2}')$
{-4}	$\text{plain_memory?}(\text{pm}')$
{-5}	$\text{pm}'\text{states}(\text{s}')$
{-6}	$\text{in_blessed_memory?}(\text{dt1}', \text{addr1}', (\text{pm}'\text{ro_addr} \cup \text{pm}'\text{rw_addr}))$
{-7}	$\text{in_blessed_memory?}(\text{dt2}', \text{addr2}', (\text{pm}'\text{ro_addr} \cup \text{pm}'\text{rw_addr}))$
{-8}	$\text{valid_in_mem}(\text{pm}', \text{dt1}')(\text{addr1}')(\text{s}')$
{-9}	$\text{valid_in_mem}(\text{pm}', \text{dt2}')(\text{addr2}')(\text{s}')$
{1}	$\text{OK?}((\text{read_data}(\text{pm}', \text{dt1}')(\text{addr1}') \text{ ## } \text{read_data}(\text{pm}', \text{dt2}')(\text{addr2}'))(\text{s}'))$

Installing automatic rewrites from: $(\text{expr_2_super! expr_2_super_res! ##!})$

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Using lemma $\text{plain_memory_read_data_q_ok}[\text{State}, \text{Data2}]$,

Instantiating the top quantifier in -1 with the terms: $\text{expr_2_super}(\text{read_data}(\text{pm}', \text{dt1}')(\text{addr1}'))$,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Using lemma $\text{plain_memory_unchanged_memory_invariant_read_data}[\text{State}, \text{Data1}]$,

Using lemma $\text{unchanged_memory_invariant_mono}$,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 2 subgoals:

C Proof scripts

plain_memory_read_read_ok.1.1:

{-1}	unchanged_memory_invariant?(pm' mem, pm' states, singleton(expr_2_super(read_data(pm', dt1')(addr1'))), (pm' ro_addr \cup pm' rw_addr))
{-2}	plain_memory?(pm')
{-3}	in_blessed_memory?(dt1', addr1', (pm' ro_addr \cup pm' rw_addr))
{-4}	in_blessed_memory?(dt2', addr2', (pm' ro_addr \cup pm' rw_addr))
{-5}	pm' states(s')
{-6}	valid_in_mem(pm', dt2')(addr2')(s')
{-7}	OK?(read_data(pm', dt1')(addr1')(s'))
{-8}	interpreted_data_type?[Data1](dt1')
{-9}	interpreted_data_type?[Data2](dt2')
{-10}	valid_in_mem(pm', dt1')(addr1')(s')
{1}	(address_block(addr2', size(uidt(dt2')))) \subseteq (pm' ro_addr \cup pm' rw_addr)
{2}	unchanged_memory_invariant?(pm' mem, pm' states, singleton(expr_2_super(read_data(pm', dt1')(addr1'))), address_block(addr2', size(uidt(dt2'))))
{3}	OK?(read_data(pm', dt2')(addr2')(state(read_data(pm', dt1')(addr1')(s'))))

Expanding the definition of in_blessed_memory?,

which is trivially true.

This completes the proof of plain_memory_read_read_ok.1.1.

plain_memory_read_read_ok.1.2:

{-1}	unchanged_memory_invariant?(pm' mem, pm' states, singleton(expr_2_super(read_data(pm', dt1')(addr1'))), (pm' ro_addr \cup pm' rw_addr))
{-2}	plain_memory?(pm')
{-3}	in_blessed_memory?(dt1', addr1', (pm' ro_addr \cup pm' rw_addr))
{-4}	in_blessed_memory?(dt2', addr2', (pm' ro_addr \cup pm' rw_addr))
{-5}	pm' states(s')
{-6}	valid_in_mem(pm', dt2')(addr2')(s')
{-7}	OK?(read_data(pm', dt1')(addr1')(s'))
{-8}	interpreted_data_type?[Data1](dt1')
{-9}	interpreted_data_type?[Data2](dt2')
{-10}	valid_in_mem(pm', dt1')(addr1')(s')
{1}	(singleton(expr_2_super(read_data(pm', dt1')(addr1')))) \subseteq singleton(expr_2_super(read_data(pm',
{2}	unchanged_memory_invariant?(pm' mem, pm' states, singleton(expr_2_super(read_data(pm', dt1')(addr1'))), address_block(addr2', size(uidt(dt2'))))
{3}	OK?(read_data(pm', dt2')(addr2')(state(read_data(pm', dt1')(addr1')(s'))))

Rewriting using subset_reflexive, matching in *,

This completes the proof of plain_memory_read_read_ok.1.2.

plain_memory_read_read_ok.2:

{-1}	interpreted_data_type?[Data1](dt1')
{-2}	interpreted_data_type?[Data2](dt2')
{-3}	plain_memory?(pm')
{-4}	pm' 'states(s')
{-5}	in_blessed_memory?(dt1', addr1', (pm'ro_addr ∪ pm'rw_addr))
{-6}	in_blessed_memory?(dt2', addr2', (pm'ro_addr ∪ pm'rw_addr))
{-7}	valid_in_mem(pm', dt1')(addr1')(s')
{-8}	valid_in_mem(pm', dt2')(addr2')(s')
<hr/>	
{1}	OK?(read_data(pm', dt1')(addr1')(s'))
{2}	OK?((read_data(pm', dt1')(addr1') ## read_data(pm', dt2')(addr2'))(s'))

Rewriting using plain_memory_read_data_ok[State, Data1], matching in *,

This completes the proof of plain_memory_read_read_ok.2.

Q.E.D.

C.173.6 Plain_Mem_Properties_3.plain_memory_read_read_TCC1

Terse proof for plain_memory_read_read_TCC1.

plain_memory_read_read_TCC1:

{1}	$\forall (\text{addr1}, \text{addr2}: \text{Address}, \text{dt1}: (\text{interpreted_data_type?}[\text{Data1}]),$ $\text{dt2}: (\text{interpreted_data_type?}[\text{Data2}]), \text{pm}: \text{Plain_Memory}[\text{State}], \text{s}: \text{State}):$ $\text{valid_in_mem}(\text{pm}, \text{dt2})(\text{addr2})(\text{s}) \wedge$ $\text{valid_in_mem}(\text{pm}, \text{dt1})(\text{addr1})(\text{s}) \wedge$ $\text{in_blessed_memory?}(\text{dt2}, \text{addr2}, (\text{pm}'\text{ro_addr} \cup \text{pm}'\text{rw_addr})) \wedge$ $\text{in_blessed_memory?}(\text{dt1}, \text{addr1}, (\text{pm}'\text{ro_addr} \cup \text{pm}'\text{rw_addr})) \wedge$ $\text{pm}'\text{states}(\text{s}) \wedge \text{plain_memory?}(\text{pm})$ \supset $\text{OK?}[\text{State}, \text{Data2}]$ $((\text{##}[\text{State}, \text{Data1}, \text{Data2}]$ $\quad (\text{read_data}[\text{State}, \text{Data1}](\text{pm}, \text{dt1})(\text{addr1}),$ $\quad \text{read_data}[\text{State}, \text{Data2}](\text{pm}, \text{dt2})(\text{addr2})))$ $(\text{s}))$
-----	---

Repeatedly Skolemizing and flattening,

Using lemma plain_memory_read_read_ok,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of plain_memory_read_read_TCC1.

Q.E.D.

C.173.7 Plain_Mem_Properties_3.plain_memory_read_read_TCC2

Terse proof for plain_memory_read_read_TCC2.

plain_memory_read_read_TCC2:

$\{1\} \quad \forall (\text{addr1}, \text{addr2}: \text{Address}, \text{dt1}: (\text{interpreted_data_type?}[\text{Data1}]), \\ \text{dt2}: (\text{interpreted_data_type?}[\text{Data2}]), \text{pm}: \text{Plain_Memory}[\text{State}], s: \text{State}): \\ \text{valid_in_mem}(\text{pm}, \text{dt2})(\text{addr2})(s) \wedge \\ \text{valid_in_mem}(\text{pm}, \text{dt1})(\text{addr1})(s) \wedge \\ \text{in_blessed_memory?}(\text{dt2}, \text{addr2}, (\text{pm}'\text{ro_addr} \cup \text{pm}'\text{rw_addr})) \wedge \\ \text{in_blessed_memory?}(\text{dt1}, \text{addr1}, (\text{pm}'\text{ro_addr} \cup \text{pm}'\text{rw_addr})) \wedge \\ \text{pm}'\text{states}(s) \wedge \text{plain_memory?}(\text{pm}) \\ \supset \text{OK?}[\text{State}, \text{Data2}](\text{read_data}[\text{State}, \text{Data2}](\text{pm}, \text{dt2})(\text{addr2})(s))$

Repeatedly Skolemizing and flattening,

Using lemma plain_memory_read_data_ok[State, Data2],

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of plain_memory_read_read_TCC2.

Q.E.D.

C.173.8 Plain_Mem_Properties_3.plain_memory_read_read

Terse proof for plain_memory_read_read.

plain_memory_read_read:

$\{1\} \quad \forall (\text{addr1}, \text{addr2}: \text{Address}, \text{dt1}: (\text{interpreted_data_type?}[\text{Data1}]), \\ \text{dt2}: (\text{interpreted_data_type?}[\text{Data2}]), \text{pm}: \text{Plain_Memory}[\text{State}], s: \text{State}): \\ \text{plain_memory?}(\text{pm}) \wedge \\ \text{pm}'\text{states}(s) \wedge \\ \text{in_blessed_memory?}(\text{dt1}, \text{addr1}, (\text{pm}'\text{ro_addr} \cup \text{pm}'\text{rw_addr})) \wedge \\ \text{in_blessed_memory?}(\text{dt2}, \text{addr2}, (\text{pm}'\text{ro_addr} \cup \text{pm}'\text{rw_addr})) \wedge \\ \text{valid_in_mem}(\text{pm}, \text{dt1})(\text{addr1})(s) \wedge \text{valid_in_mem}(\text{pm}, \text{dt2})(\text{addr2})(s) \\ \supset \\ \text{data}((\text{read_data}(\text{pm}, \text{dt1})(\text{addr1}) \text{ ## } \text{read_data}(\text{pm}, \text{dt2})(\text{addr2}))(s)) = \\ \text{data}(\text{read_data}(\text{pm}, \text{dt2})(\text{addr2})(s))$
--

Repeatedly Skolemizing and flattening,

Using lemma plain_memory_read_data_ok[State, Data1],

Installing automatic rewrites from: (expr_2_super! expr_2_super_res! ##!)

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Using lemma plain_memory_read_data_q_same,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Using lemma plain_memory_unchanged_memory_invariant_read_data[State, Data1],

Using lemma unchanged_memory_invariant_mono,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 2 subgoals:

plain_memory_read_read.1:

{-1}	unchanged_memory_invariant?(pm' mem, pm' states, singleton(expr_2_super(read_data(pm', dt1')(addr1'))), (pm'ro_addr \cup pm'rw_addr))
{-2}	plain_memory?(pm')
{-3}	in_blessed_memory?(dt1', addr1', (pm'ro_addr \cup pm'rw_addr))
{-4}	in_blessed_memory?(dt2', addr2', (pm'ro_addr \cup pm'rw_addr))
{-5}	pm' states(s')
{-6}	valid_in_mem(pm', dt2')(addr2')(s')
{-7}	valid_in_mem(pm', dt1')(addr1')(s')
{-8}	OK?(read_data(pm', dt1')(addr1')(s'))
{1}	(address_block(addr2', size(uidt(dt2')))) \subseteq (pm'ro_addr \cup pm'rw_addr)
{2}	unchanged_memory_invariant?(pm' mem, pm' states, singleton(expr_2_super(read_data(pm', dt1')(addr1'))), address_block(addr2', size(uidt(dt2'))))
{3}	data(read_data(pm', dt2')(addr2')(state(read_data(pm', dt1')(addr1')(s')))) = data(read_data(pm', dt2')(addr2')(s'))

Expanding the definition of in_blessed_memory?,

which is trivially true.

This completes the proof of plain_memory_read_read.1.

plain_memory_read_read.2:

{-1}	unchanged_memory_invariant?(pm' mem, pm' states, singleton(expr_2_super(read_data(pm', dt1')(addr1'))), (pm'ro_addr \cup pm'rw_addr))
{-2}	plain_memory?(pm')
{-3}	in_blessed_memory?(dt1', addr1', (pm'ro_addr \cup pm'rw_addr))
{-4}	in_blessed_memory?(dt2', addr2', (pm'ro_addr \cup pm'rw_addr))
{-5}	pm' states(s')
{-6}	valid_in_mem(pm', dt2')(addr2')(s')
{-7}	valid_in_mem(pm', dt1')(addr1')(s')
{-8}	OK?(read_data(pm', dt1')(addr1')(s'))
{1}	(singleton(expr_2_super(read_data(pm', dt1')(addr1')))) \subseteq singleton(expr_2_super(read_data(pm', dt1')(addr1')))
{2}	unchanged_memory_invariant?(pm' mem, pm' states, singleton(expr_2_super(read_data(pm', dt1')(addr1'))), address_block(addr2', size(uidt(dt2'))))
{3}	data(read_data(pm', dt2')(addr2')(state(read_data(pm', dt1')(addr1')(s')))) = data(read_data(pm', dt2')(addr2')(s'))

Rewriting using subset_reflexive, matching in *,

This completes the proof of plain_memory_read_read.2.

Q.E.D.

C.173.9

Plain_Mem_Properties_3.plain_memory_unchanged_composition

Terse proof for plain_memory_unchanged_composition.

C Proof scripts

plain_memory_unchanged_composition:

<pre> {1} ∀ (addresses: PRED[Address], pm: Plain_Memory[State], q: [State → ExprResult[State, Data1]], r: [Data1 → [State → ExprResult[State, Data2]]], P: PRED[Data1]): (plain_memory?(pm) ∧ (addresses ⊆ (pm'ro_addr ∪ pm'rw_addr)) ∧ (∀ (s: (pm'states)): OK?(q(s)) ⊃ P(data(q(s)))) ∧ unchanged_memory_invariant?(pm'mem, pm'states, singleton(expr_2_super(q)), addresses) ∧ (∀ (d: (P')): unchanged_memory_invariant?(pm'mem, pm'states, singleton(expr_2_super(r(d))), addresses))) ⊃ unchanged_memory_invariant?(pm'mem, pm'states, singleton(expr_2_super(q ## r)), addresses) </pre>
--

Repeatedly Skolemizing and flattening,

Using lemma `fexpr_unchanged_memory_invariant_composition`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 2 subgoals:

plain_memory_unchanged_composition.1:

<pre> {-1} unchanged_memory_invariant?(pm'mem, pm'states, singleton(expr_2_super(q')), addresses') {-2} ∀ (s: (pm'states)): OK?(q'(s)) ⊃ P'(data(q'(s))) {-3} ∀ (d: (P')): unchanged_memory_invariant?(pm'mem, pm'states, singleton(expr_2_super(r'(d))), addresses') {-4} plain_memory?(pm') {-5} (addresses' ⊆ (pm'ro_addr ∪ pm'rw_addr)) </pre> <hr/> <pre> {1} transformers_ok?(pm'states, memory_read_transformers(pm'mem, addresses')) {2} unchanged_memory_invariant?(pm'mem, pm'states, singleton(expr_2_super(q' ## r')), addresses') </pre>
--

Keeping (-4 -5 1) and hiding *,

Using lemma `plain_memory_transformers_ok_read_ro_rw`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Rewriting using `transformers_ok_mono_transformers`, matching in * where `states` gets `pm!1'states`, `transformers_1` gets `memory_read_transformers(pm!1'mem, addresses!1)`, `transformers_2` gets `memory_read_transformers(pm!1'mem, union(pm!1'ro_addr, pm!1'rw_addr))`,

Rewriting using `memory_read_transformers_mono`, matching in *,

This completes the proof of `plain_memory_unchanged_composition.1`.

plain_memory_unchanged_composition.2:

<p>{-1} unchanged_memory_invariant?(pm' mem, pm' states, singleton(expr_2_super(q')), addresses')</p> <p>{-2} $\forall (s: (pm' states)): OK?(q'(s)) \supset P'(data(q'(s)))$</p> <p>{-3} $\forall (d: (P')):$ unchanged_memory_invariant?(pm' mem, pm' states, singleton(expr_2_super(r'(d))), addresses')</p> <p>{-4} plain_memory?(pm')</p> <p>{-5} (addresses' \subseteq (pm' ro_addr \cup pm' rw_addr))</p>	<hr style="border: 0.5px solid black;"/> <p>{1} singleton(expr_2_super(q'))(expr_2_super(q'))</p> <p>{2} unchanged_memory_invariant?(pm' mem, pm' states, singleton(expr_2_super(q' ## r')), addresses')</p>
---	--

Expanding the definition of singleton, which is trivially true.

This completes the proof of plain_memory_unchanged_composition.2.
 Q.E.D.

C.174 Proofs for Plain_Mem_Rewrites (plain_memory_rewrites.pvs)

This theory contains no provable formal statements.

C.175 Proofs for Plain_Mem_Rewrites1 (plain_memory_rewrites.pvs)

C.175.1 Plain_Mem_Rewrites1.pm_q_prop_TCC1

Terse proof for pm_q_prop_TCC1.

pm_q_prop_TCC1:

<p>{1} $\forall (res: StmtResult[State]):$ OK?(res) \supset OK?[State](res) \vee Break?[State](res) \vee Continue?[State](res) \vee Return?[State](res) \vee Switch?[State](res) \vee Default?[State](res) \vee Exception?[State](res)</p>	<hr style="border: 0.5px solid black;"/>
--	--

Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of pm_q_prop_TCC1.
 Q.E.D.

C.175.2 Plain_Mem_Rewrites1.pm_q_prop_ok_stmt

Terse proof for pm_q_prop_ok_stmt.

pm_q_prop_ok_stmt:

$$\{1\} \quad \forall (pm: \text{Plain_Memory}[\text{State}], s: \text{State}, q: [\text{State} \rightarrow \text{StmtResult}[\text{State}]]):$$

$$\text{plain_memory?}(pm) \wedge \text{pm_q_prop}(pm)(q(s)) \supset \text{OK?}(q(s))$$

Expanding the definition of pm_q_prop,
 Repeatedly Skolemizing and flattening,
 This completes the proof of pm_q_prop_ok_stmt.
 Q.E.D.

C.176 Proofs for Plain_Mem_Rewrites2 (plain_memory_rewrites.pvs)

C.176.1 Plain_Mem_Rewrites2.in_blessed_memory_rw_ro

Terse proof for in_blessed_memory_rw_ro.

in_blessed_memory_rw_ro:

$$\{1\} \quad \forall (addr: \text{Address}, dt: (\text{interpreted_data_type?}[\text{Data}]), pm: \text{Plain_Memory}[\text{State}]):$$

$$\text{in_blessed_memory?}(dt, addr, pm\text{'rw_addr}) \supset$$

$$\text{in_blessed_memory?}(dt, addr, (pm\text{'ro_addr} \cup pm\text{'rw_addr}))$$

Repeatedly Skolemizing and flattening,
 Expanding the definition of in_blessed_memory?,
 Rewriting using subset_bigger_union_right, matching in *,
 This completes the proof of in_blessed_memory_rw_ro.
 Q.E.D.

C.176.2 Plain_Mem_Rewrites2.pm_q_prop_TCC1

Terse proof for pm_q_prop_TCC1.

pm_q_prop_TCC1:

$$\{1\} \quad \forall (res: \text{ExprResult}[\text{State}, \text{Data}]):$$

$$\text{OK?}(res) \supset \text{OK?}[\text{State}, \text{Data}](res) \vee \text{Exception?}[\text{State}, \text{Data}](res)$$

Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of pm_q_prop_TCC1.
 Q.E.D.

C.176.3 Plain_Mem_Rewrites2.pm_q_prop_ok_expr

Terse proof for pm_q_prop_ok_expr.

pm_q_prop_ok_expr:

$$\{1\} \quad \forall (pm: \text{Plain_Memory}[\text{State}], s: \text{State}, q: [\text{State} \rightarrow \text{ExprResult}[\text{State}, \text{Data}]]):$$

$$\text{plain_memory?}(pm) \wedge \text{pm_q_prop}(pm)(q(s)) \supset \text{OK?}(q(s))$$

Expanding the definition of pm_q_prop,
 Repeatedly Skolemizing and flattening,

This completes the proof of pm_q_prop_ok_expr.
Q.E.D.

C.176.4 Plain_Mem_Rewrites2.pm_q_prop_single_read

Terse proof for pm_q_prop_single_read.
pm_q_prop_single_read:

$\{1\} \quad \forall (\text{addr: Address, dt: (interpreted_data_type? [Data]), pm: Plain_Memory [State], s: State):$ $\text{plain_memory?}(pm) \wedge$ $\text{pm}'\text{states}(s) \wedge$ $\text{in_blessed_memory?}(dt, \text{addr}, (\text{pm}'\text{ro_addr} \cup \text{pm}'\text{rw_addr})) \wedge$ $\text{valid_in_mem}(pm, dt)(\text{addr})(s)$ $\supset \text{pm_q_prop}(pm)(\text{read_data}(pm, dt)(\text{addr})(s))$

Expanding the definition of pm_q_prop,
Repeatedly Skolemizing and flattening,
Using lemma plain_memory_transformer_invariant_read_data [State, Data],
Using lemma expr_transformer_invariant_next_ok [State, Data],
Using lemma plain_memory_read_data_ok [State, Data],
Expanding the definition of has_next_state,
Expanding the definition of singleton,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of pm_q_prop_single_read.
Q.E.D.

C.176.5 Plain_Mem_Rewrites2.pm_q_prop_ok_result_single

Terse proof for pm_q_prop_ok_result_single.
pm_q_prop_ok_result_single:

$\{1\} \quad \forall (\text{data: Data, pm: Plain_Memory [State], s: State):$ $\text{pm}'\text{states}(s) \supset \text{pm_q_prop}(pm)(\text{ok_result}(\text{data})(s))$

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of pm_q_prop_ok_result_single.
Q.E.D.

C.176.6 Plain_Mem_Rewrites2.plain_memory_write_ok_single

Terse proof for plain_memory_write_ok_single.
plain_memory_write_ok_single:

$\{1\} \quad \forall (\text{addr: Address, data: Data, dt: (interpreted_data_type? [Data]), pm: Plain_Memory [State],$ $s: State):$ $\text{plain_memory?}(pm) \wedge \text{pm}'\text{states}(s) \wedge \text{in_blessed_memory?}(dt, \text{addr}, \text{pm}'\text{rw_addr}) \supset$ $\text{OK?}(\text{write_data}(pm, dt)(\text{addr}, \text{data})(s))$

Repeatedly Skolemizing and flattening,
Rewriting using plain_memory_write_data_ok, matching in *,
This completes the proof of plain_memory_write_ok_single.
Q.E.D.

C.176.7 Plain_Mem_Rewrites2.pm_q_prop_read_TCC1

Terse proof for pm_q_prop_read_TCC1.

pm_q_prop_read_TCC1:

$$\{1\} \quad \forall (dt: (\text{interpreted_data_type?}[\text{Data}]), \text{res}: \text{StmtResult}[\text{State}]):$$

$$\text{OK?}(\text{res}) \supset$$

$$\text{OK?}[\text{State}](\text{res}) \vee$$

$$\text{Break?}[\text{State}](\text{res}) \vee$$

$$\text{Continue?}[\text{State}](\text{res}) \vee$$

$$\text{Return?}[\text{State}](\text{res}) \vee$$

$$\text{Switch?}[\text{State}](\text{res}) \vee \text{Default?}[\text{State}](\text{res}) \vee \text{Exception?}[\text{State}](\text{res})$$

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of pm_q_prop_read_TCC1.

Q.E.D.

C.177 Proofs for Plain_Mem_Rewrites3 (plain_memory_rewrites.pvs)

C.177.1 Plain_Mem_Rewrites3.pm_q_prop_read_TCC1

Terse proof for pm_q_prop_read_TCC1.

pm_q_prop_read_TCC1:

$$\{1\} \quad \forall (dt: (\text{interpreted_data_type?}[\text{Data}]), \text{res}: \text{ExprResult}[\text{State}, \text{Data}_q]):$$

$$\text{OK?}(\text{res}) \supset \text{OK?}[\text{State}, \text{Data}_q](\text{res}) \vee \text{Exception?}[\text{State}, \text{Data}_q](\text{res})$$

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of pm_q_prop_read_TCC1.

Q.E.D.

C.178 Proofs for Plain_Mem_Rewrites4 (plain_memory_rewrites.pvs)

C.178.1 Plain_Mem_Rewrites4.plain_memory_write_ok_q_expr

Terse proof for plain_memory_write_ok_q_expr.

plain_memory_write_ok_q_expr:

$$\{1\} \quad \forall (\text{addr}: \text{Address}, \text{data}: \text{Data}, dt: (\text{interpreted_data_type?}[\text{Data}]), \text{pm}: \text{Plain_Memory}[\text{State}]$$

$$s: \text{State}, q: [\text{State} \rightarrow \text{ExprResult}[\text{State}, \text{Data}_q]]):$$

$$\text{plain_memory?}(\text{pm}) \wedge \text{in_blessed_memory?}(dt, \text{addr}, \text{pm}'\text{rw_addr}) \wedge \text{pm_q_prop}(\text{pm})(q(s))$$

$$\supset \text{OK?}((q \ \#\# \ \text{write_data}(\text{pm}, dt)(\text{addr}, \text{data}))(s))$$

Repeatedly Skolemizing and flattening,

Expanding the definition of ##,

Expanding the definition of ##,

Expanding the definition of pm_q_prop,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 Rewriting using plain_memory_write_data_ok, matching in *,
 This completes the proof of plain_memory_write_ok_q_expr.
 Q.E.D.

C.178.2 Plain_Mem_Rewrites4.pm_q_prop_read_ok_expr

Terse proof for pm_q_prop_read_ok_expr.
 pm_q_prop_read_ok_expr:

$$\frac{\{1\} \quad \forall (\text{addr}: \text{Address}, \text{dt}: (\text{interpreted_data_type?}[\text{Data}]), \text{pm}: \text{Plain_Memory}[\text{State}], s: \text{State}, \\ q: [\text{State} \rightarrow \text{ExprResult}[\text{State}, \text{Data}_q]]): \\ \text{plain_memory?}(\text{pm}) \wedge \text{pm_q_prop_read}(\text{pm}, \text{dt}, \text{addr})(q(s)) \supset \\ \text{OK?}((q \ \#\# \ \text{read_data}(\text{pm}, \text{dt})(\text{addr}))(s))}{}$$

Repeatedly Skolemizing and flattening,
 Expanding the definition of pm_q_prop_read,
 Expanding the definition of ##,
 Expanding the definition of ##,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 This completes the proof of pm_q_prop_read_ok_expr.
 Q.E.D.

C.178.3 Plain_Mem_Rewrites4.pm_q_prop_read_pm_q_prop_expr

Terse proof for pm_q_prop_read_pm_q_prop_expr.
 pm_q_prop_read_pm_q_prop_expr:

$$\frac{\{1\} \quad \forall (\text{addr}: \text{Address}, \text{dt}: (\text{interpreted_data_type?}[\text{Data}]), \text{pm}: \text{Plain_Memory}[\text{State}], s: \text{State}, \\ q: [\text{State} \rightarrow \text{ExprResult}[\text{State}, \text{Data}_q]]): \\ \text{plain_memory?}(\text{pm}) \wedge \\ \text{in_blessed_memory?}(\text{dt}, \text{addr}, (\text{pm}'\text{ro_addr} \cup \text{pm}'\text{rw_addr})) \wedge \\ \text{pm_q_prop_read}(\text{pm}, \text{dt}, \text{addr})(q(s)) \\ \supset \text{pm_q_prop}(\text{pm})((q \ \#\# \ \text{read_data}(\text{pm}, \text{dt})(\text{addr}))(s))}{}$$

Repeatedly Skolemizing and flattening,
 Expanding the definition of ##,
 Expanding the definition of ##,
 Expanding the definition of pm_q_prop,
 Expanding the definition of pm_q_prop_read,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 Using lemma plain_memory_transformer_invariant_read_data[State, Data],
 Using lemma expr_transformer_invariant_next_ok[State, Data],
 Expanding the definition of has_next_state,
 Expanding the definition of singleton,
 which is trivially true.
 This completes the proof of pm_q_prop_read_pm_q_prop_expr.
 Q.E.D.

C.178.4 Plain_Mem_Rewrites4.pm_q_prop_read_write_q_expr

Terse proof for pm_q_prop_read_write_q_expr.

pm_q_prop_read_write_q_expr:

$$\{1\} \quad \forall (\text{addr: Address, data: Data, dt: (interpreted_data_type?}[\text{Data}]), \text{pm: Plain_Memory}[\text{State}], \text{s: State, q: [State} \rightarrow \text{ExprResult}[\text{State, Data_q}]]):$$

$$\text{plain_memory?}(\text{pm}) \wedge \text{in_blessed_memory?}(\text{dt, addr, pm}'\text{rw_addr}) \wedge \text{pm_q_prop}(\text{pm})(\text{q}(\text{s}))$$

$$\supset \text{pm_q_prop_read}(\text{pm, dt, addr})((\text{q} \ \#\# \ \text{write_data}(\text{pm, dt})(\text{addr, data}))(s))$$

Repeatedly Skolemizing and flattening,

Expanding the definition of ##,

Expanding the definition of ##,

Expanding the definition of pm_q_prop,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Expanding the definition of pm_q_prop_read,

Using lemma plain_memory_write_data_ok[State, Data],

Using lemma plain_memory_transformer_invariant_write_data[State, Data],

Using lemma expr_transformer_invariant_next_ok,

Expanding the definition of singleton,

Expanding the definition of has_next_state,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Using lemma plain_memory_read_write_ok[State, Data],

Expanding the definition of ##,

Expanding the definition of ##,

which is trivially true.

This completes the proof of pm_q_prop_read_write_q_expr.

Q.E.D.

C.178.5 Plain_Mem_Rewrites4.pm_q_prop_ok_result_q_expr

Terse proof for pm_q_prop_ok_result_q_expr.

pm_q_prop_ok_result_q_expr:

$$\{1\} \quad \forall (\text{data: Data, pm: Plain_Memory}[\text{State}], \text{s: State, q: [State} \rightarrow \text{ExprResult}[\text{State, Data_q}]]):$$

$$\text{plain_memory?}(\text{pm}) \wedge \text{pm_q_prop}(\text{pm})(\text{q}(\text{s})) \supset$$

$$\text{pm_q_prop}(\text{pm})((\text{q} \ \#\# \ \text{ok_result}[\text{State, Data}](\text{data}))(s))$$

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of pm_q_prop_ok_result_q_expr.

Q.E.D.

C.178.6 Plain_Mem_Rewrites4.pm_q_prop_read_ok_result_single

Terse proof for pm_q_prop_read_ok_result_single.

pm_q_prop_read_ok_result_single:

$$\{1\} \quad \forall (\text{addr: Address, data_q: Data_q, dt: (interpreted_data_type? [Data]),} \\ \text{pm: Plain_Memory [State], s: State):} \\ \text{plain_memory? (pm) } \wedge \\ \text{pm' states (s) } \wedge \\ \text{in_blessed_memory? (dt, addr, (pm'ro_addr } \cup \text{ pm'rw_addr)) } \wedge \\ \text{valid_in_mem (pm, dt) (addr) (s)} \\ \supset \text{pm_q_prop_read (pm, dt, addr) (ok_result [State, Data_q] (data_q) (s))}$$

Repeatedly Skolemizing and flattening,
 Expanding the definition of pm_q_prop_read,
 Using lemma pm_q_prop_ok_result_single [State, Data_q],
 Expanding the definition of pm_q_prop,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 Rewriting using plain_memory_read_data_ok, matching in *,
 Rewriting using ok_result_state, matching in *,
 This completes the proof of pm_q_prop_read_ok_result_single.
 Q.E.D.

C.178.7

Plain_Mem_Rewrites4.plain_memory_read_write_q_data_expr_TCC1

Terse proof for plain_memory_read_write_q_data_expr_TCC1.

plain_memory_read_write_q_data_expr_TCC1:

$$\{1\} \quad \forall (\text{addr: Address, data: Data, dt: (interpreted_data_type? [Data]), pm: Plain_Memory [State],} \\ \text{s: State, q: [State } \rightarrow \text{ ExprResult [State, Data_q])}): \\ \text{plain_memory? (pm) } \wedge \text{in_blessed_memory? (dt, addr, pm'rw_addr) } \wedge \text{pm_q_prop (pm) (q (s))} \\ \supset \\ \text{OK? [State, Data]} \\ \quad ((\#\# [\text{State, Unit, Data}] \\ \quad \quad (\#\# [\text{State, Data_q, Unit}] \\ \quad \quad \quad (q, \text{write_data [State, Data] (pm, dt) (addr, data)}, \\ \quad \quad \quad \text{read_data [State, Data] (pm, dt) (addr)}))) \\ \quad (s))$$

Repeatedly Skolemizing and flattening,
 Using lemma pm_q_prop_read_write_q_expr,
 Expanding the definition of pm_q_prop_read,
 Installing automatic rewrites from: ## lift
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 This completes the proof of plain_memory_read_write_q_data_expr_TCC1.
 Q.E.D.

C.178.8

Plain_Mem_Rewrites4.plain_memory_read_write_q_data_expr

Terse proof for plain_memory_read_write_q_data_expr.

plain_memory_read_write_q_data_expr:

$$\{1\} \quad \forall (\text{addr}: \text{Address}, \text{data1}: \text{Data}, \text{dt}: (\text{interpreted_data_type?}[\text{Data}]), \text{pm}: \text{Plain_Memory}[\text{State}], \text{s}: \text{State}, \text{q}: [\text{State} \rightarrow \text{ExprResult}[\text{State}, \text{Data_q}]]):$$

$$\text{plain_memory?}(\text{pm}) \wedge \text{in_blessed_memory?}(\text{dt}, \text{addr}, \text{pm}'\text{rw_addr}) \wedge \text{pm_q_prop}(\text{pm})(\text{q}(\text{s}))$$

$$\supset$$

$$\text{data}((\text{q} \text{ ## } \text{write_data}(\text{pm}, \text{dt})(\text{addr}, \text{data1}) \text{ ## } \text{read_data}(\text{pm}, \text{dt})(\text{addr}))(\text{s})) = \text{data1}$$

Repeatedly Skolemizing and flattening,
 Installing automatic rewrites from: ((##!!))
 Expanding the definition of pm_q_prop,
 Using lemma plain_memory_read_write_res[State, Data],
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 This completes the proof of plain_memory_read_write_q_data_expr.
 Q.E.D.

C.178.9 Plain_Mem_Rewrites4.pm_q_prop_read_write_other_single

Terse proof for pm_q_prop_read_write_other_single.

pm_q_prop_read_write_other_single:

$$\{1\} \quad \forall (\text{addr1}, \text{addr2}: \text{Address}, \text{data}: \text{Data}, \text{dt1}: (\text{interpreted_data_type?}[\text{Data}]), \text{dt2}: (\text{interpreted_data_type?}[\text{Data_q}]), \text{pm}: \text{Plain_Memory}[\text{State}], \text{s}: \text{State}):$$

$$\text{plain_memory?}(\text{pm}) \wedge$$

$$\text{blocks_disjoint?}(\text{addr1}, \text{size}(\text{uidt}(\text{dt1})), \text{addr2}, \text{size}(\text{uidt}(\text{dt2}))) \wedge$$

$$\text{in_blessed_memory?}(\text{dt1}, \text{addr1}, \text{pm}'\text{rw_addr}) \wedge$$

$$\text{in_blessed_memory?}(\text{dt2}, \text{addr2}, (\text{pm}'\text{ro_addr} \cup \text{pm}'\text{rw_addr})) \wedge$$

$$\text{valid_in_mem}(\text{pm}, \text{dt2})(\text{addr2})(\text{s}) \wedge \text{pm}'\text{states}(\text{s})$$

$$\supset \text{pm_q_prop_read}(\text{pm}, \text{dt2}, \text{addr2})(\text{write_data}(\text{pm}, \text{dt1})(\text{addr1}, \text{data})(\text{s}))$$

Repeatedly Skolemizing and flattening,
 Expanding the definition of pm_q_prop_read,
 Using lemma plain_memory_write_data_ok[State, Data],
 Using lemma plain_memory_read_write_other_ok[State, Data, Data_q],
 Installing automatic rewrites from: ## lift
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 Using lemma plain_memory_transformer_invariant_write_data[State, Data],
 Using lemma expr_transformer_invariant_next_ok,
 Expanding the definition of singleton,
 Expanding the definition of has_next_state,
 which is trivially true.
 This completes the proof of pm_q_prop_read_write_other_single.
 Q.E.D.

C.178.10 Plain_Mem_Rewrites4.pm_q_prop_read_read_single

Terse proof for pm_q_prop_read_read_single.

pm_q_prop_read_read_single:

$$\{1\} \quad \forall (\text{addr1, addr2: Address, dt1: (interpreted_data_type?[Data]),} \\ \text{dt2: (interpreted_data_type?[Data_q]), pm: Plain_Memory[State], s: State):} \\ \text{plain_memory?(pm) } \wedge \\ \text{in_blessed_memory?(dt1, addr1, (pm'ro_addr } \cup \text{ pm'rw_addr)) } \wedge \\ \text{in_blessed_memory?(dt2, addr2, (pm'ro_addr } \cup \text{ pm'rw_addr)) } \wedge \\ \text{valid_in_mem(pm, dt1)(addr1)(s) } \wedge \\ \text{valid_in_mem(pm, dt2)(addr2)(s) } \wedge \text{pm'states}(s) \\ \supset \text{pm_q_prop_read(pm, dt2, addr2)(read_data(pm, dt1)(addr1)(s))}$$

Repeatedly Skolemizing and flattening,
 Expanding the definition of pm_q_prop_read,
 Using lemma pm_q_prop_single_read[State, Data],
 Expanding the definition of pm_q_prop,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 Using lemma plain_memory_read_read_ok[State, Data, Data_q],
 Installing automatic rewrites from: ##
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 This completes the proof of pm_q_prop_read_read_single.
 Q.E.D.

C.179 Proofs for Plain_Mem_Rewrites5 (plain_memory_rewrites.pvs)

C.179.1 Plain_Mem_Rewrites5.plain_memory_write_ok_q_stmt

Terse proof for plain_memory_write_ok_q_stmt.

plain_memory_write_ok_q_stmt:

$$\{1\} \quad \forall (\text{addr: Address, data: Data, dt: (interpreted_data_type?[Data]), pm: Plain_Memory[State],} \\ \text{s: State, q: [State } \rightarrow \text{ StmtResult[State]])}: \\ \text{plain_memory?(pm) } \wedge \text{in_blessed_memory?(dt, addr, pm'rw_addr) } \wedge \text{pm_q_prop(pm)(q)(s)} \\ \supset \text{OK?((q ## write_data(pm, dt)(addr, data))(s))}$$

Installing automatic rewrites from: ##
 Repeatedly Skolemizing and flattening,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 Expanding the definition of pm_q_prop,
 Expanding the definition of lift,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 Rewriting using plain_memory_write_data_ok, matching in *,
 This completes the proof of plain_memory_write_ok_q_stmt.
 Q.E.D.

C.179.2 Plain_Mem_Rewrites5.pm_q_prop_read_ok_stmt

Terse proof for pm_q_prop_read_ok_stmt.

pm_q_prop_read_ok_stmt:

$\{1\} \quad \forall (\text{addr}: \text{Address}, \text{dt}: (\text{interpreted_data_type?}[\text{Data}]), \text{pm}: \text{Plain_Memory}[\text{State}], s: \text{State}, \\ q: [\text{State} \rightarrow \text{StmtResult}[\text{State}]]): \\ \text{plain_memory?}(\text{pm}) \wedge \text{pm_q_prop_read}(\text{pm}, \text{dt}, \text{addr})(q(s)) \supset \\ \text{OK?}((q \ \#\# \ \text{read_data}(\text{pm}, \text{dt})(\text{addr}))(s))$

Repeatedly Skolemizing and flattening,
 Expanding the definition of ##,
 Expanding the definition of lift,
 Expanding the definition of pm_q_prop_read,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 This completes the proof of pm_q_prop_read_ok_stmt.
 Q.E.D.

C.179.3 Plain_Mem_Rewrites5.pm_q_prop_read_pm_q_prop_stmt

Terse proof for pm_q_prop_read_pm_q_prop_stmt.

pm_q_prop_read_pm_q_prop_stmt:

$\{1\} \quad \forall (\text{addr}: \text{Address}, \text{dt}: (\text{interpreted_data_type?}[\text{Data}]), \text{pm}: \text{Plain_Memory}[\text{State}], s: \text{State}, \\ q: [\text{State} \rightarrow \text{StmtResult}[\text{State}]]): \\ \text{plain_memory?}(\text{pm}) \wedge \\ \text{in_blessed_memory?}(\text{dt}, \text{addr}, (\text{pm}'\text{ro_addr} \cup \text{pm}'\text{rw_addr})) \wedge \\ \text{pm_q_prop_read}(\text{pm}, \text{dt}, \text{addr})(q(s)) \\ \supset \text{pm_q_prop}(\text{pm})((q \ \#\# \ \text{read_data}(\text{pm}, \text{dt})(\text{addr}))(s))$

Repeatedly Skolemizing and flattening,
 Expanding the definition of ##,
 Expanding the definition of pm_q_prop,
 Expanding the definition of pm_q_prop_read,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 we get 2 subgoals:

pm_q_prop_read_pm_q_prop_stmt.1:

$\{-1\} \quad \text{interpreted_data_type?}[\text{Data}](\text{dt}')$
$\{-2\} \quad \text{plain_memory?}(\text{pm}')$
$\{-3\} \quad \text{in_blessed_memory?}(\text{dt}', \text{addr}', (\text{pm}'\text{ro_addr} \cup \text{pm}'\text{rw_addr}))$
$\{-4\} \quad \text{OK?}(q'(s'))$
$\{-5\} \quad \text{OK?}(\text{read_data}(\text{pm}', \text{dt}')(\text{addr}')(\text{state}(q'(s'))))$
$\{-6\} \quad \text{pm}'\text{states}(\text{state}(q'(s')))$
$\{1\} \quad \text{pm}'\text{states}(\text{state}(\text{lift}(\text{read_data}(\text{pm}', \text{dt}')(\text{addr}')))(q'(s')))$

Using lemma plain_memory_transformer_invariant_read_data[State, Data],
 Using lemma expr_transformer_invariant_next_ok[State, Data],
 Expanding the definition of has_next_state,
 Expanding the definition of singleton,
 Expanding the definition of lift,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 This completes the proof of pm_q_prop_read_pm_q_prop_stmt.1.

pm_q_prop_read_pm_q_prop_stmt.2:

{-1}	interpreted_data_type?[Data](dt')
{-2}	plain_memory?(pm')
{-3}	in_blessed_memory?(dt', addr', (pm'ro_addr ∪ pm'rw_addr))
{-4}	OK?(q'(s'))
{-5}	OK?(read_data(pm', dt')(addr')(state(q'(s'))))
{-6}	pm' states(state(q'(s')))
{1}	OK?(lift(read_data(pm', dt')(addr'))(q'(s')))

Expanding the definition of lift,
which is trivially true.

This completes the proof of pm_q_prop_read_pm_q_prop_stmt.2.

Q.E.D.

C.179.4 Plain_Mem_Rewrites5.pm_q_prop_read_write_q_stmt

Terse proof for pm_q_prop_read_write_q_stmt.

pm_q_prop_read_write_q_stmt:

{1}	\forall (addr: Address, data: Data, dt: (interpreted_data_type?[Data]), pm: Plain_Memory [State], s: State, q: [State → StmtResult [State]]): plain_memory?(pm) ∧ in_blessed_memory?(dt, addr, pm'rw_addr) ∧ pm_q_prop(pm)(q(s)) ⊃ pm_q_prop_read(pm, dt, addr)((q ## write_data(pm, dt)(addr, data))(s))
-----	--

Installing automatic rewrites from: ##

Repeatedly Skolemizing and flattening,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Expanding the definition of lift,

Expanding the definition of pm_q_prop,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Expanding the definition of pm_q_prop_read,

Using lemma plain_memory_write_data_ok [State, Data],

Using lemma plain_memory_transformer_invariant_write_data [State, Data],

Using lemma expr_transformer_invariant_next_ok,

Expanding the definition of singleton,

Expanding the definition of has_next_state,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Using lemma plain_memory_read_write_ok [State, Data],

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of pm_q_prop_read_write_q_stmt.

Q.E.D.

C.179.5 Plain_Mem_Rewrites5.pm_q_prop_ok_result_q_stmt

Terse proof for pm_q_prop_ok_result_q_stmt.

pm_q_prop_ok_result_q_stmt:

{1}	\forall (data: Data, pm: Plain_Memory [State], s: State, q: [State → StmtResult [State]]): plain_memory?(pm) ∧ pm_q_prop(pm)(q(s)) ⊃ pm_q_prop(pm)((q ## ok_result [State, Data] (data))(s))
-----	--

Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of `pm_q_prop_ok_result_q_stmt`.
 Q.E.D.

C.179.6

Plain_Mem_Rewrites5.plain_memory_read_write_q_data_stmt_TCC1

Terse proof for `plain_memory_read_write_q_data_stmt_TCC1`.
`plain_memory_read_write_q_data_stmt_TCC1`:

$\{1\} \quad \forall (\text{addr: Address, data: Data, dt: (interpreted_data_type?}[\text{Data}]), \text{pm: Plain_Memory}[\text{State}])$ $s: \text{State}, q: [\text{State} \rightarrow \text{StmtResult}[\text{State}]]:$ $\text{plain_memory?}(\text{pm}) \wedge \text{in_blessed_memory?}(\text{dt}, \text{addr}, \text{pm}'\text{rw_addr}) \wedge \text{pm_q_prop}(\text{pm})(q(s))$ \supset $\text{OK?}[\text{State}, \text{Data}]$ $((\#\#[\text{State}, \text{Unit}, \text{Data}]$ $(\#\#[\text{State}, \text{Unit}](q, \text{write_data}[\text{State}, \text{Data}](\text{pm}, \text{dt})(\text{addr}, \text{data})),$ $\text{read_data}[\text{State}, \text{Data}](\text{pm}, \text{dt})(\text{addr})))$ $(s))$

Repeatedly Skolemizing and flattening,
 Using lemma `pm_q_prop_read_write_q_stmt`,
 Expanding the definition of `pm_q_prop_read`,
 Installing automatic rewrites from: `## lift`
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 This completes the proof of `plain_memory_read_write_q_data_stmt_TCC1`.
 Q.E.D.

C.179.7

Plain_Mem_Rewrites5.plain_memory_read_write_q_data_stmt

Terse proof for `plain_memory_read_write_q_data_stmt`.
`plain_memory_read_write_q_data_stmt`:

$\{1\} \quad \forall (\text{addr: Address, data1: Data, dt: (interpreted_data_type?}[\text{Data}]), \text{pm: Plain_Memory}[\text{State}])$ $s: \text{State}, q: [\text{State} \rightarrow \text{StmtResult}[\text{State}]]:$ $\text{plain_memory?}(\text{pm}) \wedge \text{in_blessed_memory?}(\text{dt}, \text{addr}, \text{pm}'\text{rw_addr}) \wedge \text{pm_q_prop}(\text{pm})(q(s))$ \supset $\text{data}((q \#\# \text{write_data}(\text{pm}, \text{dt})(\text{addr}, \text{data1}) \#\# \text{read_data}(\text{pm}, \text{dt})(\text{addr}))(s)) =$ data1
--

Repeatedly Skolemizing and flattening,
 Installing automatic rewrites from: `((##!))`
 Expanding the definition of `pm_q_prop`,
 Using lemma `plain_memory_read_write_res` `[State, Data]`,
 Installing automatic rewrites from: `lift`
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 This completes the proof of `plain_memory_read_write_q_data_stmt`.
 Q.E.D.

C.179.8 Plain_Mem_Rewrites5.pm_q_prop_single_write

Terse proof for pm_q_prop_single_write.

pm_q_prop_single_write:

$$\{1\} \quad \forall (\text{addr: Address, data: Data, dt: (interpreted_data_type? [Data]), pm: Plain_Memory [State], s: State):$$

$$\text{plain_memory?}(pm) \wedge \text{in_blessed_memory?}(dt, \text{addr}, pm\text{'rw_addr}) \wedge pm\text{'states}(s) \supset$$

$$\text{pm_q_prop}(pm)(\text{write_data}(pm, dt)(\text{addr}, \text{data})(s))$$

Expanding the definition of pm_q_prop,
 Repeatedly Skolemizing and flattening,
 Using lemma plain_memory_write_data_ok[State, Data],
 Using lemma plain_memory_transformer_invariant_write_data[State, Data],
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 Using lemma expr_transformer_invariant_next_ok,
 Expanding the definition of singleton,
 Expanding the definition of has_next_state,
 which is trivially true.
 This completes the proof of pm_q_prop_single_write.
 Q.E.D.

C.179.9 Plain_Mem_Rewrites5.pm_q_prop_read_write_single

Terse proof for pm_q_prop_read_write_single.

pm_q_prop_read_write_single:

$$\{1\} \quad \forall (\text{addr: Address, data: Data, dt: (interpreted_data_type? [Data]), pm: Plain_Memory [State], s: State):$$

$$\text{plain_memory?}(pm) \wedge \text{in_blessed_memory?}(dt, \text{addr}, pm\text{'rw_addr}) \wedge pm\text{'states}(s) \supset$$

$$\text{pm_q_prop_read}(pm, dt, \text{addr})(\text{write_data}(pm, dt)(\text{addr}, \text{data})(s))$$

Repeatedly Skolemizing and flattening,
 Expanding the definition of pm_q_prop_read,
 Using lemma pm_q_prop_single_write,
 Using lemma plain_memory_read_write_ok[State, Data],
 Expanding the definition of pm_q_prop,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 Expanding the definition of ##,
 Expanding the definition of ##,
 which is trivially true.
 This completes the proof of pm_q_prop_read_write_single.
 Q.E.D.

C.179.10 Plain_Mem_Rewrites5.pm_q_prop_e2s

Terse proof for pm_q_prop_e2s.

pm_q_prop_e2s:

$$\{1\} \quad \forall (\text{expr: [State} \rightarrow \text{ExprResult [State, Data]], pm: Plain_Memory [State], s: State):$$

$$\text{pm_q_prop}(pm)(\text{e2s}(\text{expr})(s)) = \text{pm_q_prop}(pm)(\text{expr}(s))$$

Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of `pm_q_prop_e2s`.
 Q.E.D.

C.179.11 Plain_Mem_Rewrites5.pm_q_prop_stmt_e2s

Terse proof for `pm_q_prop_stmt_e2s`.
`pm_q_prop_stmt_e2s`:

$\{1\} \quad \forall (\text{expr}: [\text{State} \rightarrow \text{ExprResult}[\text{State}, \text{Data}]], \text{pm}: \text{Plain_Memory}[\text{State}], s: \text{State}, \\ \text{stmt}: [\text{State} \rightarrow \text{StmtResult}[\text{State}]]): \\ \text{pm_q_prop}(\text{pm})(\text{stmt}(s)) \supset \\ \text{pm_q_prop}(\text{pm})((\text{stmt} \## \text{e2s}(\text{expr}))(s)) = \text{pm_q_prop}(\text{pm})((\text{stmt} \## \text{expr})(s))$

Expanding the definition of `pm_q_prop`,
 Repeatedly Skolemizing and flattening,
 Using lemma `pm_q_prop_e2s`,
 Installing automatic rewrites from: `## lift`
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 This completes the proof of `pm_q_prop_stmt_e2s`.
 Q.E.D.

C.180 Proofs for Plain_Mem_Rewrites6 (plain_memory_rewrites.pvs)

C.180.1 Plain_Mem_Rewrites6.pm_q_prop_read_write_other_q_expr

Terse proof for `pm_q_prop_read_write_other_q_expr`.
`pm_q_prop_read_write_other_q_expr`:

$\{1\} \quad \forall (\text{addr1}, \text{addr2}: \text{Address}, \text{data2}: \text{Data2}, \text{dt1}: (\text{interpreted_data_type?}[\text{Data1}]), \\ \text{dt2}: (\text{interpreted_data_type?}[\text{Data2}]), \text{pm}: \text{Plain_Memory}[\text{State}], s: \text{State}, \\ q: [\text{State} \rightarrow \text{ExprResult}[\text{State}, \text{Data}_q]]): \\ \text{plain_memory?}(\text{pm}) \wedge \\ \text{blocks_disjoint?}(\text{addr1}, \text{size}(\text{uidt}(\text{dt1})), \text{addr2}, \text{size}(\text{uidt}(\text{dt2}))) \wedge \\ \text{in_blessed_memory?}(\text{dt2}, \text{addr2}, \text{pm}'\text{rw_addr}) \wedge \\ \text{in_blessed_memory?}(\text{dt1}, \text{addr1}, (\text{pm}'\text{ro_addr} \cup \text{pm}'\text{rw_addr})) \wedge \\ \text{pm_q_prop_read}(\text{pm}, \text{dt1}, \text{addr1})(q(s)) \\ \supset \text{pm_q_prop_read}(\text{pm}, \text{dt1}, \text{addr1})((q \## \text{write_data}(\text{pm}, \text{dt2})(\text{addr2}, \text{data2}))(s))$
--

Repeatedly Skolemizing and flattening,
 Expanding the definition of `pm_q_prop_read`,
 Using lemma `pm_q_prop_single_write[State, Data2]`,
 Expanding the definition of `pm_q_prop`,
 Using lemma `plain_memory_read_write_other_ok[State, Data2, Data1]`,
 Installing automatic rewrites from: `## lift`
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 we get 2 subgoals:

pm_q_prop_read_write_other_q_expr.1:

<pre> {-1} plain_memory?(pm') {-2} pm' 'states(state(q'(s'))) {-3} in_blessed_memory?(dt2', addr2', pm' 'rw_addr) {-4} in_blessed_memory?(dt1', addr1', (pm' 'ro_addr ∪ pm' 'rw_addr)) {-5} OK?(write_data(pm', dt2')(addr2', data2')(state(q'(s')))) {-6} pm' 'states(state(write_data(pm', dt2')(addr2', data2')(state(q'(s'))))) {-7} interpreted_data_type?[Data1](dt1') {-8} interpreted_data_type?[Data2](dt2') {-9} blocks_disjoint?(addr1', size(uidt(dt1')), addr2', size(uidt(dt2'))) {-10} OK?(q'(s')) {-11} OK?(read_data(pm', dt1')(addr1')(state(q'(s')))) </pre>	<hr style="border: 0.5px solid black;"/> <pre> {1} valid_in_mem(pm', dt1')(addr1')(state(q'(s'))) {2} OK?(read_data(pm', dt1')(addr1') (state(write_data(pm', dt2')(addr2', data2')(state(q'(s')))))) </pre>
---	--

Expanding the definition of valid_in_mem,

Expanding the definition of read_data,

Rewriting using read_data_valid_in_mem, matching in *,

This completes the proof of pm_q_prop_read_write_other_q_expr.1.

pm_q_prop_read_write_other_q_expr.2:

<pre> {-1} plain_memory?(pm') {-2} pm' 'states(state(q'(s'))) {-3} in_blessed_memory?(dt2', addr2', pm' 'rw_addr) {-4} in_blessed_memory?(dt1', addr1', (pm' 'ro_addr ∪ pm' 'rw_addr)) {-5} OK?(write_data(pm', dt2')(addr2', data2')(state(q'(s')))) {-6} pm' 'states(state(write_data(pm', dt2')(addr2', data2')(state(q'(s'))))) {-7} interpreted_data_type?[Data1](dt1') {-8} interpreted_data_type?[Data2](dt2') {-9} blocks_disjoint?(addr1', size(uidt(dt1')), addr2', size(uidt(dt2'))) {-10} OK?(q'(s')) {-11} OK?(read_data(pm', dt1')(addr1')(state(q'(s')))) </pre>	<hr style="border: 0.5px solid black;"/> <pre> {1} blocks_disjoint?(addr2', size(uidt(dt2')), addr1', size(uidt(dt1'))) {2} OK?(read_data(pm', dt1')(addr1') (state(write_data(pm', dt2')(addr2', data2')(state(q'(s')))))) </pre>
---	--

Keeping (-9 1) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of pm_q_prop_read_write_other_q_expr.2.

Q.E.D.

C.180.2 Plain_Mem_Rewrites6.pm_q_prop_read_read_q_expr

Terse proof for pm_q_prop_read_read_q_expr.

pm_q_prop_read_read_q_expr :

<p>{1}</p>	$\begin{aligned} & \forall (\text{addr1}, \text{addr2}: \text{Address}, \text{dt1}: (\text{interpreted_data_type?}[\text{Data1}]), \\ & \quad \text{dt2}: (\text{interpreted_data_type?}[\text{Data2}]), \text{pm}: \text{Plain_Memory}[\text{State}], s: \text{State}, \\ & \quad q: [\text{State} \rightarrow \text{ExprResult}[\text{State}, \text{Data}_q]]): \\ & \text{plain_memory?}(\text{pm}) \wedge \\ & \text{in_blessed_memory?}(\text{dt1}, \text{addr1}, (\text{pm}'\text{ro_addr} \cup \text{pm}'\text{rw_addr})) \wedge \\ & \text{in_blessed_memory?}(\text{dt2}, \text{addr2}, (\text{pm}'\text{ro_addr} \cup \text{pm}'\text{rw_addr})) \wedge \\ & \text{pm_q_prop_read}(\text{pm}, \text{dt1}, \text{addr1})(q(s)) \wedge \text{pm_q_prop_read}(\text{pm}, \text{dt2}, \text{addr2})(q(s)) \\ & \supset \text{pm_q_prop_read}(\text{pm}, \text{dt2}, \text{addr2})((q \text{ ## read_data}(\text{pm}, \text{dt1})(\text{addr1}))(s)) \end{aligned}$
------------	--

Repeatedly Skolemizing and flattening,

Expanding the definition of pm_q_prop_read,

Applying disjunctive simplification to flatten sequent,

Using lemma pm_q_prop_read_read_single[State, Data1, Data2],

Expanding the definition of read_data,

Expanding the definition of valid_in_mem,

Installing automatic rewrites from: ## lift

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 2 subgoals:

pm_q_prop_read_read_q_expr.1:

<p>{-1}</p>	plain_memory?(pm')
<p>{-2}</p>	in_blessed_memory?(dt1', addr1', (pm'ro_addr ∪ pm'rw_addr))
<p>{-3}</p>	in_blessed_memory?(dt2', addr2', (pm'ro_addr ∪ pm'rw_addr))
<p>{-4}</p>	pm' states(state(q'(s')))
<p>{-5}</p>	interpreted_data_type?[Data1](dt1')
<p>{-6}</p>	interpreted_data_type?[Data2](dt2')
<p>{-7}</p>	OK?(q'(s'))
<p>{-8}</p>	OK?(read_data(pm' mem, dt1')(addr1')(state(q'(s'))))
<p>{-9}</p>	OK?(read_data(pm' mem, dt2')(addr2')(state(q'(s'))))
<p>{1}</p>	valid_in_mem(pm' mem, dt2')(addr2')(state(q'(s')))
<p>{2}</p>	pm_q_prop_read(pm', dt2', addr2') (read_data(pm' mem, dt1')(addr1')(state(q'(s'))))

Rewriting using read_data_valid_in_mem, matching in *,

This completes the proof of pm_q_prop_read_read_q_expr.1.

pm_q_prop_read_read_q_expr.2:

<p>{-1}</p>	plain_memory?(pm')
<p>{-2}</p>	in_blessed_memory?(dt1', addr1', (pm'ro_addr ∪ pm'rw_addr))
<p>{-3}</p>	in_blessed_memory?(dt2', addr2', (pm'ro_addr ∪ pm'rw_addr))
<p>{-4}</p>	pm' states(state(q'(s')))
<p>{-5}</p>	interpreted_data_type?[Data1](dt1')
<p>{-6}</p>	interpreted_data_type?[Data2](dt2')
<p>{-7}</p>	OK?(q'(s'))
<p>{-8}</p>	OK?(read_data(pm' mem, dt1')(addr1')(state(q'(s'))))
<p>{-9}</p>	OK?(read_data(pm' mem, dt2')(addr2')(state(q'(s'))))
<p>{1}</p>	valid_in_mem(pm' mem, dt1')(addr1')(state(q'(s')))
<p>{2}</p>	pm_q_prop_read(pm', dt2', addr2') (read_data(pm' mem, dt1')(addr1')(state(q'(s'))))

Rewriting using read_data_valid_in_mem, matching in *,

This completes the proof of pm_q_prop_read_read_q_expr.2.

Q.E.D.

C.180.3 Plain_Mem_Rewrites6.pm_q_prop_read_ok_result_q_expr

Terse proof for pm_q_prop_read_ok_result_q_expr.

pm_q_prop_read_ok_result_q_expr:

$$\{1\} \quad \forall (\text{addr2: Address, data1: Data1, dt2: (interpreted_data_type?[Data2]),} \\ \text{pm: Plain_Memory[State], s: State, q: [State} \rightarrow \text{ExprRe-} \\ \text{sult[State, Data_q])}: \\ \text{plain_memory?(pm)} \wedge \text{pm_q_prop_read(pm, dt2, addr2)(q(s))} \supset \\ \text{pm_q_prop_read(pm, dt2, addr2)((q \#\# \text{ok_result[State, Data1](data1)})(s))$$

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of pm_q_prop_read_ok_result_q_expr.
Q.E.D.

C.180.4

Plain_Mem_Rewrites6.plain_memory_read_read_q_data_expr_TCC1

Terse proof for plain_memory_read_read_q_data_expr_TCC1.

plain_memory_read_read_q_data_expr_TCC1:

$$\{1\} \quad \forall (\text{addr1, addr2: Address, dt1: (interpreted_data_type?[Data1]),} \\ \text{dt2: (interpreted_data_type?[Data2]), pm: Plain_Memory[State], s: State,} \\ \text{q: [State} \rightarrow \text{ExprResult[State, Data_q])}: \\ \text{plain_memory?(pm)} \wedge \\ \text{in_blessed_memory?(dt1, addr1, (pm'ro_addr} \cup \text{pm'rw_addr))} \wedge \\ \text{in_blessed_memory?(dt2, addr2, (pm'ro_addr} \cup \text{pm'rw_addr))} \wedge \\ \text{pm_q_prop_read(pm, dt2, addr2)(q(s))} \wedge \text{pm_q_prop_read(pm, dt1, addr1)(q(s))} \\ \supset \\ \text{OK?[State, Data1} \\ ((\#\#[\text{State, Data2, Data1} \\ (\#\#[\text{State, Data_q, Data2}](q, \text{read_data[State, Data2](pm, dt2)(addr2)),} \\ \text{read_data[State, Data1](pm, dt1)(addr1))} \\ (s))$$

Expanding the definition of pm_q_prop_read,
Repeatedly Skolemizing and flattening,
Using lemma plain_memory_read_read_ok[State, Data2, Data1],
Installing automatic rewrites from: ## lift
Installing automatic rewrites from: valid_in_mem read_data read_data_valid_in_mem
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of plain_memory_read_read_q_data_expr_TCC1.
Q.E.D.

C.180.5

Plain_Mem_Rewrites6.plain_memory_read_read_q_data_expr_TCC2

Terse proof for plain_memory_read_read_q_data_expr_TCC2.

plain_memory_read_read_q_data_expr_TCC2:

<pre>{1} ∃ (addr1, addr2: Address, dt1: (interpreted_data_type?[Data1]), dt2: (interpreted_data_type?[Data2]), pm: Plain_Memory[State], s: State, q: [State → ExprResult[State, Data_q]]): plain_memory?(pm) ∧ in_blessed_memory?(dt1, addr1, (pm'ro_addr ∪ pm'rw_addr)) ∧ in_blessed_memory?(dt2, addr2, (pm'ro_addr ∪ pm'rw_addr)) ∧ pm_q_prop_read(pm, dt2, addr2)(q(s)) ∧ pm_q_prop_read(pm, dt1, addr1)(q(s)) ⊃ OK?[State, Data1] ((##[State, Data_q, Data1](q, read_data[State, Data1](pm, dt1)(addr1)))(s))</pre>

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of plain_memory_read_read_q_data_expr_TCC2.

Q.E.D.

C.180.6 Plain_Mem_Rewrites6.plain_memory_read_read_q_data_expr

Terse proof for plain_memory_read_read_q_data_expr.

plain_memory_read_read_q_data_expr:

<pre>{1} ∃ (addr1, addr2: Address, dt1: (interpreted_data_type?[Data1]), dt2: (interpreted_data_type?[Data2]), pm: Plain_Memory[State], s: State, q: [State → ExprResult[State, Data_q]]): plain_memory?(pm) ∧ in_blessed_memory?(dt1, addr1, (pm'ro_addr ∪ pm'rw_addr)) ∧ in_blessed_memory?(dt2, addr2, (pm'ro_addr ∪ pm'rw_addr)) ∧ pm_q_prop_read(pm, dt2, addr2)(q(s)) ∧ pm_q_prop_read(pm, dt1, addr1)(q(s)) ⊃ data((q ## read_data(pm, dt2)(addr2) ## read_data(pm, dt1)(addr1))(s)) = data((q ## read_data(pm, dt1)(addr1))(s))</pre>
--

Expanding the definition of pm_q_prop_read,

Repeatedly Skolemizing and flattening,

Using lemma plain_memory_read_read[State, Data2, Data1],

Installing automatic rewrites from: ## lift valid_in_mem read_data read_data_valid_in_mem

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of plain_memory_read_read_q_data_expr.

Q.E.D.

C.181 Proofs for Plain_Mem_Rewrites7 (plain_memory_rewrites.pvs)

C.181.1 Plain_Mem_Rewrites7.pm_q_prop_read_write_other_q_stmt

Terse proof for pm_q_prop_read_write_other_q_stmt.

pm_q_prop_read_write_other_q_stmt:

{1}	$\forall (\text{addr1}, \text{addr2}: \text{Address}, \text{data2}: \text{Data2}, \text{dt1}: (\text{interpreted_data_type?}[\text{Data1}]),$ $\text{dt2}: (\text{interpreted_data_type?}[\text{Data2}]), \text{pm}: \text{Plain_Memory}[\text{State}], \text{s}: \text{State},$ $q: [\text{State} \rightarrow \text{StmtResult}[\text{State}]]):$ $\text{plain_memory?}(\text{pm}) \wedge$ $\text{blocks_disjoint?}(\text{addr1}, \text{size}(\text{uidt}(\text{dt1})), \text{addr2}, \text{size}(\text{uidt}(\text{dt2}))) \wedge$ $\text{in_blessed_memory?}(\text{dt2}, \text{addr2}, \text{pm}'\text{rw_addr}) \wedge$ $\text{in_blessed_memory?}(\text{dt1}, \text{addr1}, (\text{pm}'\text{ro_addr} \cup \text{pm}'\text{rw_addr})) \wedge$ $\text{pm_q_prop_read}(\text{pm}, \text{dt1}, \text{addr1})(q(\text{s}))$ $\supset \text{pm_q_prop_read}(\text{pm}, \text{dt1}, \text{addr1})((q \text{ ## write_data}(\text{pm}, \text{dt2})(\text{addr2}, \text{data2}))(s))$
-----	---

Repeatedly Skolemizing and flattening,

Expanding the definition of pm_q_prop_read,

Using lemma pm_q_prop_single_write[State, Data2],

Expanding the definition of pm_q_prop,

Using lemma plain_memory_read_write_other_ok[State, Data2, Data1],

Installing automatic rewrites from: ## lift

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 2 subgoals:

pm_q_prop_read_write_other_q_stmt.1:

{-1}	plain_memory?(pm')
{-2}	pm' states(state(q'(s')))
{-3}	in_blessed_memory?(dt2', addr2', pm'rw_addr)
{-4}	in_blessed_memory?(dt1', addr1', (pm'ro_addr \cup pm'rw_addr))
{-5}	OK?(write_data(pm', dt2')(addr2', data2')(state(q'(s'))))
{-6}	pm' states(state(write_data(pm', dt2')(addr2', data2')(state(q'(s')))))
{-7}	interpreted_data_type?[Data1](dt1')
{-8}	interpreted_data_type?[Data2](dt2')
{-9}	blocks_disjoint?(addr1', size(uidt(dt1')), addr2', size(uidt(dt2')))
{-10}	OK?(q'(s'))
{-11}	OK?(read_data(pm', dt1')(addr1')(state(q'(s'))))
{1}	valid_in_mem(pm', dt1')(addr1')(state(q'(s')))
{2}	OK?(read_data(pm', dt1')(addr1') (state(write_data(pm', dt2')(addr2', data2')(state(q'(s')))))

Expanding the definition of valid_in_mem,

Expanding the definition of read_data,

Rewriting using read_data_valid_in_mem, matching in *,

This completes the proof of pm_q_prop_read_write_other_q_stmt.1.

pm_q_prop_read_write_other_q_stmt.2:

{-1}	plain_memory?(pm')
{-2}	pm' 'states(state(q'(s')))
{-3}	in_blessed_memory?(dt2', addr2', pm' 'rw_addr)
{-4}	in_blessed_memory?(dt1', addr1', (pm' 'ro_addr \cup pm' 'rw_addr))
{-5}	OK?(write_data(pm', dt2')(addr2', data2')(state(q'(s'))))
{-6}	pm' 'states(state(write_data(pm', dt2')(addr2', data2')(state(q'(s')))))
{-7}	interpreted_data_type?[Data1](dt1')
{-8}	interpreted_data_type?[Data2](dt2')
{-9}	blocks_disjoint?(addr1', size(uidt(dt1')), addr2', size(uidt(dt2')))
{-10}	OK?(q'(s'))
{-11}	OK?(read_data(pm', dt1')(addr1')(state(q'(s'))))
{1}	blocks_disjoint?(addr2', size(uidt(dt2')), addr1', size(uidt(dt1')))
{2}	OK?(read_data(pm', dt1')(addr1') (state(write_data(pm', dt2')(addr2', data2')(state(q'(s'))))))

Keeping (-9 1) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of pm_q_prop_read_write_other_q_stmt.2.

Q.E.D.

C.181.2

Plain_Mem_Rewrites7.plain_memory_read_write_other_single_data_TCC1

Terse proof for plain_memory_read_write_other_single_data_TCC1.

plain_memory_read_write_other_single_data_TCC1:

{1}	\forall (addr1, addr2: Address, data2: Data2, dt1: (interpreted_data_type?[Data1]), dt2: (interpreted_data_type?[Data2]), pm: Plain_Memory[State], s: State): pm' 'states(s) \wedge pm_q_prop(pm)(read_data(pm, dt1)(addr1)(s)) \wedge blocks_disjoint?(addr1, size(uidt(dt1)), addr2, size(uidt(dt2))) \wedge in_blessed_memory?(dt2, addr2, pm' 'rw_addr) \wedge in_blessed_memory?(dt1, addr1, (pm' 'ro_addr \cup pm' 'rw_addr)) \wedge plain_memory?(pm) \supset OK?[State, Data1] ((##[State, Unit, Data1] (write_data[State, Data2](pm, dt2)(addr2, data2), read_data[State, Data1](pm, dt1)(addr1))) (s))
-----	--

Repeatedly Skolemizing and flattening,

Rewriting using plain_memory_read_write_other_ok[State, Data2, Data1], matching in *,

we get 2 subgoals:

plain_memory_read_write_other_single_data_TCC1.1:

{-1}	interpreted_data_type?[Data1](dt1')
{-2}	interpreted_data_type?[Data2](dt2')
{-3}	pm' 'states(s')
{-4}	pm_q_prop(pm')(read_data(pm', dt1')(addr1')(s'))
{-5}	blocks_disjoint?(addr1', size(uidt(dt1')), addr2', size(uidt(dt2')))
{-6}	in_blessed_memory?(dt2', addr2', pm' 'rw_addr)
{-7}	in_blessed_memory?(dt1', addr1', (pm' 'ro_addr \cup pm' 'rw_addr))
{-8}	plain_memory?(pm')
{1}	blocks_disjoint?(addr2', size(uidt(dt2')), addr1', size(uidt(dt1')))
{2}	OK?[State, Data1] ((##[State, Unit, Data1] (write_data[State, Data2](pm', dt2')(addr2', data2'), read_data[State, Data1](pm', dt1')(addr1')) (s'))

Keeping (-5 1) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of plain_memory_read_write_other_single_data_TCC1.1.

plain_memory_read_write_other_single_data_TCC1.2:

{-1}	interpreted_data_type?[Data1](dt1')
{-2}	interpreted_data_type?[Data2](dt2')
{-3}	pm' 'states(s')
{-4}	pm_q_prop(pm')(read_data(pm', dt1')(addr1')(s'))
{-5}	blocks_disjoint?(addr1', size(uidt(dt1')), addr2', size(uidt(dt2')))
{-6}	in_blessed_memory?(dt2', addr2', pm' 'rw_addr)
{-7}	in_blessed_memory?(dt1', addr1', (pm' 'ro_addr \cup pm' 'rw_addr))
{-8}	plain_memory?(pm')
{1}	valid_in_mem(pm', dt1')(addr1')(s')
{2}	OK?[State, Data1] ((##[State, Unit, Data1] (write_data[State, Data2](pm', dt2')(addr2', data2'), read_data[State, Data1](pm', dt1')(addr1')) (s'))

Keeping (-4 1) and hiding *,

Expanding the definition of pm_q_prop,

Installing automatic rewrites from: read_data valid_in_mem read_data_valid_in_mem

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of plain_memory_read_write_other_single_data_TCC1.2.

Q.E.D.

C.181.3

Plain_Mem_Rewrites7.plain_memory_read_write_other_single_data_TCC2

Terse proof for plain_memory_read_write_other_single_data_TCC2.

plain_memory_read_write_other_single_data_TCC2:

<pre>{1} ∃ (addr1, addr2: Address, dt1: (interpreted_data_type?[Data1]), dt2: (interpreted_data_type?[Data2]), pm: Plain_Memory[State], s: State): pm'states(s) ∧ pm_q_prop(pm)(read_data(pm, dt1)(addr1)(s)) ∧ blocks_disjoint?(addr1, size(uidt(dt1)), addr2, size(uidt(dt2))) ∧ in_blessed_memory?(dt2, addr2, pm'rw_addr) ∧ in_blessed_memory?(dt1, addr1, (pm'ro_addr ∪ pm'rw_addr)) ∧ plain_memory?(pm) ⊃ OK?[State, Data1](read_data[State, Data1](pm, dt1)(addr1)(s))</pre>
--

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of plain_memory_read_write_other_single_data_TCC2.

Q.E.D.

C.181.4

Plain_Mem_Rewrites7.plain_memory_read_write_other_single_data

Terse proof for plain_memory_read_write_other_single_data.

plain_memory_read_write_other_single_data:

<pre>{1} ∃ (addr1, addr2: Address, data2: Data2, dt1: (interpreted_data_type?[Data1]), dt2: (interpreted_data_type?[Data2]), pm: Plain_Memory[State], s: State): plain_memory?(pm) ∧ in_blessed_memory?(dt1, addr1, (pm'ro_addr ∪ pm'rw_addr)) ∧ in_blessed_memory?(dt2, addr2, pm'rw_addr) ∧ blocks_disjoint?(addr1, size(uidt(dt1)), addr2, size(uidt(dt2))) ∧ pm_q_prop(pm)(read_data(pm, dt1)(addr1)(s)) ∧ pm'states(s) ⊃ data((write_data(pm, dt2)(addr2, data2) ## read_data(pm, dt1)(addr1))(s)) = data(read_data(pm, dt1)(addr1)(s))</pre>

Repeatedly Skolemizing and flattening,

Expanding the definition of pm_q_prop,

Rewriting using plain_memory_read_write_other_res, matching in *,

we get 2 subgoals:

plain_memory_read_write_other_single_data.1:

<pre>{-1} interpreted_data_type?[Data1](dt1') {-2} interpreted_data_type?[Data2](dt2') {-3} plain_memory?(pm') {-4} in_blessed_memory?(dt1', addr1', (pm'ro_addr ∪ pm'rw_addr)) {-5} in_blessed_memory?(dt2', addr2', pm'rw_addr) {-6} blocks_disjoint?(addr1', size(uidt(dt1')), addr2', size(uidt(dt2')))) {-7} OK?(read_data(pm', dt1')(addr1')(s')) ∧ pm'states(state(read_data(pm', dt1')(addr1')(s'))) {-8} pm'states(s')</pre>
<pre>{1} blocks_disjoint?(addr2', size(uidt(dt2')), addr1', size(uidt(dt1')))) {2} data((write_data(pm', dt2')(addr2', data2') ## read_data(pm', dt1')(addr1')) (s')) = data(read_data(pm', dt1')(addr1')(s'))</pre>

Keeping (-6 1) and hiding *,
 Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of plain_memory_read_write_other_single_data.1.
 plain_memory_read_write_other_single_data.2:

{-1}	interpreted_data_type?[Data1](dt1')
{-2}	interpreted_data_type?[Data2](dt2')
{-3}	plain_memory?(pm')
{-4}	in_blessed_memory?(dt1', addr1', (pm'ro_addr ∪ pm'rw_addr))
{-5}	in_blessed_memory?(dt2', addr2', pm'rw_addr)
{-6}	blocks_disjoint?(addr1', size(uidt(dt1')), addr2', size(uidt(dt2')))
{-7}	OK?(read_data(pm', dt1')(addr1')(s')) ∧ pm'states(state(read_data(pm', dt1')(addr1')(s')))
{-8}	pm'states(s')
{1}	valid_in_mem(pm', dt1')(addr1')(s')
{2}	data((write_data(pm', dt2')(addr2', data2') ## read_data(pm', dt1')(addr1')) (s')) = data(read_data(pm', dt1')(addr1')(s'))

Expanding the definition of valid_in_mem,
 Expanding the definition of read_data,
 Rewriting using read_data_valid_in_mem, matching in *,
 This completes the proof of plain_memory_read_write_other_single_data.2.
 Q.E.D.

C.181.5 Plain_Mem_Rewrites7.pm_q_prop_read_read_q_stmt

Terse proof for pm_q_prop_read_read_q_stmt.

pm_q_prop_read_read_q_stmt:

{1}	$\forall (\text{addr1}, \text{addr2}: \text{Address}, \text{dt1}: (\text{interpreted_data_type?}[\text{Data1}]),$ $\text{dt2}: (\text{interpreted_data_type?}[\text{Data2}]), \text{pm}: \text{Plain_Memory}[\text{State}], \text{s}: \text{State},$ $\text{q}: [\text{State} \rightarrow \text{StmtResult}[\text{State}]]):$ $\text{plain_memory?}(\text{pm}) \wedge$ $\text{in_blessed_memory?}(\text{dt1}, \text{addr1}, (\text{pm}'\text{ro_addr} \cup \text{pm}'\text{rw_addr})) \wedge$ $\text{in_blessed_memory?}(\text{dt2}, \text{addr2}, (\text{pm}'\text{ro_addr} \cup \text{pm}'\text{rw_addr})) \wedge$ $\text{pm_q_prop_read}(\text{pm}, \text{dt1}, \text{addr1})(\text{q}(\text{s})) \wedge \text{pm_q_prop_read}(\text{pm}, \text{dt2}, \text{addr2})(\text{q}(\text{s}))$ $\supset \text{pm_q_prop_read}(\text{pm}, \text{dt2}, \text{addr2})((\text{q} \ ## \ \text{read_data}(\text{pm}, \text{dt1})(\text{addr1}))(s))$
-----	--

Repeatedly Skolemizing and flattening,
 Expanding the definition of pm_q_prop_read,
 Applying disjunctive simplification to flatten sequent,
 Using lemma pm_q_prop_read_read_single[State, Data1, Data2],
 Installing automatic rewrites from: ## lift read_data valid_in_mem read_data_valid_in_mem
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 This completes the proof of pm_q_prop_read_read_q_stmt.
 Q.E.D.

C.181.6 Plain_Mem_Rewrites7.pm_q_prop_read_ok_result_q_stmt

Terse proof for pm_q_prop_read_ok_result_q_stmt.

pm_q_prop_read_ok_result_q_stmt:

$$\{1\} \quad \forall (\text{addr2: Address, data1: Data1, dt2: (interpreted_data_type? [Data2]),} \\ \text{pm: Plain_Memory [State], s: State, q: [State} \rightarrow \text{StmtResult [State])]:} \\ \text{plain_memory?(pm)} \wedge \text{pm_q_prop_read(pm, dt2, addr2)(q(s))} \supset \\ \text{pm_q_prop_read(pm, dt2, addr2)((q \#\# \text{ok_result [State, Data1] (data1)) (s))}$$

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of pm_q_prop_read_ok_result_q_stmt.

Q.E.D.

C.181.7

Plain_Mem_Rewrites7.plain_memory_read_write_other_q_data_stmt_TCC1

Terse proof for plain_memory_read_write_other_q_data_stmt_TCC1.

plain_memory_read_write_other_q_data_stmt_TCC1:

$$\{1\} \quad \forall (\text{addr1, addr2: Address, data2: Data2, dt1: (interpreted_data_type? [Data1]),} \\ \text{dt2: (interpreted_data_type? [Data2]), pm: Plain_Memory [State], s: State,} \\ \text{q: [State} \rightarrow \text{StmtResult [State])}: \\ \text{plain_memory?(pm)} \wedge \\ \text{in_blessed_memory?(dt1, addr1, (pm'ro_addr} \cup \text{pm'rw_addr))} \wedge \\ \text{in_blessed_memory?(dt2, addr2, pm'rw_addr)} \wedge \\ \text{blocks_disjoint?(addr1, size(uidt(dt1)), addr2, size(uidt(dt2)))} \wedge \\ \text{pm_q_prop_read(pm, dt1, addr1)(q(s))} \\ \supset \\ \text{OK? [State, Data1]} \\ ((\#\# [\text{State, Unit, Data1}] \\ (\#\# [\text{State, Unit}] (q, \text{write_data [State, Data2] (pm, dt2) (addr2, data2)), \\ \text{read_data [State, Data1] (pm, dt1) (addr1)}))) \\ (s))$$

Repeatedly Skolemizing and flattening,

Using lemma pm_q_prop_read_write_other_q_stmt,

Expanding the definition of pm_q_prop_read,

Installing automatic rewrites from: $\#\#$ lift

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of plain_memory_read_write_other_q_data_stmt_TCC1.

Q.E.D.

C.181.8

Plain_Mem_Rewrites7.plain_memory_read_write_other_q_data_stmt_TCC2

Terse proof for plain_memory_read_write_other_q_data_stmt_TCC2.

plain_memory_read_write_other_q_data_stmt_TCC2:

<pre> {1} ∃ (addr1, addr2: Address, dt1: (interpreted_data_type?[Data1]), dt2: (interpreted_data_type?[Data2]), pm: Plain_Memory[State], s: State, q: [State → StmtResult[State]]): plain_memory?(pm) ∧ in_blessed_memory?(dt1, addr1, (pm'ro_addr ∪ pm'rw_addr)) ∧ in_blessed_memory?(dt2, addr2, pm'rw_addr) ∧ blocks_disjoint?(addr1, size(uidt(dt1)), addr2, size(uidt(dt2))) ∧ pm_q_prop_read(pm, dt1, addr1)(q(s)) ⊃ OK?[State, Data1] ((##[State, Data1](q, read_data[State, Data1](pm, dt1)(addr1)))(s) </pre>

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of plain_memory_read_write_other_q_data_stmt_TCC2.
Q.E.D.

C.181.9

Plain_Mem_Rewrites7.plain_memory_read_write_other_q_data_stmt

Terse proof for plain_memory_read_write_other_q_data_stmt.

plain_memory_read_write_other_q_data_stmt:

<pre> {1} ∃ (addr1, addr2: Address, data2: Data2, dt1: (interpreted_data_type?[Data1]), dt2: (interpreted_data_type?[Data2]), pm: Plain_Memory[State], s: State, q: [State → StmtResult[State]]): plain_memory?(pm) ∧ in_blessed_memory?(dt1, addr1, (pm'ro_addr ∪ pm'rw_addr)) ∧ in_blessed_memory?(dt2, addr2, pm'rw_addr) ∧ blocks_disjoint?(addr1, size(uidt(dt1)), addr2, size(uidt(dt2))) ∧ pm_q_prop_read(pm, dt1, addr1)(q(s)) ⊃ data((q ## write_data(pm, dt2)(addr2, data2) ## read_data(pm, dt1)(addr1)))(s) = data((q ## read_data(pm, dt1)(addr1)))(s) </pre>

Repeatedly Skolemizing and flattening,
Expanding the definition of pm_q_prop_read,
Using lemma plain_memory_read_write_other_single_data,
Using lemma pm_q_prop_single_read[State, Data1],
Installing automatic rewrites from: valid_in_mem read_data read_data_valid_in_mem ## lift
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Expanding the definition of ##,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of plain_memory_read_write_other_q_data_stmt.
Q.E.D.

C.181.10

Plain_Mem_Rewrites7.plain_memory_read_read_q_data_stmt_TCC1

Terse proof for plain_memory_read_read_q_data_stmt_TCC1.

plain_memory_read_read_q_data_stmt_TCC1:

$$\begin{array}{l}
 \{1\} \quad \forall (\text{addr1}, \text{addr2}: \text{Address}, \text{dt1}: (\text{interpreted_data_type?}[\text{Data1}]), \\
 \quad \text{dt2}: (\text{interpreted_data_type?}[\text{Data2}]), \text{pm}: \text{Plain_Memory}[\text{State}], s: \text{State}, \\
 \quad q: [\text{State} \rightarrow \text{StmtResult}[\text{State}]]) : \\
 \text{plain_memory?}(\text{pm}) \wedge \\
 \text{in_blessed_memory?}(\text{dt1}, \text{addr1}, (\text{pm}'\text{ro_addr} \cup \text{pm}'\text{rw_addr})) \wedge \\
 \text{in_blessed_memory?}(\text{dt2}, \text{addr2}, (\text{pm}'\text{ro_addr} \cup \text{pm}'\text{rw_addr})) \wedge \\
 \text{pm_q_prop_read}(\text{pm}, \text{dt2}, \text{addr2})(q(s)) \wedge \text{pm_q_prop_read}(\text{pm}, \text{dt1}, \text{addr1})(q(s)) \\
 \supset \\
 \text{OK?}[\text{State}, \text{Data1}] \\
 \quad ((\#\#[\text{State}, \text{Data2}, \text{Data1}] \\
 \quad \quad (\#\#[\text{State}, \text{Data2}](q, \text{read_data}[\text{State}, \text{Data2}](\text{pm}, \text{dt2})(\text{addr2})), \\
 \quad \quad \text{read_data}[\text{State}, \text{Data1}](\text{pm}, \text{dt1})(\text{addr1}))) \\
 \quad (s))
 \end{array}$$

Expanding the definition of pm_q_prop_read,

Repeatedly Skolemizing and flattening,

Using lemma plain_memory_read_read_ok[State, Data2, Data1],

Installing automatic rewrites from: valid_in_mem read_data read_data_valid_in_mem ## lift

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of plain_memory_read_read_q_data_stmt_TCC1.

Q.E.D.

C.181.11

Plain_Mem_Rewrites7.plain_memory_read_read_q_data_stmt_TCC2

Terse proof for plain_memory_read_read_q_data_stmt_TCC2.

plain_memory_read_read_q_data_stmt_TCC2:

$$\begin{array}{l}
 \{1\} \quad \forall (\text{addr1}, \text{addr2}: \text{Address}, \text{dt1}: (\text{interpreted_data_type?}[\text{Data1}]), \\
 \quad \text{dt2}: (\text{interpreted_data_type?}[\text{Data2}]), \text{pm}: \text{Plain_Memory}[\text{State}], s: \text{State}, \\
 \quad q: [\text{State} \rightarrow \text{StmtResult}[\text{State}]]) : \\
 \text{plain_memory?}(\text{pm}) \wedge \\
 \text{in_blessed_memory?}(\text{dt1}, \text{addr1}, (\text{pm}'\text{ro_addr} \cup \text{pm}'\text{rw_addr})) \wedge \\
 \text{in_blessed_memory?}(\text{dt2}, \text{addr2}, (\text{pm}'\text{ro_addr} \cup \text{pm}'\text{rw_addr})) \wedge \\
 \text{pm_q_prop_read}(\text{pm}, \text{dt2}, \text{addr2})(q(s)) \wedge \text{pm_q_prop_read}(\text{pm}, \text{dt1}, \text{addr1})(q(s)) \\
 \supset \\
 \text{OK?}[\text{State}, \text{Data1}] \\
 \quad ((\#\#[\text{State}, \text{Data1}](q, \text{read_data}[\text{State}, \text{Data1}](\text{pm}, \text{dt1})(\text{addr1}))) (s))
 \end{array}$$

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of plain_memory_read_read_q_data_stmt_TCC2.

Q.E.D.

C.181.12

Plain_Mem_Rewrites7.plain_memory_read_read_q_data_stmt

Terse proof for plain_memory_read_read_q_data_stmt.

plain_memory_read_read_q_data_stmt:

$\{1\} \quad \forall (\text{addr1, addr2: Address, dt1: (interpreted_data_type?[Data1]),}$ $\quad \text{dt2: (interpreted_data_type?[Data2]), pm: Plain_Memory[State], s: State,}$ $\quad q: [State \rightarrow \text{StmtResult}[State]]):$ $\text{plain_memory?}(pm) \wedge$ $\text{in_blessed_memory?}(dt1, \text{addr1}, (pm'ro_addr \cup pm'rw_addr)) \wedge$ $\text{in_blessed_memory?}(dt2, \text{addr2}, (pm'ro_addr \cup pm'rw_addr)) \wedge$ $\text{pm_q_prop_read}(pm, dt2, \text{addr2})(q(s)) \wedge \text{pm_q_prop_read}(pm, dt1, \text{addr1})(q(s))$ \supset $\text{data}((q \text{ \#\# } \text{read_data}(pm, dt2)(\text{addr2}) \text{ \#\# } \text{read_data}(pm, dt1)(\text{addr1}))(s)) =$ $\text{data}((q \text{ \#\# } \text{read_data}(pm, dt1)(\text{addr1}))(s))$
--

Expanding the definition of pm_q_prop_read,

Repeatedly Skolemizing and flattening,

Using lemma plain_memory_read_read[State, Data2, Data1],

Installing automatic rewrites from: `\#\# lift valid_in_mem read_data read_data_valid_in_mem`

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of plain_memory_read_read_q_data_stmt.

Q.E.D.

C.181.13 Plain_Mem_Rewrites7.pm_q_prop_read_e2s

Terse proof for pm_q_prop_read_e2s.

pm_q_prop_read_e2s:

$\{1\} \quad \forall (\text{addr1: Address, dt1: (interpreted_data_type?[Data1]),}$ $\quad \text{expr1: [State} \rightarrow \text{ExprResult[State, Data2]], pm: Plain_Memory[State], s: State):$ $\text{pm_q_prop_read}(pm, dt1, \text{addr1})(\text{expr1}(s)) \supset$ $\text{pm_q_prop_read}(pm, dt1, \text{addr1})(\text{e2s}(\text{expr1})(s))$
--

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of pm_q_prop_read_e2s.

Q.E.D.

C.181.14 Plain_Mem_Rewrites7.pm_q_prop_read_stmt_e2s

Terse proof for pm_q_prop_read_stmt_e2s.

pm_q_prop_read_stmt_e2s:

$\{1\} \quad \forall (\text{addr1: Address, dt1: (interpreted_data_type?[Data1]),}$ $\quad \text{expr1: [State} \rightarrow \text{ExprResult[State, Data2]], pm: Plain_Memory[State], s: State,}$ $\quad \text{stmt: [State} \rightarrow \text{StmtResult[State]]):}$ $\text{pm_q_prop_read}(pm, dt1, \text{addr1})(\text{stmt}(s)) \supset$ $\text{pm_q_prop_read}(pm, dt1, \text{addr1})((\text{stmt} \text{ \#\# } \text{e2s}(\text{expr1}))(s)) =$ $\text{pm_q_prop_read}(pm, dt1, \text{addr1})((\text{stmt} \text{ \#\# } \text{expr1})(s))$
--

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of pm_q_prop_read_stmt_e2s.

Q.E.D.

C.182 Proofs for Plain_Mem_Rewrites8 (plain_memory_rewrites.pvs)

C.182.1

Plain_Mem_Rewrites8.plain_memory_read_write_other_q_data_expr_TCC1

Terse proof for plain_memory_read_write_other_q_data_expr_TCC1.

plain_memory_read_write_other_q_data_expr_TCC1:

$$\begin{array}{l} \{1\} \quad \forall (\text{addr1}, \text{addr2}: \text{Address}, \text{data2}: \text{Data2}, \text{dt1}: (\text{interpreted_data_type?}[\text{Data1}]), \\ \quad \text{dt2}: (\text{interpreted_data_type?}[\text{Data2}]), \text{pm}: \text{Plain_Memory}[\text{State}], \text{s}: \text{State}, \\ \quad \text{q}: [\text{State} \rightarrow \text{ExprResult}[\text{State}, \text{Data_q}]]): \\ \text{plain_memory?}(\text{pm}) \wedge \\ \text{in_blessed_memory?}(\text{dt1}, \text{addr1}, (\text{pm}'\text{ro_addr} \cup \text{pm}'\text{rw_addr})) \wedge \\ \text{in_blessed_memory?}(\text{dt2}, \text{addr2}, \text{pm}'\text{rw_addr}) \wedge \\ \text{blocks_disjoint?}(\text{addr1}, \text{size}(\text{uidt}(\text{dt1})), \text{addr2}, \text{size}(\text{uidt}(\text{dt2}))) \wedge \\ \text{pm_q_prop_read}(\text{pm}, \text{dt1}, \text{addr1})(\text{q}(\text{s})) \\ \supset \\ \text{OK?}[\text{State}, \text{Data1}] \\ \quad ((\#\#[\text{State}, \text{Unit}, \text{Data1}] \\ \quad \quad (\#\#[\text{State}, \text{Data_q}, \text{Unit}] \\ \quad \quad \quad (\text{q}, \text{write_data}[\text{State}, \text{Data2}](\text{pm}, \text{dt2})(\text{addr2}, \text{data2})), \\ \quad \quad \quad \text{read_data}[\text{State}, \text{Data1}](\text{pm}, \text{dt1})(\text{addr1}))) \\ \quad \quad (s)) \end{array}$$

Repeatedly Skolemizing and flattening,

Using lemma pm_q_prop_read_write_other_q_expr[State, Data1, Data2, Data_q],

Expanding the definition of pm_q_prop_read,

Installing automatic rewrites from: ## lift

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of plain_memory_read_write_other_q_data_expr_TCC1.

Q.E.D.

C.182.2

Plain_Mem_Rewrites8.plain_memory_read_write_other_q_data_expr_TCC2

Terse proof for plain_memory_read_write_other_q_data_expr_TCC2.

plain_memory_read_write_other_q_data_expr_TCC2:

$\{1\} \quad \forall (\text{addr1}, \text{addr2}: \text{Address}, \text{dt1}: (\text{interpreted_data_type?}[\text{Data1}]), \\ \text{dt2}: (\text{interpreted_data_type?}[\text{Data2}]), \text{pm}: \text{Plain_Memory}[\text{State}], \text{s}: \text{State}, \\ \text{q}: [\text{State} \rightarrow \text{ExprResult}[\text{State}, \text{Data_q}]]): \\ \text{plain_memory?}(\text{pm}) \wedge \\ \text{in_blessed_memory?}(\text{dt1}, \text{addr1}, (\text{pm}'\text{ro_addr} \cup \text{pm}'\text{rw_addr})) \wedge \\ \text{in_blessed_memory?}(\text{dt2}, \text{addr2}, \text{pm}'\text{rw_addr}) \wedge \\ \text{blocks_disjoint?}(\text{addr1}, \text{size}(\text{uidt}(\text{dt1})), \text{addr2}, \text{size}(\text{uidt}(\text{dt2}))) \wedge \\ \text{pm_q_prop_read}(\text{pm}, \text{dt1}, \text{addr1})(\text{q}(\text{s})) \\ \supset \\ \text{OK?}[\text{State}, \text{Data1}] \\ ((\#\#[\text{State}, \text{Data_q}, \text{Data1}](\text{q}, \text{read_data}[\text{State}, \text{Data1}](\text{pm}, \text{dt1})(\text{addr1}))) (\text{s}))$
--

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of plain_memory_read_write_other_q_data_expr_TCC2.

Q.E.D.

C.182.3

Plain_Mem_Rewrites8.plain_memory_read_write_other_q_data_expr

Terse proof for plain_memory_read_write_other_q_data_expr.

plain_memory_read_write_other_q_data_expr:

$\{1\} \quad \forall (\text{addr1}, \text{addr2}: \text{Address}, \text{data2}: \text{Data2}, \text{dt1}: (\text{interpreted_data_type?}[\text{Data1}]), \\ \text{dt2}: (\text{interpreted_data_type?}[\text{Data2}]), \text{pm}: \text{Plain_Memory}[\text{State}], \text{s}: \text{State}, \\ \text{q}: [\text{State} \rightarrow \text{ExprResult}[\text{State}, \text{Data_q}]]): \\ \text{plain_memory?}(\text{pm}) \wedge \\ \text{in_blessed_memory?}(\text{dt1}, \text{addr1}, (\text{pm}'\text{ro_addr} \cup \text{pm}'\text{rw_addr})) \wedge \\ \text{in_blessed_memory?}(\text{dt2}, \text{addr2}, \text{pm}'\text{rw_addr}) \wedge \\ \text{blocks_disjoint?}(\text{addr1}, \text{size}(\text{uidt}(\text{dt1})), \text{addr2}, \text{size}(\text{uidt}(\text{dt2}))) \wedge \\ \text{pm_q_prop_read}(\text{pm}, \text{dt1}, \text{addr1})(\text{q}(\text{s})) \\ \supset \\ \text{data}((\text{q} \#\# \text{write_data}(\text{pm}, \text{dt2})(\text{addr2}, \text{data2}) \#\# \text{read_data}(\text{pm}, \text{dt1})(\text{addr1}))) (\text{s}) \\ = \text{data}((\text{q} \#\# \text{read_data}(\text{pm}, \text{dt1})(\text{addr1}))) (\text{s}))$

Repeatedly Skolemizing and flattening,

Expanding the definition of pm_q_prop_read,

Using lemma plain_memory_read_write_other_single_data[State, Data1, Data2],

Expanding the definition of ##,

Expanding the definition of ##,

Using lemma pm_q_prop_single_read[State, Data1],

Installing automatic rewrites from: valid_in_mem read_data read_data_valid_in_mem

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of plain_memory_read_write_other_q_data_expr.

Q.E.D.

C.183 Proofs for Plain_Memory (plain_memory.pvs)

C.183.1

Plain_Memory.plain_memory_invariant_read_transformers_ro_addr

Terse proof for plain_memory_invariant_read_transformers_ro_addr.

plain_memory_invariant_read_transformers_ro_addr:

$\{1\} \quad \forall (pm: Plain_Memory):$ $\quad plain_memory?(pm) \supset$ $\quad transformer_invariant?(pm'states, memory_read_transformers(pm'mem, pm'ro_addr))$

Repeatedly Skolemizing and flattening,
Expanding the definition of plain_memory?,
Applying disjunctive simplification to flatten sequent,
Keeping (-1 1) and hiding *,
Using lemma unchanged_memory_invariant_invariant,
Simplifying, rewriting, and recording with decision procedures,
Hiding formulas: -2,
Using lemma transformer_invariant_mono_transformers,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Keeping (1) and hiding *,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of plain_memory_invariant_read_transformers_ro_addr.
Q.E.D.

C.183.2

Plain_Memory.plain_memory_invariant_read_transformers_rw_addr

Terse proof for plain_memory_invariant_read_transformers_rw_addr.

plain_memory_invariant_read_transformers_rw_addr:

$\{1\} \quad \forall (pm: Plain_Memory):$ $\quad plain_memory?(pm) \supset$ $\quad transformer_invariant?(pm'states, memory_read_transformers(pm'mem, pm'rw_addr))$

Skolemizing and flattening,
Expanding the definition of plain_memory?,
Expanding the definition of changed_memory_invariant?,
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of plain_memory_invariant_read_transformers_rw_addr.
Q.E.D.

C.183.3

Plain_Memory.plain_memory_invariant_read_transformers_ro_rw_addr

Terse proof for plain_memory_invariant_read_transformers_ro_rw_addr.

plain_memory_invariant_read_transformers_ro_rw_addr:

{1}	$\forall (pm: Plain_Memory):$ $plain_memory?(pm) \supset$ $transformer_invariant?(pm' states,$ <div style="text-align: right;">$memory_read_transformers(pm' mem, (pm' ro_addr \cup pm' rw_addr)))$</div>
-----	--

Repeatedly Skolemizing and flattening,
 Using lemma plain_memory_invariant_read_transformers_ro_addr,
 Using lemma plain_memory_invariant_read_transformers_rw_addr,
 Simplifying, rewriting, and recording with decision procedures,
 Using lemma transformer_invariant_union_transformers,
 Simplifying, rewriting, and recording with decision procedures,
 Rewriting using memory_read_transformers_union, matching in *,
 This completes the proof of plain_memory_invariant_read_transformers_ro_rw_addr.
 Q.E.D.

C.183.4

Plain_Memory.plain_memory_invariant_write_transformers_rw_addr

Terse proof for plain_memory_invariant_write_transformers_rw_addr.

plain_memory_invariant_write_transformers_rw_addr:

{1}	$\forall (pm: Plain_Memory):$ $plain_memory?(pm) \supset$ $transformer_invariant?(pm' states, memory_write_transformers(pm' mem, pm' rw_addr))$
-----	---

Skolemizing and flattening,
 Expanding the definition of plain_memory?,
 Expanding the definition of changed_memory_invariant?,
 Simplifying, rewriting, and recording with decision procedures,
 This completes the proof of plain_memory_invariant_write_transformers_rw_addr.
 Q.E.D.

C.183.5 Plain_Memory.plain_memory_invariant

Terse proof for plain_memory_invariant.

plain_memory_invariant:

{1}	$\forall (pm: Plain_Memory):$ $plain_memory?(pm) \supset$ $transformer_invariant?(pm' states,$ <div style="text-align: right;">$(pm' other_actions \cup ((memory_read_transformers(pm' mem, (pm' ro_addr \cup pm'$</div>
-----	---

Expanding the definition of plain_memory?,
 Repeatedly Skolemizing and flattening,
 Expanding the definition of unchanged_memory_invariant?,
 Applying disjunctive simplification to flatten sequent,
 Keeping (-1 -3 1) and hiding *,
 Using lemma transformer_invariant_union_transformers,

plain_memory_unchanged_invariant:

$$\{1\} \quad \forall (pm: \text{Plain_Memory}):$$

$$\text{plain_memory?}(pm) \supset$$

$$\text{unchanged_memory_invariant?}(pm' \text{mem}, pm' \text{states},$$

$$\quad ((pm' \text{other_actions} \cup \text{memory_read_transformers}(pm' \text{mem}, (pm' \text{ro_addr}$$

$$\quad (pm' \text{ro_addr} \cup pm' \text{rw_addr}))$$

Expanding the definition of plain_memory?,
 Repeatedly Skolemizing and flattening,
 This completes the proof of plain_memory_unchanged_invariant.
 Q.E.D.

C.183.7

Plain_Memory.plain_memory_unchanged_memory_invariant_write

Terse proof for plain_memory_unchanged_memory_invariant_write.

plain_memory_unchanged_memory_invariant_write:

$$\{1\} \quad \forall (pm: \text{Plain_Memory}):$$

$$\text{plain_memory?}(pm) \supset$$

$$\text{unchanged_memory_invariant?}(pm' \text{mem}, pm' \text{states}, \text{mem-}$$

$$\text{ory_write_transformers}(pm' \text{mem}, pm' \text{rw_addr}),$$

$$\quad pm' \text{ro_addr})$$

Expanding the definition of plain_memory?,
 Repeatedly Skolemizing and flattening,
 This completes the proof of plain_memory_unchanged_memory_invariant_write.
 Q.E.D.

C.183.8

Plain_Memory.plain_memory_unchanged_memory_write_invariant

Terse proof for plain_memory_unchanged_memory_write_invariant.

plain_memory_unchanged_memory_write_invariant:

$$\{1\} \quad \forall (pm: \text{Plain_Memory}):$$

$$\text{plain_memory?}(pm) \supset \text{unchanged_memory_write_invariant?}(pm' \text{mem}, pm' \text{states}, pm' \text{rw_addr})$$

Expanding the definition of plain_memory?,
 Repeatedly Skolemizing and flattening,
 This completes the proof of plain_memory_unchanged_memory_write_invariant.
 Q.E.D.

C.183.9 Plain_Memory.plain_memory_changed_memory_invariant

Terse proof for plain_memory_changed_memory_invariant.

plain_memory_changed_memory_invariant:

$$\{1\} \quad \forall (pm: \text{Plain_Memory}):$$

$$\text{plain_memory?}(pm) \supset \text{changed_memory_invariant?}(pm' \text{mem}, pm' \text{states}, pm' \text{rw_addr})$$

Expanding the definition of `plain_memory?`,
 Repeatedly Skolemizing and flattening,
 This completes the proof of `plain_memory_changed_memory_invariant`.
 Q.E.D.

C.183.10 Plain_Memory.plain_memory_invariant_read_block

Terse proof for `plain_memory_invariant_read_block`.

`plain_memory_invariant_read_block`:

$$\{1\} \quad \forall (\text{addr: Address, pm: Plain_Memory, size: nat}):$$

$$\text{plain_memory?}(pm) \wedge (\text{address_block}(\text{addr, size}) \subseteq (\text{pm}'\text{ro_addr} \cup \text{pm}'\text{rw_addr})) \supset$$

$$\text{transformer_invariant?}(\text{pm}'\text{states,}$$

$$\text{memory_read_transformers}(\text{pm}'\text{mem, ad-}$$

$$\text{dress_block}(\text{addr, size}))$$

Repeatedly Skolemizing and flattening,
 Installing automatic rewrites from: `memory_read_transformers_mono`
 Using lemma `plain_memory_invariant_read_transformers_ro_rw_addr`,
 Simplifying, rewriting, and recording with decision procedures,
 Using lemma `transformer_invariant_mono_transformers`,
 Simplifying, rewriting, and recording with decision procedures,
 This completes the proof of `plain_memory_invariant_read_block`.
 Q.E.D.

C.183.11 Plain_Memory.plain_memory_invariant_write_block

Terse proof for `plain_memory_invariant_write_block`.

`plain_memory_invariant_write_block`:

$$\{1\} \quad \forall (\text{addr: Address, pm: Plain_Memory, size: nat}):$$

$$\text{plain_memory?}(pm) \wedge (\text{address_block}(\text{addr, size}) \subseteq \text{pm}'\text{rw_addr}) \supset$$

$$\text{transformer_invariant?}(\text{pm}'\text{states,}$$

$$\text{memory_write_transformers}(\text{pm}'\text{mem, ad-}$$

$$\text{dress_block}(\text{addr, size}))$$

Repeatedly Skolemizing and flattening,
 Using lemma `plain_memory_invariant_write_transformers_rw_addr`,
 Simplifying, rewriting, and recording with decision procedures,
 Installing automatic rewrites from: `memory_write_transformers_mono`
 Rewriting using `transformer_invariant_mono_transformers`, matching in * where `transformers_1`
 gets `memory_write_transformers(pm!1'mem, address_block(addr!1, size!1))`, `transformers_2` gets `mem-`
`ory_write_transformers(pm!1'mem, pm!1'rw_addr)`,
 This completes the proof of `plain_memory_invariant_write_block`.
 Q.E.D.

C.183.12

Plain_Memory.plain_memory_transformer_invariant_read_side_effects

Terse proof for `plain_memory_transformer_invariant_read_side_effects`.

plain_memory_transformer_invariant_read_side_effects:

$\{1\} \quad \forall (\text{addr: Address, pm: Plain_Memory, size: nat}):$ $\text{plain_memory?}(pm) \wedge (\text{address_block}(\text{addr}, \text{size}) \subseteq (\text{pm}'\text{ro_addr} \cup \text{pm}'\text{rw_addr})) \supset$ $\text{transformer_invariant?}(\text{pm}'\text{states},$ $\text{memory_read_side_effect_super_transformers}(\text{pm}'\text{mem},$ $\text{address_block}(\text{addr}, \text{size}))$

Expanding the definition of plain_memory?,

Repeatedly Skolemizing and flattening,

Using lemma transformer_invariant_mono_transformers,

Using lemma unchanged_memory_invariant_invariant,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Keeping (-9 1) and hiding *,

Rewriting using subset_transitive, matching in * where a gets memory_read_side_effect_super_transformers(pm!1'mem, address_block(addr!1, size!1)), b gets memory_read_side_effect_super_transformers(pm!1'mem, union(pm!1'ro_addr, pm!1'rw_addr)), c gets union(union(pm!1'other_actions, memory_read_transformers(pm!1'mem, union(pm!1'ro_addr, pm!1'rw_addr))), union(memory_read_side_effect_super_transformers(pm!1'mem, union(pm!1'ro_addr, pm!1'rw_addr)), memory_write_side_effect_super_transformers(pm!1'mem, pm!1'rw_addr))),

we get 2 subgoals:

plain_memory_transformer_invariant_read_side_effects.1:

$\{-1\} \quad \text{side_effect_content_unchanged}(\text{pm}'\text{rw_addr}, \text{pm}'\text{states}, \text{memory_write_side_effect}(\text{pm}'\text{mem}))$
$\{1\} \quad (\text{memory_read_side_effect_super_transformers}(\text{pm}'\text{mem}, \text{address_block}(\text{addr}', \text{size}')) \subseteq \text{memory_read_side_effect_super_transformers}(\text{pm}'\text{mem}, \text{address_block}(\text{addr}', \text{size}')) \subseteq ((\text{pm}'\text{other_actions} \cup \text{memory_write_side_effect_super_transformers}(\text{pm}'\text{mem}, \text{address_block}(\text{addr}', \text{size}'))))$
$\{2\} \quad (\text{memory_read_side_effect_super_transformers}(\text{pm}'\text{mem}, \text{address_block}(\text{addr}', \text{size}')) \subseteq ((\text{pm}'\text{other_actions} \cup \text{memory_write_side_effect_super_transformers}(\text{pm}'\text{mem}, \text{address_block}(\text{addr}', \text{size}'))))$

Hiding formulas: 2,

Expanding the definition of subset?,

Expanding the definition of member,

Repeatedly Skolemizing and flattening,

Expanding the definition of memory_read_side_effect_super_transformers,

Repeatedly Skolemizing and flattening,

Instantiating quantified variables,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Keeping (-2 -4 1) and hiding *,

Rewriting using subset_transitive, matching in * where a gets address_block(a!1, length(bl!1)), b gets address_block(addr!1, size!1), c gets union(pm!1'ro_addr, pm!1'rw_addr),

This completes the proof of plain_memory_transformer_invariant_read_side_effects.1.

plain_memory_transformer_invariant_read_side_effects.2:

$\{-1\} \quad \text{side_effect_content_unchanged}(\text{pm}'\text{rw_addr}, \text{pm}'\text{states}, \text{memory_write_side_effect}(\text{pm}'\text{mem}))$
$\{1\} \quad (\text{memory_read_side_effect_super_transformers}(\text{pm}'\text{mem}, (\text{pm}'\text{ro_addr} \cup \text{pm}'\text{rw_addr})) \subseteq ((\text{pm}'\text{other_actions} \cup \text{memory_write_side_effect_super_transformers}(\text{pm}'\text{mem}, \text{address_block}(\text{addr}', \text{size}'))))$
$\{2\} \quad (\text{memory_read_side_effect_super_transformers}(\text{pm}'\text{mem}, \text{address_block}(\text{addr}', \text{size}')) \subseteq ((\text{pm}'\text{other_actions} \cup \text{memory_write_side_effect_super_transformers}(\text{pm}'\text{mem}, \text{address_block}(\text{addr}', \text{size}'))))$

Hiding formulas: (-1 2),

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of plain_memory_transformer_invariant_read_side_effects.2.

Q.E.D.

C.183.13**Plain_Memory.plain_memory_transformer_invariant_write_side_effects**

Terse proof for `plain_memory_transformer_invariant_write_side_effects`.

`plain_memory_transformer_invariant_write_side_effects:`

$\{1\} \quad \forall (\text{addr}: \text{Address}, \text{pm}: \text{Plain_Memory}, \text{size}: \text{nat}):$ $\text{plain_memory?}(\text{pm}) \wedge (\text{address_block}(\text{addr}, \text{size}) \subseteq \text{pm}'\text{rw_addr}) \supset$ $\text{transformer_invariant?}(\text{pm}'\text{states},$ $\text{memory_write_side_effect_super_transformers}(\text{pm}'\text{mem},$ $\text{address_block}(\text{addr}, \text{size}))$	$)$
---	-----

Expanding the definition of `plain_memory?`,

Repeatedly Skolemizing and flattening,

Using lemma `transformer_invariant_mono_transformers`,

Using lemma `unchanged_memory_invariant_invariant`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Keeping (-9 1) and hiding *,

Rewriting using `subset_transitive`, matching in * where a gets `memory_write_side_effect_super_transformers(pm!1'rw_addr)`, b gets `memory_write_side_effect_super_transformers(pm!1'mem, pm!1'rw_addr)`, c gets `union(union(pm!1'other_actions, memory_read_transformers(pm!1'mem, union(pm!1'ro_addr, pm!1'rw_addr))), union(memory_read_side_effect_super_transformers(pm!1'mem, union(pm!1'ro_addr, pm!1'rw_addr)), memory_write_side_effect_super_transformers(pm!1'mem, pm!1'rw_addr))`,

we get 2 subgoals:

`plain_memory_transformer_invariant_write_side_effects.1:`

$\{-1\} \quad \text{side_effect_content_unchanged}(\text{pm}'\text{rw_addr}, \text{pm}'\text{states}, \text{memory_write_side_effect}(\text{pm}'\text{mem}))$	$\{1\} \quad (\text{memory_write_side_effect_super_transformers}(\text{pm}'\text{mem}, \text{address_block}(\text{addr}', \text{size}')) \subseteq \text{memory_write_side_effect_super_transformers}(\text{pm}'\text{mem}, \text{address_block}(\text{addr}', \text{size}')) \subseteq ((\text{pm}'\text{other_actions} \cup \text{memory_read_transformers}(\text{pm}'\text{mem}, \text{union}(\text{pm}'\text{ro_addr}, \text{pm}'\text{rw_addr}))))$
$\{2\} \quad (\text{memory_write_side_effect_super_transformers}(\text{pm}'\text{mem}, \text{address_block}(\text{addr}', \text{size}')) \subseteq ((\text{pm}'\text{other_actions} \cup \text{memory_read_transformers}(\text{pm}'\text{mem}, \text{union}(\text{pm}'\text{ro_addr}, \text{pm}'\text{rw_addr}))))$	$\subseteq ((\text{pm}'\text{other_actions} \cup \text{memory_read_transformers}(\text{pm}'\text{mem}, \text{union}(\text{pm}'\text{ro_addr}, \text{pm}'\text{rw_addr}))))$

Hiding formulas: 2,

Expanding the definition of `subset?`,

Expanding the definition of `member`,

Repeatedly Skolemizing and flattening,

Expanding the definition of `memory_write_side_effect_super_transformers`,

Repeatedly Skolemizing and flattening,

Instantiating quantified variables,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Keeping (-2 -4 1) and hiding *,

Rewriting using `subset_transitive`, matching in * where a gets `address_block(a!1, length(bl!1))`, b gets `address_block(addr!1, size!1)`, c gets `pm!1'rw_addr`,

This completes the proof of `plain_memory_transformer_invariant_write_side_effects.1`.

`plain_memory_transformer_invariant_write_side_effects.2:`

$\{-1\} \quad \text{side_effect_content_unchanged}(\text{pm}'\text{rw_addr}, \text{pm}'\text{states}, \text{memory_write_side_effect}(\text{pm}'\text{mem}))$	$\{1\} \quad (\text{memory_write_side_effect_super_transformers}(\text{pm}'\text{mem}, \text{pm}'\text{rw_addr}) \subseteq ((\text{pm}'\text{other_actions} \cup \text{memory_read_transformers}(\text{pm}'\text{mem}, \text{union}(\text{pm}'\text{ro_addr}, \text{pm}'\text{rw_addr}))))$
$\{2\} \quad (\text{memory_write_side_effect_super_transformers}(\text{pm}'\text{mem}, \text{address_block}(\text{addr}', \text{size}')) \subseteq ((\text{pm}'\text{other_actions} \cup \text{memory_read_transformers}(\text{pm}'\text{mem}, \text{union}(\text{pm}'\text{ro_addr}, \text{pm}'\text{rw_addr}))))$	$\subseteq ((\text{pm}'\text{other_actions} \cup \text{memory_read_transformers}(\text{pm}'\text{mem}, \text{union}(\text{pm}'\text{ro_addr}, \text{pm}'\text{rw_addr}))))$

Hiding formulas: (-1 2),

Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of plain_memory_transformer_invariant_write_side_effects.2.
 Q.E.D.

C.183.14

Plain_Memory.plain_memory_transformer_invariant_read_list_block

Terse proof for plain_memory_transformer_invariant_read_list_block.

plain_memory_transformer_invariant_read_list_block:

{1}	$\forall (\text{addr}: \text{Address}, \text{pm}: \text{Plain_Memory}, \text{size}: \text{nat}):$ $\text{plain_memory?}(\text{pm}) \wedge (\text{address_block}(\text{addr}, \text{size}) \subseteq (\text{pm}'\text{ro_addr} \cup \text{pm}'\text{rw_addr})) \supset$ $\text{transformer_invariant?}(\text{pm}'\text{states},$ $\qquad (\text{memory_read_transformers}(\text{pm}'\text{mem}, \text{address_block}(\text{addr}, \text{size})) \cup \text{memory_r...})$
-----	--

Repeatedly Skolemizing and flattening,
 Using lemma plain_memory_transformer_invariant_read_side_effects,
 Using lemma plain_memory_invariant_read_block,
 Rewriting using transformer_invariant_union_transformers, matching in *,
 This completes the proof of plain_memory_transformer_invariant_read_list_block.
 Q.E.D.

C.183.15

Plain_Memory.plain_memory_transformer_invariant_write_list_block

Terse proof for plain_memory_transformer_invariant_write_list_block.

plain_memory_transformer_invariant_write_list_block:

{1}	$\forall (\text{addr}: \text{Address}, \text{pm}: \text{Plain_Memory}, \text{size}: \text{nat}):$ $\text{plain_memory?}(\text{pm}) \wedge (\text{address_block}(\text{addr}, \text{size}) \subseteq \text{pm}'\text{rw_addr}) \supset$ $\text{transformer_invariant?}(\text{pm}'\text{states},$ $\qquad (\text{memory_write_transformers}(\text{pm}'\text{mem}, \text{address_block}(\text{addr}, \text{size})) \cup \text{memory_w...})$
-----	--

Repeatedly Skolemizing and flattening,
 Using lemma plain_memory_invariant_write_block,
 Using lemma plain_memory_transformer_invariant_write_side_effects,
 Rewriting using transformer_invariant_union_transformers, matching in *,
 This completes the proof of plain_memory_transformer_invariant_write_list_block.
 Q.E.D.

C.183.16 Plain_Memory.plain_memory_transformers_ok_read_ro_rw

Terse proof for plain_memory_transformers_ok_read_ro_rw.

plain_memory_transformers_ok_read_ro_rw:

{1}	$\forall (\text{pm}: \text{Plain_Memory}):$ $\text{plain_memory?}(\text{pm}) \supset$ $\text{transformers_ok?}(\text{pm}'\text{states}, \text{memory_read_transformers}(\text{pm}'\text{mem}, (\text{pm}'\text{ro_addr} \cup \text{pm}'\text{rw_addr})))$
-----	--

Repeatedly Skolemizing and flattening,
 Installing automatic rewrites from: memory_read_transformers_mono

Expanding the definition of `plain_memory?`,
 Applying disjunctive simplification to flatten sequent,
 Keeping (-5 1) and hiding *,
 Using lemma `transformers_ok_mono_transformers`,
 Simplifying, rewriting, and recording with decision procedures,
 Keeping (1) and hiding *,
 Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of `plain_memory_transformers_ok_read_ro_rw`.
 Q.E.D.

C.183.17 Plain_Memory.plain_memory_transformers_ok_read_block

Terse proof for `plain_memory_transformers_ok_read_block`.

`plain_memory_transformers_ok_read_block`:

$\{1\} \quad \forall (\text{addr} : \text{Address}, \text{pm} : \text{Plain_Memory}, \text{size} : \text{nat}) : \\ \text{plain_memory?}(\text{pm}) \wedge (\text{address_block}(\text{addr}, \text{size}) \subseteq (\text{pm}'\text{ro_addr} \cup \text{pm}'\text{rw_addr})) \supset \\ \text{transformers_ok?}(\text{pm}'\text{states}, \text{memory_read_transformers}(\text{pm}'\text{mem}, \text{address_block}(\text{addr}, \text{size})))$

Repeatedly Skolemizing and flattening,
 Installing automatic rewrites from: `memory_read_transformers_mono`
 Rewriting using `transformers_ok_mono_transformers`, matching in * where `transformers_1` gets
`memory_read_transformers(pm!1'mem, address_block(addr!1, size!1))`, `transformers_2` gets `mem-`
`ory_read_transformers(pm!1'mem, union(pm!1'ro_addr, pm!1'rw_addr))`,
 we get 2 subgoals:

`plain_memory_transformers_ok_read_block.1`:

$\{-1\} \quad \text{size}' \geq 0$
$\{-2\} \quad \text{plain_memory?}(\text{pm}')$
$\{-3\} \quad (\text{address_block}(\text{addr}', \text{size}') \subseteq (\text{pm}'\text{ro_addr} \cup \text{pm}'\text{rw_addr}))$
$\{1\} \quad (\text{memory_read_transformers}(\text{pm}'\text{mem}, \text{address_block}(\text{addr}', \text{size}')) \subseteq \text{memory_read_transformers}(\text{pm}'\text{mem}, \text{union}(\text{pm}'\text{ro_addr}, \text{pm}'\text{rw_addr})))$
$\{2\} \quad \text{transformers_ok?}(\text{pm}'\text{states}, \text{memory_read_transformers}(\text{pm}'\text{mem}, \text{address_block}(\text{addr}', \text{size}')))$

Simplifying, rewriting, and recording with decision procedures,
 This completes the proof of `plain_memory_transformers_ok_read_block.1`.

`plain_memory_transformers_ok_read_block.2`:

$\{-1\} \quad \text{size}' \geq 0$
$\{-2\} \quad \text{plain_memory?}(\text{pm}')$
$\{-3\} \quad (\text{address_block}(\text{addr}', \text{size}') \subseteq (\text{pm}'\text{ro_addr} \cup \text{pm}'\text{rw_addr}))$
$\{1\} \quad \text{transformers_ok?}(\text{pm}'\text{states}, \text{memory_read_transformers}(\text{pm}'\text{mem}, (\text{pm}'\text{ro_addr} \cup \text{pm}'\text{rw_addr})))$
$\{2\} \quad \text{transformers_ok?}(\text{pm}'\text{states}, \text{memory_read_transformers}(\text{pm}'\text{mem}, \text{address_block}(\text{addr}', \text{size}')))$

Expanding the definition of `plain_memory?`,
 Applying disjunctive simplification to flatten sequent,
 Keeping (-6 1) and hiding *,
 Using lemma `transformers_ok_mono_transformers`,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 Keeping (1) and hiding *,
 Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of plain_memory_transformers_ok_read_block.2.
Q.E.D.

C.183.18 Plain_Memory.plain_memory_transformers_ok_write_rw

Terse proof for plain_memory_transformers_ok_write_rw.

plain_memory_transformers_ok_write_rw:

{1}	$\forall (pm: \text{Plain_Memory}):$ $\text{plain_memory?}(pm) \supset$ $\text{transformers_ok?}(pm' \text{states}, \text{memory_write_transformers}(pm' \text{mem}, pm' \text{rw_addr}))$
-----	--

Installing automatic rewrites from: subset_equal union_subset1 union_subset3 union_left union_right subset_singleton subset_bigger_union_right subset_bigger_union_left union_left union_right

Skolemizing and flattening,

Expanding the definition of plain_memory?,

Applying disjunctive simplification to flatten sequent,

Rewriting using transformers_ok_mono_transformers, matching in * where transformers_1 gets memory_write_transformers(pm!1'mem, pm!1'rw_addr), transformers_2 gets union(union(memory_read_transformers(pm!1'mem, union(pm!1'ro_addr, pm!1'rw_addr)), memory_write_transformers(pm!1'mem, pm!1'rw_addr)), union(memory_read_side_effect_super_transformers(pm!1'mem, union(pm!1'ro_addr, pm!1'rw_addr)), memory_write_side_effect_super_transformers(pm!1'mem, pm!1'rw_addr))),

This completes the proof of plain_memory_transformers_ok_write_rw.

Q.E.D.

C.183.19 Plain_Memory.plain_memory_transformers_ok_write_block

Terse proof for plain_memory_transformers_ok_write_block.

plain_memory_transformers_ok_write_block:

{1}	$\forall (addr: \text{Address}, pm: \text{Plain_Memory}, size: \text{nat}):$ $\text{plain_memory?}(pm) \wedge (\text{address_block}(addr, size) \subseteq pm' \text{rw_addr}) \supset$ $\text{transformers_ok?}(pm' \text{states}, \text{memory_write_transformers}(pm' \text{mem}, \text{address_block}(addr, size)))$
-----	---

Skolemizing and flattening,

Rewriting using transformers_ok_mono_transformers, matching in * where transformers_1 gets memory_write_transformers(pm!1'mem, address_block(addr!1, size!1)), transformers_2 gets memory_write_transformers(pm!1'mem, pm!1'rw_addr),

we get 2 subgoals:

plain_memory_transformers_ok_write_block.1:

{-1}	plain_memory?(pm')
{-2}	$(\text{address_block}(addr', size') \subseteq pm' \text{rw_addr})$
{1}	$(\text{memory_write_transformers}(pm' \text{mem}, \text{address_block}(addr', size')) \subseteq \text{memory_write_transformers}(pm' \text{mem},$
{2}	$\text{transformers_ok?}(pm' \text{states},$ $\text{memory_write_transformers}(pm' \text{mem}, \text{address_block}(addr', size')))$

Rewriting using memory_write_transformers_mono, matching in *,

This completes the proof of plain_memory_transformers_ok_write_block.1.

plain_memory_transformers_ok_write_block.2:

{-1}	plain_memory?(pm')
{-2}	(address_block(addr', size') \subseteq pm'rw_addr)
{1}	transformers_ok?(pm' states, memory_write_transformers(pm' mem, pm'rw_addr))
{2}	transformers_ok?(pm' states, memory_write_transformers(pm' mem, ad- dress_block(addr', size')))

Rewriting using plain_memory_transformers_ok_write_rw, matching in *,
This completes the proof of plain_memory_transformers_ok_write_block.2.
Q.E.D.

C.183.20

Plain_Memory.plain_memory_transformers_ok_read_side_effects_ro_rw

Terse proof for plain_memory_transformers_ok_read_side_effects_ro_rw.

plain_memory_transformers_ok_read_side_effects_ro_rw:

{1}	\forall (pm: Plain_Memory): plain_memory?(pm) \supset transformers_ok?(pm states, memory_read_side_effect_super_transformers(pm mem, (pm'ro_addr \cup pm'rw_
-----	--

Expanding the definition of plain_memory?,
Repeatedly Skolemizing and flattening,
Keeping (-5 1) and hiding *,
Rewriting using transformers_ok_mono_transformers, matching in * where transformers_1 gets mem-
ory_read_side_effect_super_transformers(pm!1 mem, union (pm!1 ro_addr, pm!1 rw_addr)), transform-
ers_2 gets union(union(memory_read_transformers(pm!1 mem, union (pm!1 ro_addr, pm!1 rw_addr)),
memory_write_transformers(pm!1 mem, pm!1 rw_addr)), union(memory_read_side_effect_super_transformers(pm!1
union (pm!1 ro_addr, pm!1 rw_addr)), memory_write_side_effect_super_transformers(pm!1 mem,
pm!1 rw_addr))),
Keeping 1 and hiding *,
Rewriting using subset_bigger_union_right, matching in *,
Rewriting using union_subset1, matching in *,
This completes the proof of plain_memory_transformers_ok_read_side_effects_ro_rw.
Q.E.D.

C.183.21

Plain_Memory.plain_memory_transformers_ok_read_side_effects_block

Terse proof for plain_memory_transformers_ok_read_side_effects_block.

plain_memory_transformers_ok_read_side_effects_block:

{1}	\forall (addr: Address, pm: Plain_Memory, size: nat): plain_memory?(pm) \wedge (address_block(addr, size) \subseteq (pm'ro_addr \cup pm'rw_addr)) \supset transformers_ok?(pm states, memory_read_side_effect_super_transformers(pm mem, ad- dress_block(addr, size)))
-----	---

Repeatedly Skolemizing and flattening,
 Using lemma plain_memory_transformers_ok_read_side_effects_ro_rw,
 Simplifying, rewriting, and recording with decision procedures,
 Installing automatic rewrites from: memory_read_side_effect_super_transformers_mono
 Using lemma transformers_ok_mono_transformers,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 This completes the proof of plain_memory_transformers_ok_read_side_effects_block.
 Q.E.D.

C.183.22

Plain_Memory.plain_memory_transformers_ok_write_side_effects_ro_rw

Terse proof for plain_memory_transformers_ok_write_side_effects_ro_rw.

plain_memory_transformers_ok_write_side_effects_ro_rw:

$\{1\} \quad \forall (pm: \text{Plain_Memory}):$ $\text{plain_memory?}(pm) \supset$ $\text{transformers_ok?}(pm' \text{states}, \text{memory_write_side_effect_super_transformers}(pm' \text{mem}, pm' \text{rw_addr}))$

Expanding the definition of plain_memory?,
 Repeatedly Skolemizing and flattening,
 Keeping (-5 1) and hiding *,
 Rewriting using transformers_ok_mono_transformers, matching in * where transformers_1
 gets memory_write_side_effect_super_transformers(pm!1'mem, pm!1'rw_addr), transformers_2 gets
 union(union(memory_read_transformers(pm!1'mem, union (pm!1'ro_addr, pm!1'rw_addr)), mem-
 ory_write_transformers(pm!1'mem, pm!1'rw_addr)), union(memory_read_side_effect_super_transformers(pm!1'mem,
 union (pm!1'ro_addr, pm!1'rw_addr)), memory_write_side_effect_super_transformers(pm!1'mem,
 pm!1'rw_addr))),
 Keeping 1 and hiding *,
 Rewriting using subset_bigger_union_right, matching in *,
 Rewriting using union_subset3, matching in *,
 This completes the proof of plain_memory_transformers_ok_write_side_effects_ro_rw.
 Q.E.D.

C.183.23

Plain_Memory.plain_memory_transformers_ok_write_side_effects_block

Terse proof for plain_memory_transformers_ok_write_side_effects_block.

plain_memory_transformers_ok_write_side_effects_block:

$\{1\} \quad \forall (\text{addr}: \text{Address}, pm: \text{Plain_Memory}, \text{size}: \text{nat}):$ $\text{plain_memory?}(pm) \wedge (\text{address_block}(\text{addr}, \text{size}) \subseteq pm' \text{rw_addr}) \supset$ $\text{transformers_ok?}(pm' \text{states},$ $\text{memory_write_side_effect_super_transformers}(pm' \text{mem},$ $\text{address_block}(\text{addr}, \text{size}))$

Repeatedly Skolemizing and flattening,
 Using lemma plain_memory_transformers_ok_write_side_effects_ro_rw,
 Simplifying, rewriting, and recording with decision procedures,
 Installing automatic rewrites from: memory_write_side_effect_super_transformers_mono

C Proof scripts

Using lemma `transformers_ok_mono_transformers`,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `plain_memory_transformers_ok_write_side_effects_block`.
Q.E.D.

C.183.24

Plain_Memory.plain_memory_transformers_ok_read_list_block

Terse proof for `plain_memory_transformers_ok_read_list_block`.

`plain_memory_transformers_ok_read_list_block`:

$$\{1\} \quad \forall (\text{addr}: \text{Address}, \text{pm}: \text{Plain_Memory}, \text{size}: \text{nat}):$$
$$\text{plain_memory?}(\text{pm}) \wedge (\text{address_block}(\text{addr}, \text{size}) \subseteq (\text{pm}'\text{ro_addr} \cup \text{pm}'\text{rw_addr})) \supset$$
$$\text{transformers_ok?}(\text{pm}'\text{states},$$
$$(\text{memory_read_transformers}(\text{pm}'\text{mem}, \text{address_block}(\text{addr}, \text{size})) \cup \text{memo})$$

Repeatedly Skolemizing and flattening,
Using lemma `plain_memory_transformers_ok_read_block`,
Using lemma `plain_memory_transformers_ok_read_side_effects_block`,
Rewriting using `transformers_ok_union_transformers`, matching in `*`,
This completes the proof of `plain_memory_transformers_ok_read_list_block`.
Q.E.D.

C.183.25

Plain_Memory.plain_memory_transformers_ok_write_list_block

Terse proof for `plain_memory_transformers_ok_write_list_block`.

`plain_memory_transformers_ok_write_list_block`:

$$\{1\} \quad \forall (\text{addr}: \text{Address}, \text{pm}: \text{Plain_Memory}, \text{size}: \text{nat}):$$
$$\text{plain_memory?}(\text{pm}) \wedge (\text{address_block}(\text{addr}, \text{size}) \subseteq \text{pm}'\text{rw_addr}) \supset$$
$$\text{transformers_ok?}(\text{pm}'\text{states},$$
$$(\text{memory_write_transformers}(\text{pm}'\text{mem}, \text{address_block}(\text{addr}, \text{size})) \cup \text{memo})$$

Repeatedly Skolemizing and flattening,
Using lemma `plain_memory_transformers_ok_write_block`,
Using lemma `plain_memory_transformers_ok_write_side_effects_block`,
Rewriting using `transformers_ok_union_transformers`, matching in `*`,
This completes the proof of `plain_memory_transformers_ok_write_list_block`.
Q.E.D.

C.183.26

Plain_Memory.plain_memory_unchanged_read_block_rw_addr

Terse proof for `plain_memory_unchanged_read_block_rw_addr`.

plain_memory_unchanged_read_block_rw_addr:

$$\{1\} \quad \forall (\text{addr: Address, pm: Plain_Memory, size: nat}):$$

$$\text{plain_memory?}(pm) \wedge (\text{address_block}(\text{addr, size}) \subseteq pm'rw_addr) \supset$$

$$\text{unchanged_memory_invariant?}(pm' \text{mem, pm' states,}$$

$$\text{memory_read_transformers}(pm' \text{mem, ad-}$$

$$\text{dress_block}(\text{addr, size}),$$

$$pm'rw_addr)$$

Repeatedly Skolemizing and flattening,
 Using lemma plain_memory_unchanged_invariant,
 Using lemma unchanged_memory_invariant_mono,
 Rewriting using union_commutative, matching in *,
 Rewriting using union_subset1, matching in *,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 Keeping (-4 1) and hiding *,
 Using lemma memory_read_transformers_mono,
 Rewriting using subset_bigger_union_left, matching in * where a gets address_block(addr!1, size!1),
 b gets pm!1'rw_addr, c gets pm!1'ro_addr,
 Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of plain_memory_unchanged_read_block_rw_addr.
 Q.E.D.

C.183.27 Plain_Memory.plain_memory_unchanged_read_ro_rw_addr

Terse proof for plain_memory_unchanged_read_ro_rw_addr.

plain_memory_unchanged_read_ro_rw_addr:

$$\{1\} \quad \forall (pm: Plain_Memory):$$

$$\text{plain_memory?}(pm) \supset$$

$$\text{unchanged_memory_invariant?}(pm' \text{mem, pm' states,}$$

$$\text{memory_read_transformers}(pm' \text{mem, (pm'ro_addr} \cup pm'rw_addr)),$$

$$(pm'ro_addr \cup pm'rw_addr))$$

Skolemizing and flattening,
 Installing automatic rewrites from: subset_equal union_subset1 union_subset3 union_left union_right
 subset_singleton subset_bigger_union_right subset_bigger_union_left union_left union_right
 Using lemma plain_memory_unchanged_invariant,
 Using lemma unchanged_memory_invariant_mono,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 This completes the proof of plain_memory_unchanged_read_ro_rw_addr.
 Q.E.D.

C.183.28

Plain_Memory.plain_memory_unchanged_read_block_ro_rw_addr

Terse proof for plain_memory_unchanged_read_block_ro_rw_addr.

`plain_memory_unchanged_read_block_ro_rw_addr:`

$$\{1\} \quad \forall (\text{addr: Address, pm: Plain_Memory, size: nat}):$$

$$\text{plain_memory?}(pm) \wedge (\text{address_block}(\text{addr, size}) \subseteq (\text{pm}'\text{ro_addr} \cup \text{pm}'\text{rw_addr})) \supset$$

$$\text{unchanged_memory_invariant?}(\text{pm}'\text{mem, pm}'\text{states,}$$

$$\text{memory_read_transformers}(\text{pm}'\text{mem, ad-}$$

$$\text{dress_block}(\text{addr, size})),$$

$$(\text{pm}'\text{ro_addr} \cup \text{pm}'\text{rw_addr}))$$

Skolemizing and flattening,

Using lemma `plain_memory_unchanged_read_ro_rw_addr`,

Simplifying, rewriting, and recording with decision procedures,

Using lemma `unchanged_memory_invariant_mono`,

Installing automatic rewrites from: `subset_equal memory_read_transformers_mono`

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `plain_memory_unchanged_read_block_ro_rw_addr`.

Q.E.D.

C.183.29

Plain_Memory.plain_memory_unchanged_write_block_ro_addr

Terse proof for `plain_memory_unchanged_write_block_ro_addr`.

`plain_memory_unchanged_write_block_ro_addr:`

$$\{1\} \quad \forall (\text{addr: Address, pm: Plain_Memory, size: nat}):$$

$$\text{plain_memory?}(pm) \wedge (\text{address_block}(\text{addr, size}) \subseteq \text{pm}'\text{rw_addr}) \supset$$

$$\text{unchanged_memory_invariant?}(\text{pm}'\text{mem, pm}'\text{states,}$$

$$\text{memory_write_transformers}(\text{pm}'\text{mem, ad-}$$

$$\text{dress_block}(\text{addr, size})),$$

$$\text{pm}'\text{ro_addr})$$

Repeatedly Skolemizing and flattening,

Expanding the definition of `plain_memory?`,

Applying disjunctive simplification to flatten sequent,

Using lemma `unchanged_memory_invariant_mono`,

Rewriting using `subset_reflexive`, matching in `*`,

Simplifying, rewriting, and recording with decision procedures,

Rewriting using `memory_write_transformers_mono[State]`, matching in `*`,

This completes the proof of `plain_memory_unchanged_write_block_ro_addr`.

Q.E.D.

C.183.30 Plain_Memory.plain_memory_unchanged_read_block

Terse proof for `plain_memory_unchanged_read_block`.

plain_memory_unchanged_read_block:

$\{1\} \quad \forall (\text{addr: Address, pm: Plain_Memory, size: nat}):$ $\text{plain_memory?}(pm) \wedge (\text{address_block}(\text{addr, size}) \subseteq (\text{pm}'\text{ro_addr} \cup \text{pm}'\text{rw_addr})) \supset$ $\text{unchanged_memory_invariant?}(\text{pm}'\text{mem, pm}'\text{states,}$ $\text{memory_read_transformers}(\text{pm}'\text{mem, ad-}$ $\text{dress_block}(\text{addr, size}),$ $\text{address_block}(\text{addr, size}))$

Repeatedly Skolemizing and flattening,
 Using lemma plain_memory_unchanged_read_block_ro_rw_addr,
 Simplifying, rewriting, and recording with decision procedures,
 Using lemma unchanged_memory_invariant_mono,
 Simplifying, rewriting, and recording with decision procedures,
 Rewriting using subset_reflexive, matching in *,
 This completes the proof of plain_memory_unchanged_read_block.
 Q.E.D.

C.183.31

Plain_Memory.plain_memory_unchanged_read_side_effect_block_ro_rw

Terse proof for plain_memory_unchanged_read_side_effect_block_ro_rw.

plain_memory_unchanged_read_side_effect_block_ro_rw:

$\{1\} \quad \forall (\text{addr: Address, pm: Plain_Memory, size: nat}):$ $\text{plain_memory?}(pm) \wedge (\text{address_block}(\text{addr, size}) \subseteq (\text{pm}'\text{ro_addr} \cup \text{pm}'\text{rw_addr})) \supset$ $\text{unchanged_memory_invariant?}(\text{pm}'\text{mem, pm}'\text{states,}$ $\text{memory_read_side_effect_super_transformers}(\text{pm}'\text{mem,}$ ad- dress_block $(\text{addr, size}),$ $(\text{pm}'\text{ro_addr} \cup \text{pm}'\text{rw_addr}))$
--

Repeatedly Skolemizing and flattening,
 Using lemma plain_memory_unchanged_invariant,
 Simplifying, rewriting, and recording with decision procedures,
 Using lemma unchanged_memory_invariant_mono,
 Installing automatic rewrites from: subset_equal subset_bigger_union_right subset_bigger_union_left
 memory_read_side_effect_super_transformers_mono
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 This completes the proof of plain_memory_unchanged_read_side_effect_block_ro_rw.
 Q.E.D.

C.183.32

Plain_Memory.plain_memory_unchanged_read_side_effect_block

Terse proof for plain_memory_unchanged_read_side_effect_block.

plain_memory_unchanged_read_side_effect_block:

{1}	$\forall (\text{addr}: \text{Address}, \text{pm}: \text{Plain_Memory}, \text{size}: \text{nat}):$ $\text{plain_memory?}(\text{pm}) \wedge (\text{address_block}(\text{addr}, \text{size}) \subseteq (\text{pm}'\text{ro_addr} \cup \text{pm}'\text{rw_addr})) \supset$ $\text{unchanged_memory_invariant?}(\text{pm}'\text{mem}, \text{pm}'\text{states},$ $\text{memory_read_side_effect_super_transformers}(\text{pm}'\text{mem}, \text{ad-}$ $\text{dress_block} \quad \text{address_block}(\text{addr}, \text{size})) \quad (\text{addr}, \text{s}$
-----	---

Repeatedly Skolemizing and flattening,
 Using lemma plain_memory_unchanged_read_side_effect_block_ro_rw,
 Simplifying, rewriting, and recording with decision procedures,
 Using lemma unchanged_memory_invariant_mono,
 Installing automatic rewrites from: subset_equal
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 This completes the proof of plain_memory_unchanged_read_side_effect_block.
 Q.E.D.

C.183.33

Plain_Memory.plain_memory_unchanged_write_side_effect_block_ro_rw

Terse proof for plain_memory_unchanged_write_side_effect_block_ro_rw.

plain_memory_unchanged_write_side_effect_block_ro_rw:

{1}	$\forall (\text{addr}: \text{Address}, \text{pm}: \text{Plain_Memory}, \text{size}: \text{nat}):$ $\text{plain_memory?}(\text{pm}) \wedge (\text{address_block}(\text{addr}, \text{size}) \subseteq \text{pm}'\text{rw_addr}) \supset$ $\text{unchanged_memory_invariant?}(\text{pm}'\text{mem}, \text{pm}'\text{states},$ $\text{memory_write_side_effect_super_transformers}(\text{pm}'\text{mem}, \text{ad-}$ $\text{dress_block} \quad (\text{pm}'\text{ro_addr} \cup \text{pm}'\text{rw_addr})) \quad (\text{addr},$
-----	---

Repeatedly Skolemizing and flattening,
 Using lemma plain_memory_unchanged_invariant,
 Simplifying, rewriting, and recording with decision procedures,
 Using lemma unchanged_memory_invariant_mono,
 Installing automatic rewrites from: subset_equal subset_bigger_union_right subset_bigger_union_left
 memory_write_side_effect_super_transformers_mono
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 This completes the proof of plain_memory_unchanged_write_side_effect_block_ro_rw.
 Q.E.D.

C.183.34

Plain_Memory.plain_memory_unchanged_write_side_effect_block

Terse proof for plain_memory_unchanged_write_side_effect_block.

plain_memory_unchanged_write_side_effect_block:

$\{1\} \quad \forall (\text{addr: Address, pm: Plain_Memory, size: nat}):$ $\text{plain_memory?}(pm) \wedge (\text{address_block}(\text{addr, size}) \subseteq pm'rw_addr) \supset$ $\text{unchanged_memory_invariant?}(pm' mem, pm' states,$ $\text{memory_write_side_effect_super_transformers}(pm' mem,$ $\text{dress_block} \text{ address_block}(\text{addr, size}),$ $\text{address_block}(\text{addr, size}))$	ad- $(\text{addr, size}),$
---	-------------------------------------

Repeatedly Skolemizing and flattening,
 Using lemma plain_memory_unchanged_write_side_effect_block_ro_rw,
 Simplifying, rewriting, and recording with decision procedures,
 Installing automatic rewrites from: subset_equal union_subset1 union_subset3 union_left union_right
 subset_singleton subset_bigger_union_right subset_bigger_union_left
 Using lemma unchanged_memory_invariant_mono,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 This completes the proof of plain_memory_unchanged_write_side_effect_block.
 Q.E.D.

C.183.35 Plain_Memory.plain_memory_unchanged_block

Terse proof for plain_memory_unchanged_block.

plain_memory_unchanged_block:

$\{1\} \quad \forall (\text{addr: Address, pm: Plain_Memory, size: nat}):$ $\text{plain_memory?}(pm) \wedge (\text{address_block}(\text{addr, size}) \subseteq pm'rw_addr) \supset$ $\text{unchanged_memory_write_invariant?}(pm' mem, pm' states, \text{ad-}$ $\text{dress_block}(\text{addr, size}))$

Repeatedly Skolemizing and flattening,
 Expanding the definition of plain_memory?,
 Applying disjunctive simplification to flatten sequent,
 Keeping (-4 -9 1) and hiding *,
 Rewriting using unchanged_memory_write_invariant_mono_addresses, matching in * where ad-
 dresses_1 gets address_block(addr!1, size!1), addresses_2 gets pm!1'rw_addr,
 This completes the proof of plain_memory_unchanged_block.
 Q.E.D.

C.183.36 Plain_Memory.plain_memory_changed_block

Terse proof for plain_memory_changed_block.

plain_memory_changed_block:

$\{1\} \quad \forall (\text{addr: Address, pm: Plain_Memory, size: nat}):$ $\text{plain_memory?}(pm) \wedge (\text{address_block}(\text{addr, size}) \subseteq pm'rw_addr) \supset$ $\text{changed_memory_invariant?}(pm' mem, pm' states, \text{address_block}(\text{addr, size}))$

Repeatedly Skolemizing and flattening,
 Expanding the definition of plain_memory?,
 Applying disjunctive simplification to flatten sequent,

Rewriting using `changed_memory_invariant_mono`, matching in `*` where `addresses_1` gets `address_block(addr!1, size!1)`, `addresses_2` gets `pm!1'rw_addr`,
 This completes the proof of `plain_memory_changed_block`.
 Q.E.D.

C.183.37

Plain_Memory.plain_memory_side_effect_content_unchanged_read_block

Terse proof for `plain_memory_side_effect_content_unchanged_read_block`.

`plain_memory_side_effect_content_unchanged_read_block`:

$$\{1\} \quad \forall (addr: Address, pm: Plain_Memory, size: nat):$$

$$plain_memory?(pm) \wedge (address_block(addr, size) \subseteq (pm'ro_addr \cup pm'rw_addr)) \supset$$

$$side_effect_content_unchanged(address_block(addr, size), pm'states,$$

$$memory_read_side_effect(pm'mem))$$

Expanding the definition of `plain_memory?`,
 Repeatedly Skolemizing and flattening,
 Keeping (-7 -9 1) and hiding `*`,
 Rewriting using `side_effect_content_unchanged_mono[State]`, matching in `*` where `addresses_1` gets `address_block(addr!1, size!1)`, `addresses_2` gets `union(pm!1'ro_addr, pm!1'rw_addr)`,
 This completes the proof of `plain_memory_side_effect_content_unchanged_read_block`.
 Q.E.D.

C.183.38

Plain_Memory.plain_memory_side_effect_content_unchanged_write_block

Terse proof for `plain_memory_side_effect_content_unchanged_write_block`.

`plain_memory_side_effect_content_unchanged_write_block`:

$$\{1\} \quad \forall (addr: Address, pm: Plain_Memory, size: nat):$$

$$plain_memory?(pm) \wedge (address_block(addr, size) \subseteq pm'rw_addr) \supset$$

$$side_effect_content_unchanged(address_block(addr, size), pm'states,$$

$$memory_write_side_effect(pm'mem))$$

Expanding the definition of `plain_memory?`,
 Repeatedly Skolemizing and flattening,
 Keeping (-8 -9 1) and hiding `*`,
 Rewriting using `side_effect_content_unchanged_mono[State]`, matching in `*` where `addresses_1` gets `address_block(addr!1, size!1)`, `addresses_2` gets `pm!1'rw_addr`,
 This completes the proof of `plain_memory_side_effect_content_unchanged_write_block`.
 Q.E.D.

C.183.39 Plain_Memory.plain_memory_memory_read_ok

Terse proof for `plain_memory_memory_read_ok`.

`plain_memory_memory_read_ok`:

$$\{1\} \quad \forall (addr: Address, pm: Plain_Memory, s: State):$$

$$plain_memory?(pm) \wedge pm'states(s) \wedge union(pm'ro_addr, pm'rw_addr)(addr) \supset$$

$$OK?(memory_read(pm'mem)(addr)(s))$$

Repeatedly Skolemizing and flattening,
 Using lemma plain_memory_transformers_ok_read_ro_rw,
 Simplifying, rewriting, and recording with decision procedures,
 Using lemma expr_transformers_ok_ok,
 Installing automatic rewrites from: memory_read_transformers_memory_read
 Simplifying, rewriting, and recording with decision procedures,
 This completes the proof of plain_memory_memory_read_ok.
 Q.E.D.

C.183.40 Plain_Memory.plain_memory_states_memory_read_TCC1

Terse proof for plain_memory_states_memory_read_TCC1.

plain_memory_states_memory_read_TCC1:

$$\{1\} \quad \forall (\text{addr}: \text{Address}, \text{pm}: \text{Plain_Memory}, s: \text{State}):$$

$$\text{plain_memory?}(\text{pm}) \wedge \text{pm}'\text{states}(s) \wedge \text{union}(\text{pm}'\text{ro_addr}, \text{pm}'\text{rw_addr})(\text{addr}) \supset$$

$$\text{OK?}[\text{State}, \text{Byte}](\text{memory_read}(\text{pm}'\text{mem})(\text{addr})(s)) \vee$$

$$\text{Exception?}[\text{State}, \text{Byte}](\text{memory_read}(\text{pm}'\text{mem})(\text{addr})(s))$$

Repeatedly Skolemizing and flattening,
 Rewriting using plain_memory_memory_read_ok, matching in *,
 This completes the proof of plain_memory_states_memory_read_TCC1.
 Q.E.D.

C.183.41 Plain_Memory.plain_memory_states_memory_read

Terse proof for plain_memory_states_memory_read.

plain_memory_states_memory_read:

$$\{1\} \quad \forall (\text{addr}: \text{Address}, \text{pm}: \text{Plain_Memory}, s: \text{State}):$$

$$\text{plain_memory?}(\text{pm}) \wedge \text{pm}'\text{states}(s) \wedge \text{union}(\text{pm}'\text{ro_addr}, \text{pm}'\text{rw_addr})(\text{addr}) \supset$$

$$\text{pm}'\text{states}(\text{state}(\text{memory_read}(\text{pm}'\text{mem})(\text{addr})(s)))$$

Repeatedly Skolemizing and flattening,
 Using lemma plain_memory_invariant_read_transformers_ro_rw_addr,
 Simplifying, rewriting, and recording with decision procedures,
 Using lemma expr_transformer_invariant_next_ok[State, Byte],
 Installing automatic rewrites from: memory_read_transformers_memory_read plain_memory_memory_read_ok
 has_next_state!
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 This completes the proof of plain_memory_states_memory_read.
 Q.E.D.

C.183.42

Plain_Memory.plain_memory_memory_read_memory_read_TCC1

Terse proof for plain_memory_memory_read_memory_read_TCC1.

plain_memory_memory_read_memory_read_TCC1:

```
{1}  ∃ (pm: Plain_Memory, s: State, addr1, addr2: Address):
      plain_memory?(pm) ∧
      pm'states(s) ∧
      union(pm'ro_addr, pm'rw_addr)(addr1) ∧ union(pm'ro_addr, pm'rw_addr)(addr2)
      ⊃
      OK?[State, Byte](memory_read(pm'mem)(addr2)(s)) ∨
      Exception?[State, Byte](memory_read(pm'mem)(addr2)(s))
```

Repeatedly Skolemizing and flattening,
 Rewriting using plain_memory_memory_read_ok, matching in *,
 This completes the proof of plain_memory_memory_read_memory_read_TCC1.
 Q.E.D.

C.183.43

Plain_Memory.plain_memory_memory_read_memory_read_TCC2

Terse proof for plain_memory_memory_read_memory_read_TCC2.

plain_memory_memory_read_memory_read_TCC2:

```
{1}  ∃ (pm: Plain_Memory, s: State, addr1, addr2: Address):
      plain_memory?(pm) ∧
      pm'states(s) ∧
      union(pm'ro_addr, pm'rw_addr)(addr1) ∧ union(pm'ro_addr, pm'rw_addr)(addr2)
      ⊃
      OK?[State, Byte]
      (memory_read(pm'mem)
       (addr1)(state[State, Byte](memory_read(pm'mem)(addr2)(s))))
```

Repeatedly Skolemizing and flattening,
 Rewriting using plain_memory_memory_read_ok, matching in *,
 Rewriting using plain_memory_states_memory_read, matching in *,
 This completes the proof of plain_memory_memory_read_memory_read_TCC2.
 Q.E.D.

C.183.44

Plain_Memory.plain_memory_memory_read_memory_read_TCC3

Terse proof for plain_memory_memory_read_memory_read_TCC3.

plain_memory_memory_read_memory_read_TCC3:

```
{1}  ∃ (pm: Plain_Memory, s: State, addr1, addr2: Address):
      plain_memory?(pm) ∧
      pm'states(s) ∧
      union(pm'ro_addr, pm'rw_addr)(addr1) ∧ union(pm'ro_addr, pm'rw_addr)(addr2)
      ⊃ OK?[State, Byte](memory_read(pm'mem)(addr1)(s))
```

Repeatedly Skolemizing and flattening,
 Rewriting using plain_memory_memory_read_ok, matching in *,
 This completes the proof of plain_memory_memory_read_memory_read_TCC3.

Q.E.D.

C.183.45 Plain_Memory.plain_memory_memory_read_memory_read

Terse proof for plain_memory_memory_read_memory_read.

plain_memory_memory_read_memory_read:

$\{1\} \quad \forall (pm: \text{Plain_Memory}, s: \text{State}, addr1, addr2: \text{Address}):$ $\text{plain_memory?}(pm) \wedge$ $pm' \text{states}(s) \wedge$ $\text{union}(pm' \text{ro_addr}, pm' \text{rw_addr})(addr1) \wedge \text{union}(pm' \text{ro_addr}, pm' \text{rw_addr})(addr2)$ \supset $\text{data}(\text{memory_read}(pm' \text{mem})(addr1)(\text{state}(\text{memory_read}(pm' \text{mem})(addr2)(s)))) =$ $\text{data}(\text{memory_read}(pm' \text{mem})(addr1)(s))$
--

Repeatedly Skolemizing and flattening,

Using lemma plain_memory_unchanged_read_ro_rw_addr,

Simplifying, rewriting, and recording with decision procedures,

Using lemma expr_unchanged_memory_invariant_unchanged[State, Byte],

Installing automatic rewrites from: plain_memory_memory_read_ok plain_memory_states_memory_read
memory_read_transformers_memory_read

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of plain_memory_memory_read_memory_read.

Q.E.D.

C.183.46 Plain_Memory.plain_memory_memory_write_ok

Terse proof for plain_memory_memory_write_ok.

plain_memory_memory_write_ok:

$\{1\} \quad \forall (addr: \text{Address}, pm: \text{Plain_Memory}, s: \text{State}, byte: \text{Byte}):$ $\text{plain_memory?}(pm) \wedge pm' \text{states}(s) \wedge pm' \text{rw_addr}(addr) \supset$ $\text{OK?}(\text{memory_write}(pm' \text{mem})(addr, byte)(s))$
--

Repeatedly Skolemizing and flattening,

Using lemma plain_memory_transformers_ok_write_rw,

Simplifying, rewriting, and recording with decision procedures,

Using lemma expr_transformers_ok_ok,

Installing automatic rewrites from: memory_write_transformers_memory_write

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of plain_memory_memory_write_ok.

Q.E.D.

C.183.47 Plain_Memory.plain_memory_states_memory_write_TCC1

Terse proof for plain_memory_states_memory_write_TCC1.

plain_memory_states_memory_write_TCC1:

$$\{1\} \quad \forall (\text{addr}: \text{Address}, \text{pm}: \text{Plain_Memory}, \text{s}: \text{State}, \text{byte}: \text{Byte}):$$

$$\text{plain_memory?}(\text{pm}) \wedge \text{pm}'\text{states}(\text{s}) \wedge \text{pm}'\text{rw_addr}(\text{addr}) \supset$$

$$\text{OK?}[\text{State}, \text{Unit}](\text{memory_write}(\text{pm}'\text{mem})(\text{addr}, \text{byte})(\text{s})) \vee$$

$$\text{Exception?}[\text{State}, \text{Unit}](\text{memory_write}(\text{pm}'\text{mem})(\text{addr}, \text{byte})(\text{s}))$$

Repeatedly Skolemizing and flattening,

Rewriting using plain_memory_memory_write_ok, matching in *,

This completes the proof of plain_memory_states_memory_write_TCC1.

Q.E.D.

C.183.48 Plain_Memory.plain_memory_states_memory_write

Terse proof for plain_memory_states_memory_write.

plain_memory_states_memory_write:

$$\{1\} \quad \forall (\text{addr}: \text{Address}, \text{pm}: \text{Plain_Memory}, \text{s}: \text{State}, \text{byte}: \text{Byte}):$$

$$\text{plain_memory?}(\text{pm}) \wedge \text{pm}'\text{states}(\text{s}) \wedge \text{pm}'\text{rw_addr}(\text{addr}) \supset$$

$$\text{pm}'\text{states}(\text{state}(\text{memory_write}(\text{pm}'\text{mem})(\text{addr}, \text{byte})(\text{s})))$$

Repeatedly Skolemizing and flattening,

Using lemma plain_memory_invariant_write_transformers_rw_addr,

Using lemma expr_transformer_invariant_next_ok[State, Unit],

Installing automatic rewrites from: memory_write_transformers_memory_write_plain_memory_memory_write_ok
has_next_state!

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of plain_memory_states_memory_write.

Q.E.D.

C.183.49

Plain_Memory.plain_memory_memory_read_memory_write_other_TCC1

Terse proof for plain_memory_memory_read_memory_write_other_TCC1.

plain_memory_memory_read_memory_write_other_TCC1:

$$\{1\} \quad \forall (\text{pm}: \text{Plain_Memory}, \text{s}: \text{State}, \text{ro_addr}, \text{rw_addr}: \text{Address}, \text{byte}: \text{Byte}):$$

$$\text{plain_memory?}(\text{pm}) \wedge$$

$$\text{pm}'\text{states}(\text{s}) \wedge$$

$$\text{union}(\text{pm}'\text{ro_addr}, \text{pm}'\text{rw_addr})(\text{ro_addr}) \wedge \text{pm}'\text{rw_addr}(\text{rw_addr}) \wedge \neg \text{ro_addr} = \text{rw_addr}$$

$$\supset$$

$$\text{OK?}[\text{State}, \text{Unit}](\text{memory_write}(\text{pm}'\text{mem})(\text{rw_addr}, \text{byte})(\text{s})) \vee$$

$$\text{Exception?}[\text{State}, \text{Unit}](\text{memory_write}(\text{pm}'\text{mem})(\text{rw_addr}, \text{byte})(\text{s}))$$

Repeatedly Skolemizing and flattening,

Rewriting using plain_memory_memory_write_ok, matching in *,

This completes the proof of plain_memory_memory_read_memory_write_other_TCC1.

Q.E.D.

C.183.50**Plain_Memory.plain_memory_memory_read_memory_write_other_TCC2**

Terse proof for plain_memory_memory_read_memory_write_other_TCC2.

plain_memory_memory_read_memory_write_other_TCC2:

$\{1\} \quad \forall (pm: \text{Plain_Memory}, s: \text{State}, ro_addr, rw_addr: \text{Address}, byte: \text{Byte}):$ $\text{plain_memory?}(pm) \wedge$ $pm' \text{states}(s) \wedge$ $\text{union}(pm'ro_addr, pm'rw_addr)(ro_addr) \wedge pm'rw_addr(rw_addr) \wedge \neg ro_addr = rw_addr$ \supset $\text{OK?}[\text{State}, \text{Byte}]$ $(\text{memory_read}(pm' \text{mem})$ $(ro_addr)(\text{state}[\text{State}, \text{Unit}](\text{memory_write}(pm' \text{mem})(rw_addr, byte)(s))))$

Repeatedly Skolemizing and flattening,

Rewriting using plain_memory_memory_read_ok, matching in *,

Rewriting using plain_memory_states_memory_write, matching in *,

This completes the proof of plain_memory_memory_read_memory_write_other_TCC2.

Q.E.D.

C.183.51**Plain_Memory.plain_memory_memory_read_memory_write_other_TCC3**

Terse proof for plain_memory_memory_read_memory_write_other_TCC3.

plain_memory_memory_read_memory_write_other_TCC3:

$\{1\} \quad \forall (pm: \text{Plain_Memory}, s: \text{State}, ro_addr, rw_addr: \text{Address}):$ $\text{plain_memory?}(pm) \wedge$ $pm' \text{states}(s) \wedge$ $\text{union}(pm'ro_addr, pm'rw_addr)(ro_addr) \wedge pm'rw_addr(rw_addr) \wedge \neg ro_addr = rw_addr$ $\supset \text{OK?}[\text{State}, \text{Byte}](\text{memory_read}(pm' \text{mem})(ro_addr)(s))$

Repeatedly Skolemizing and flattening,

Rewriting using plain_memory_memory_read_ok, matching in *,

This completes the proof of plain_memory_memory_read_memory_write_other_TCC3.

Q.E.D.

C.183.52**Plain_Memory.plain_memory_memory_read_memory_write_other**

Terse proof for plain_memory_memory_read_memory_write_other.

plain_memory_memory_read_memory_write_other:

<pre>{1} ∃ (pm: Plain_Memory, s: State, ro_addr, rw_addr: Address, byte: Byte): plain_memory?(pm) ∧ pm' states(s) ∧ union(pm' ro_addr, pm' rw_addr)(ro_addr) ∧ pm' rw_addr(rw_addr) ∧ ¬ ro_addr = rw_addr ⊃ data(memory_read(pm' mem) (ro_addr)(state(memory_write(pm' mem)(rw_addr, byte)(s)))) = data(memory_read(pm' mem)(ro_addr)(s))</pre>
--

Repeatedly Skolemizing and flattening,

Installing automatic rewrites from: union member

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 2 subgoals:

plain_memory_memory_read_memory_write_other.1:

<pre>{-1} byte' < max_byte {-2} plain_memory?(pm') {-3} pm' states(s') {-4} pm' ro_addr(ro_addr') {-5} pm' rw_addr(rw_addr')</pre>
<pre>{1} ro_addr' = rw_addr' {2} data(memory_read(pm' mem) (ro_addr')(state(memory_write(pm' mem)(rw_addr', byte')(s')))) = data(memory_read(pm' mem)(ro_addr')(s'))</pre>

Using lemma plain_memory_unchanged_memory_invariant_write,

Simplifying, rewriting, and recording with decision procedures,

Using lemma expr_unchanged_memory_invariant_unchanged [State, Unit],

Installing automatic rewrites from: plain_memory_memory_read_ok plain_memory_states_memory_write
plain_memory_memory_write_ok memory_write_transformers_memory_write

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of plain_memory_memory_read_memory_write_other.1.

plain_memory_memory_read_memory_write_other.2:

<pre>{-1} byte' < max_byte {-2} plain_memory?(pm') {-3} pm' states(s') {-4} pm' rw_addr(ro_addr') {-5} pm' rw_addr(rw_addr')</pre>
<pre>{1} ro_addr' = rw_addr' {2} data(memory_read(pm' mem) (ro_addr')(state(memory_write(pm' mem)(rw_addr', byte')(s')))) = data(memory_read(pm' mem)(ro_addr')(s'))</pre>

Using lemma plain_memory_unchanged_memory_write_invariant,

Simplifying, rewriting, and recording with decision procedures,

Expanding the definition of unchanged_memory_write_invariant?,

Instantiating the top quantifier in -1 with the terms: rw_addr', byte',

Simplifying, rewriting, and recording with decision procedures,

Using lemma expr_unchanged_memory_invariant_unchanged [State, Unit],

Installing automatic rewrites from: plain_memory_memory_read_ok plain_memory_states_memory_write
plain_memory_memory_write_ok memory_write_transformers_memory_write singleton

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 Keeping (-5 1) and hiding *,
 Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of plain_memory_memory_read_memory_write_other.2.
 Q.E.D.

C.183.53 Plain_Memory.stays_unchanged_TCC1

Terse proof for stays_unchanged_TCC1.

stays_unchanged_TCC1:

$\{1\} \quad \forall (pm: (\text{plain_memory?}), s_1, s_2: (\text{pm}'\text{states}), \text{addresses}: \text{PRED}[\text{Address}], \text{addr}: \text{Address}):$ $\text{union}(pm'\text{ro_addr}, pm'\text{rw_addr})(\text{addr}) \wedge \text{addresses}(\text{addr}) \supset$ $\text{OK?}[\text{State}, \text{Byte}](\text{memory_read}(pm'\text{mem})(\text{addr})(s_1))$
--

Repeatedly Skolemizing and flattening,
 Rewriting using plain_memory_memory_read_ok, matching in *,
 This completes the proof of stays_unchanged_TCC1.
 Q.E.D.

C.183.54 Plain_Memory.stays_unchanged_TCC2

Terse proof for stays_unchanged_TCC2.

stays_unchanged_TCC2:

$\{1\} \quad \forall (pm: (\text{plain_memory?}), s_1, s_2: (\text{pm}'\text{states}), \text{addresses}: \text{PRED}[\text{Address}], \text{addr}: \text{Address}):$ $\text{union}(pm'\text{ro_addr}, pm'\text{rw_addr})(\text{addr}) \wedge \text{addresses}(\text{addr}) \supset$ $\text{OK?}[\text{State}, \text{Byte}](\text{memory_read}(pm'\text{mem})(\text{addr})(s_2))$
--

Repeatedly Skolemizing and flattening,
 Rewriting using plain_memory_memory_read_ok, matching in *,
 This completes the proof of stays_unchanged_TCC2.
 Q.E.D.

C.183.55 Plain_Memory.stays_unchanged_unchanged_TCC1

Terse proof for stays_unchanged_unchanged_TCC1.

stays_unchanged_unchanged_TCC1:

$\{1\} \quad \forall (pm: \text{Plain_Memory}, s_1, s_2: \text{State}, \text{addresses}: \text{PRED}[\text{Address}], \text{addr}: \text{Address}):$ $\text{plain_memory?}(pm) \wedge$ $pm'\text{states}(s_1) \wedge$ $pm'\text{states}(s_2) \wedge$ $\text{union}(pm'\text{ro_addr}, pm'\text{rw_addr})(\text{addr}) \wedge$ $\text{stays_unchanged}(pm)(s_1, s_2, \text{addresses}) \wedge \text{addresses}(\text{addr})$ $\supset \text{OK?}[\text{State}, \text{Byte}](\text{memory_read}(pm'\text{mem})(\text{addr})(s_1))$
--

Repeatedly Skolemizing and flattening,
 Rewriting using plain_memory_memory_read_ok, matching in *,

This completes the proof of `stays_unchanged_unchanged_TCC1`.
Q.E.D.

C.183.56 Plain_Memory.stays_unchanged_unchanged_TCC2

Terse proof for `stays_unchanged_unchanged_TCC2`.

`stays_unchanged_unchanged_TCC2`:

$$\{1\} \quad \forall (pm: \text{Plain_Memory}, s_1, s_2: \text{State}, \text{addresses}: \text{PRED}[\text{Address}], \text{addr}: \text{Address}):$$

$$\text{plain_memory?}(pm) \wedge$$

$$pm' \text{states}(s_1) \wedge$$

$$pm' \text{states}(s_2) \wedge$$

$$\text{union}(pm' \text{ro_addr}, pm' \text{rw_addr})(\text{addr}) \wedge$$

$$\text{stays_unchanged}(pm)(s_1, s_2, \text{addresses}) \wedge \text{addresses}(\text{addr})$$

$$\supset \text{OK?}[\text{State}, \text{Byte}](\text{memory_read}(pm' \text{mem})(\text{addr})(s_2))$$

Repeatedly Skolemizing and flattening,
Rewriting using `plain_memory_memory_read_ok`, matching in *,
This completes the proof of `stays_unchanged_unchanged_TCC2`.
Q.E.D.

C.183.57 Plain_Memory.stays_unchanged_unchanged

Terse proof for `stays_unchanged_unchanged`.

`stays_unchanged_unchanged`:

$$\{1\} \quad \forall (pm: \text{Plain_Memory}, s_1, s_2: \text{State}, \text{addresses}: \text{PRED}[\text{Address}], \text{addr}: \text{Address}):$$

$$\text{plain_memory?}(pm) \wedge$$

$$pm' \text{states}(s_1) \wedge$$

$$pm' \text{states}(s_2) \wedge$$

$$\text{union}(pm' \text{ro_addr}, pm' \text{rw_addr})(\text{addr}) \wedge$$

$$\text{stays_unchanged}(pm)(s_1, s_2, \text{addresses}) \wedge \text{addresses}(\text{addr})$$

$$\supset \text{data}(\text{memory_read}(pm' \text{mem})(\text{addr})(s_1)) = \text{data}(\text{memory_read}(pm' \text{mem})(\text{addr})(s_2))$$

Repeatedly Skolemizing and flattening,
Expanding the definition of `stays_unchanged`,
Instantiating quantified variables,
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `stays_unchanged_unchanged`.
Q.E.D.

C.183.58 Plain_Memory.stays_unchanged_symmetric

Terse proof for `stays_unchanged_symmetric`.

`stays_unchanged_symmetric`:

$$\{1\} \quad \forall (pm: \text{Plain_Memory}, s_1, s_2: \text{State}, \text{addresses}: \text{PRED}[\text{Address}]):$$

$$\text{plain_memory?}(pm) \wedge pm' \text{states}(s_1) \wedge pm' \text{states}(s_2) \supset$$

$$\text{stays_unchanged}(pm)(s_1, s_2, \text{addresses}) = \text{stays_unchanged}(pm)(s_2, s_1, \text{addresses})$$

Repeatedly Skolemizing and flattening,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 2 subgoals:

`stays_unchanged_symmetric.1:`

{-1}	plain_memory?(pm')
{-2}	pm' states(s' ₁)
{-3}	pm' states(s' ₂)
{-4}	stays_unchanged(pm')(s' ₁ , s' ₂ , addresses')
{1}	
	stays_unchanged(pm')(s' ₂ , s' ₁ , addresses')

Expanding the definition of `stays_unchanged`,
 Repeatedly Skolemizing and flattening,
 Instantiating quantified variables,
 Simplifying, rewriting, and recording with decision procedures,
 This completes the proof of `stays_unchanged_symmetric.1`.

`stays_unchanged_symmetric.2:`

{-1}	plain_memory?(pm')
{-2}	pm' states(s' ₁)
{-3}	pm' states(s' ₂)
{-4}	stays_unchanged(pm')(s' ₂ , s' ₁ , addresses')
{1}	
	stays_unchanged(pm')(s' ₁ , s' ₂ , addresses')

Expanding the definition of `stays_unchanged`,
 Repeatedly Skolemizing and flattening,
 Instantiating quantified variables,
 Simplifying, rewriting, and recording with decision procedures,
 This completes the proof of `stays_unchanged_symmetric.2`.
 Q.E.D.

C.183.59 Plain_Memory.stays_unchanged_mono

Terse proof for `stays_unchanged_mono`.

`stays_unchanged_mono:`

{1}	\forall (pm: Plain_Memory, s ₁ , s ₂ : State, addresses_1, addresses_2: PRED[Address]): plain_memory?(pm) \wedge pm states(s ₁) \wedge pm states(s ₂) \wedge (addresses_1 \subseteq addresses_2) \wedge stays_unchanged(pm)(s ₁ , s ₂ , addresses_2) \supset stays_unchanged(pm)(s ₁ , s ₂ , addresses_1)
-----	--

Repeatedly Skolemizing and flattening,
 Expanding the definition of `stays_unchanged`,
 Repeatedly Skolemizing and flattening,
 Instantiating quantified variables,
 Simplifying, rewriting, and recording with decision procedures,
 Hiding formulas: -1, 2,
 Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of `stays_unchanged_mono`.
 Q.E.D.

C.183.60 Plain_Memory.stays_unchanged_only_changes

Terse proof for `stays_unchanged_only_changes`.

C Proof scripts

stays_unchanged_only_changes:

{1} \forall (pm: Plain_Memory, s_1, s_2 : State, addresses: PRED[Address]): plain_memory?(pm) \wedge pm' states(s_1) \wedge pm' states(s_2) \wedge (pm'ro_addr \subseteq addresses) \supset stays_unchanged(pm)($s_1, s_2, \text{addresses}$) = only_changes(pm)($s_1, s_2, (\text{pm}'\text{rw_addr} \setminus \text{addresses})$)

Repeatedly Skolemizing and flattening,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 2 subgoals:

stays_unchanged_only_changes.1:

{-1} plain_memory?(pm')
{-2} pm' states(s'_1)
{-3} pm' states(s'_2)
{-4} (pm'ro_addr \subseteq addresses')
{-5} stays_unchanged(pm')($s'_1, s'_2, \text{addresses}'$)
{1} only_changes(pm')($s'_1, s'_2, (\text{pm}'\text{rw_addr} \setminus \text{addresses}')$)

Expanding the definition of only_changes,

Expanding the definition of stays_unchanged,

Repeatedly Skolemizing and flattening,

Instantiating quantified variables,

Simplifying, rewriting, and recording with decision procedures,

Keeping (-4 -5 -6 1) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of stays_unchanged_only_changes.1.

stays_unchanged_only_changes.2:

{-1} plain_memory?(pm')
{-2} pm' states(s'_1)
{-3} pm' states(s'_2)
{-4} (pm'ro_addr \subseteq addresses')
{-5} only_changes(pm')($s'_1, s'_2, (\text{pm}'\text{rw_addr} \setminus \text{addresses}')$)
{1} stays_unchanged(pm')($s'_1, s'_2, \text{addresses}'$)

Expanding the definition of only_changes,

Expanding the definition of stays_unchanged,

Repeatedly Skolemizing and flattening,

Instantiating quantified variables,

Simplifying, rewriting, and recording with decision procedures,

Keeping (-4 -5 -6 2) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of stays_unchanged_only_changes.2.

Q.E.D.

C.183.61 Plain_Memory.stays_unchanged_only_changes_disjoint

Terse proof for stays_unchanged_only_changes_disjoint.

stays_unchanged_only_changes_disjoint:

$\{1\} \quad \forall (pm: Plain_Memory, s_1, s_2: State, addresses_1, addresses_2: PRED[Address]):$ $plain_memory?(pm) \wedge$ $pm'states(s_1) \wedge$ $pm'states(s_2) \wedge$ $(addresses_2 \subseteq (pm'ro_addr \cup pm'rw_addr)) \wedge$ $disjoint?(addresses_1, addresses_2) \wedge only_changes(pm)(s_1, s_2, addresses_1)$ $\supset stays_unchanged(pm)(s_1, s_2, addresses_2)$

Repeatedly Skolemizing and flattening,
 Expanding the definition of only_changes,
 Using lemma stays_unchanged_mono,
 Simplifying, rewriting, and recording with decision procedures,
 Keeping (-4 -5 1) and hiding *,
 Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of stays_unchanged_only_changes_disjoint.
 Q.E.D.

C.183.62 Plain_Memory.only_changes_symmetric

Terse proof for only_changes_symmetric.

only_changes_symmetric:

$\{1\} \quad \forall (pm: Plain_Memory, s_1, s_2: State, addresses: PRED[Address]):$ $plain_memory?(pm) \wedge pm'states(s_1) \wedge pm'states(s_2) \supset$ $only_changes(pm)(s_1, s_2, addresses) = only_changes(pm)(s_2, s_1, addresses)$
--

Repeatedly Skolemizing and flattening,
 Expanding the definition of only_changes,
 Rewriting using stays_unchanged_symmetric, matching in *,
 This completes the proof of only_changes_symmetric.
 Q.E.D.

C.183.63 Plain_Memory.only_changes_mono

Terse proof for only_changes_mono.

only_changes_mono:

$\{1\} \quad \forall (pm: Plain_Memory, s_1, s_2: State, addresses_1, addresses_2: PRED[Address]):$ $plain_memory?(pm) \wedge$ $pm'states(s_1) \wedge$ $pm'states(s_2) \wedge$ $(addresses_1 \subseteq addresses_2) \wedge only_changes(pm)(s_1, s_2, addresses_1)$ $\supset only_changes(pm)(s_1, s_2, addresses_2)$

Repeatedly Skolemizing and flattening,
 Expanding the definition of only_changes,
 Using lemma stays_unchanged_mono,
 Simplifying, rewriting, and recording with decision procedures,
 Keeping (-4 1) and hiding *,
 Trying repeated skolemization, instantiation, and if-lifting,

C Proof scripts

This completes the proof of `only_changes_mono`.

Q.E.D.

C.183.64 Plain_Memory.only_changes_unchanged_TCC1

Terse proof for `only_changes_unchanged_TCC1`.

`only_changes_unchanged_TCC1`:

$$\begin{array}{l} \{1\} \quad \forall (\text{addr: Address, pm: Plain_Memory, } s_1, s_2: \text{State, addresses: PRED[Address]):} \\ \quad \text{plain_memory?}(pm) \wedge \\ \quad \text{pm}'\text{states}(s_1) \wedge \\ \quad \text{pm}'\text{states}(s_2) \wedge \\ \quad \text{only_changes}(pm)(s_1, s_2, \text{addresses}) \wedge \\ \quad (\text{pm}'\text{ro_addr}(\text{addr}) \vee \text{pm}'\text{rw_addr}(\text{addr}) \wedge \neg \text{addresses}(\text{addr})) \\ \quad \supset \text{OK?}[State, Byte](\text{memory_read}(pm'\text{mem})(\text{addr})(s_1)) \end{array}$$

Repeatedly Skolemizing and flattening,

Rewriting using `plain_memory_memory_read_ok`, matching in `*`,

Keeping `(-5 1)` and hiding `*`,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `only_changes_unchanged_TCC1`.

Q.E.D.

C.183.65 Plain_Memory.only_changes_unchanged_TCC2

Terse proof for `only_changes_unchanged_TCC2`.

`only_changes_unchanged_TCC2`:

$$\begin{array}{l} \{1\} \quad \forall (\text{addr: Address, pm: Plain_Memory, } s_1, s_2: \text{State, addresses: PRED[Address]):} \\ \quad \text{plain_memory?}(pm) \wedge \\ \quad \text{pm}'\text{states}(s_1) \wedge \\ \quad \text{pm}'\text{states}(s_2) \wedge \\ \quad \text{only_changes}(pm)(s_1, s_2, \text{addresses}) \wedge \\ \quad (\text{pm}'\text{ro_addr}(\text{addr}) \vee \text{pm}'\text{rw_addr}(\text{addr}) \wedge \neg \text{addresses}(\text{addr})) \\ \quad \supset \text{OK?}[State, Byte](\text{memory_read}(pm'\text{mem})(\text{addr})(s_2)) \end{array}$$

Repeatedly Skolemizing and flattening,

Rewriting using `plain_memory_memory_read_ok`, matching in `*`,

Keeping `(-5 1)` and hiding `*`,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `only_changes_unchanged_TCC2`.

Q.E.D.

C.183.66 Plain_Memory.only_changes_unchanged

Terse proof for `only_changes_unchanged`.

only_changes_unchanged:

$\{1\} \quad \forall (\text{addr}: \text{Address}, \text{pm}: \text{Plain_Memory}, s_1, s_2: \text{State}, \text{addresses}: \text{PRED}[\text{Address}]):$ $\text{plain_memory?}(\text{pm}) \wedge$ $\text{pm}'\text{states}(s_1) \wedge$ $\text{pm}'\text{states}(s_2) \wedge$ $\text{only_changes}(\text{pm})(s_1, s_2, \text{addresses}) \wedge$ $(\text{pm}'\text{ro_addr}(\text{addr}) \vee \text{pm}'\text{rw_addr}(\text{addr}) \wedge \neg \text{addresses}(\text{addr}))$ $\supset \text{data}(\text{memory_read}(\text{pm}'\text{mem})(\text{addr})(s_1)) = \text{data}(\text{memory_read}(\text{pm}'\text{mem})(\text{addr})(s_2))$
--

Repeatedly Skolemizing and flattening,
 Expanding the definition of only_changes,
 Using lemma stays_unchanged_unchanged,
 Simplifying, rewriting, and recording with decision procedures,
 Keeping (-5 1) and hiding *,
 Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of only_changes_unchanged.
 Q.E.D.

C.183.67

Plain_Memory.plain_memory_stays_unchanged_memory_read_TCC1

Terse proof for plain_memory_stays_unchanged_memory_read_TCC1.

plain_memory_stays_unchanged_memory_read_TCC1:

$\{1\} \quad \forall (\text{pm}: \text{Plain_Memory}, s: \text{State}, \text{addr}: \text{Address}):$ $\text{plain_memory?}(\text{pm}) \wedge \text{pm}'\text{states}(s) \wedge \text{union}(\text{pm}'\text{ro_addr}, \text{pm}'\text{rw_addr})(\text{addr}) \supset$ $\text{pm}'\text{states}(\text{state}[\text{State}, \text{Byte}](\text{memory_read}(\text{pm}'\text{mem})(\text{addr})(s)))$
--

Repeatedly Skolemizing and flattening,
 Rewriting using plain_memory_states_memory_read, matching in *,
 This completes the proof of plain_memory_stays_unchanged_memory_read_TCC1.
 Q.E.D.

C.183.68

Plain_Memory.plain_memory_stays_unchanged_memory_read

Terse proof for plain_memory_stays_unchanged_memory_read.

plain_memory_stays_unchanged_memory_read:

$\{1\} \quad \forall (\text{pm}: \text{Plain_Memory}, s: \text{State}, \text{addr}: \text{Address}):$ $\text{plain_memory?}(\text{pm}) \wedge \text{pm}'\text{states}(s) \wedge \text{union}(\text{pm}'\text{ro_addr}, \text{pm}'\text{rw_addr})(\text{addr}) \supset$ $\text{stays_unchanged}(\text{pm})$ $(s, \text{state}(\text{memory_read}(\text{pm}'\text{mem})(\text{addr})(s)),$ $(\text{pm}'\text{ro_addr} \cup \text{pm}'\text{rw_addr}))$
--

Repeatedly Skolemizing and flattening,
 Expanding the definition of stays_unchanged,
 Repeatedly Skolemizing and flattening,
 Rewriting using plain_memory_memory_read_memory_read, matching in *,
 This completes the proof of plain_memory_stays_unchanged_memory_read.

Q.E.D.

C.183.69

Plain_Memory.plain_memory_stays_unchanged_memory_write_TCC1

Terse proof for plain_memory_stays_unchanged_memory_write_TCC1.

plain_memory_stays_unchanged_memory_write_TCC1:

$$\{1\} \quad \forall (pm: \text{Plain_Memory}, s: \text{State}, addr: \text{Address}, byte: \text{Byte}):$$
$$\text{plain_memory?}(pm) \wedge pm' \text{states}(s) \wedge pm' \text{rw_addr}(addr) \supset$$
$$pm' \text{states}(\text{state}[\text{State}, \text{Unit}](\text{memory_write}(pm' \text{mem})(addr, byte)(s)))$$

Repeatedly Skolemizing and flattening,

Rewriting using plain_memory_states_memory_write, matching in *,

This completes the proof of plain_memory_stays_unchanged_memory_write_TCC1.

Q.E.D.

C.183.70

Plain_Memory.plain_memory_stays_unchanged_memory_write

Terse proof for plain_memory_stays_unchanged_memory_write.

plain_memory_stays_unchanged_memory_write:

$$\{1\} \quad \forall (pm: \text{Plain_Memory}, s: \text{State}, addr: \text{Address}, byte: \text{Byte}):$$
$$\text{plain_memory?}(pm) \wedge pm' \text{states}(s) \wedge pm' \text{rw_addr}(addr) \supset$$
$$\text{stays_unchanged}(pm)$$
$$(s, \text{state}(\text{memory_write}(pm' \text{mem})(addr, byte)(s)),$$
$$((pm' \text{ro_addr} \cup pm' \text{rw_addr}) \setminus \{addr\}))$$

Repeatedly Skolemizing and flattening,

Expanding the definition of stays_unchanged,

Repeatedly Skolemizing and flattening,

Rewriting using plain_memory_memory_read_memory_write_other, matching in *,

Expanding the definition of remove,

which is trivially true.

This completes the proof of plain_memory_stays_unchanged_memory_write.

Q.E.D.

C.184 Proofs for PointerToMemberExpressions (expressions.pvs)

C.184.1 PointerToMemberExpressions.ptm_member_TCC1

Terse proof for ptm_member_TCC1.

ptm_member_TCC1:

$$\{1\} \quad \forall (typ: \text{Cpp_Subtype}(\text{cv}(\text{pointer_to_member?}))) : \text{pointer_to_member?}(\text{cv_base}(typ))$$

Repeatedly Skolemizing and flattening,

Rewriting using `cv_base_result`, matching in `*`,
 Keeping (1) and hiding `*`,
 Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of `ptm_member_TCC1`.
 Q.E.D.

C.184.2 PointerToMemberExpressions.ptm_arrow_TCC1

Terse proof for `ptm_arrow_TCC1`.

`ptm_arrow_TCC1`:

$\{1\} \quad \forall ((\text{ptm_typ} : \text{Cpp_Subtype}(\text{cv}(\text{pointer_to_member?}))),$ $\quad \text{ofs} : (\text{range_ptm}(\text{cv_base}(\text{ptm_typ})))) :$ $\quad \text{Cpp_Type?}(\text{pointer}(\text{typ}(\text{cv_base}(\text{ptm_typ})))) \wedge$ $\quad \text{cv}(\text{extend}[\text{Cpp_Type_}, \text{Cpp_Type}, \text{bool}, \text{FALSE}]$ $\quad \quad (\lambda (t : \text{Cpp_Type}) :$ $\quad \quad \quad \text{pointer?}(t) \wedge (\text{class?}(\text{typ}(t)) \vee \text{union?}(\text{typ}(t))))$ $\quad (\text{pointer}(\text{typ}(\text{cv_base}(\text{ptm_typ}))))$
--

Repeatedly Skolemizing and flattening,

Using lemma `cv_base_result`,

we get 2 subgoals:

`ptm_arrow_TCC1.1`:

$\{-1\} \quad \text{pointer_to_member?}(\text{cv_base}(\text{ptm_typ}'))$ $\{-2\} \quad \text{Cpp_Type?}(\text{ptm_typ}')$ $\{-3\} \quad \text{cv}(\text{pointer_to_member?})(\text{ptm_typ}')$ $\{-4\} \quad \text{range_ptm}(\text{cv_base}(\text{ptm_typ}'))(\text{ofs}')$
$\{1\} \quad \text{Cpp_Type?}(\text{pointer}(\text{typ}(\text{cv_base}(\text{ptm_typ}')))) \wedge$ $\quad \text{cv}(\text{extend}[\text{Cpp_Type_}, \text{Cpp_Type}, \text{bool}, \text{FALSE}]$ $\quad \quad (\lambda (t : \text{Cpp_Type}) : \text{pointer?}(t) \wedge (\text{class?}(\text{typ}(t)) \vee \text{union?}(\text{typ}(t))))$ $\quad (\text{pointer}(\text{typ}(\text{cv_base}(\text{ptm_typ}'))))$

Case splitting on `Cpp_Type?(pointer(typ(cv_base(ptm_typ!1))))`,

we get 3 subgoals:

`ptm_arrow_TCC1.1.1`:

$\{-1\} \quad \text{Cpp_Type?}(\text{pointer}(\text{typ}(\text{cv_base}(\text{ptm_typ}'))))$ $\{-2\} \quad \text{pointer_to_member?}(\text{cv_base}(\text{ptm_typ}'))$ $\{-3\} \quad \text{Cpp_Type?}(\text{ptm_typ}')$ $\{-4\} \quad \text{cv}(\text{pointer_to_member?})(\text{ptm_typ}')$ $\{-5\} \quad \text{range_ptm}(\text{cv_base}(\text{ptm_typ}'))(\text{ofs}')$
$\{1\} \quad \text{Cpp_Type?}(\text{pointer}(\text{typ}(\text{cv_base}(\text{ptm_typ}')))) \wedge$ $\quad \text{cv}(\text{extend}[\text{Cpp_Type_}, \text{Cpp_Type}, \text{bool}, \text{FALSE}]$ $\quad \quad (\lambda (t : \text{Cpp_Type}) : \text{pointer?}(t) \wedge (\text{class?}(\text{typ}(t)) \vee \text{union?}(\text{typ}(t))))$ $\quad (\text{pointer}(\text{typ}(\text{cv_base}(\text{ptm_typ}'))))$

Simplifying, rewriting, and recording with decision procedures,

Expanding the definition of `extend`,

Keeping (-1 -2 -4 1) and hiding `*`,

Adding type constraints for `typ(cv_base(ptm_typ!1))`,

Simplifying, rewriting, and recording with decision procedures,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `ptm_arrow_TCC1.1.1`.

C Proof scripts

ptm_arrow_TCC1.1.2:

{-1}	pointer_to_member?(cv_base(ptm_typ'))
{-2}	Cpp_Type?(ptm_typ')
{-3}	cv(pointer_to_member?)(ptm_typ')
{-4}	range_ptm(cv_base(ptm_typ'))(ofs')
{1}	Cpp_Type?(pointer(typ(cv_base(ptm_typ'))))
{2}	Cpp_Type?(pointer(typ(cv_base(ptm_typ')))) \wedge cv(extend[Cpp_Type_, Cpp_Type, bool, FALSE] ($\lambda (t: \text{Cpp_Type}): \text{pointer?}(t) \wedge (\text{class?}(\text{typ}(t)) \vee \text{union?}(\text{typ}(t)))$) (pointer(typ(cv_base(ptm_typ'))))

Hiding formulas: 2,

Expanding the definition of Cpp_Type?,

Repeatedly Skolemizing and flattening,

Expanding the definition of subterm,

Splitting conjunctions,

we get 2 subgoals:

ptm_arrow_TCC1.1.2.1:

{-1}	$t' = \text{pointer}(\text{typ}(\text{cv_base}(\text{ptm_typ}')))$
{-2}	$\text{pointer_to_member}?(\text{cv_base}(\text{ptm_typ}'))$
{-3}	$\forall (t: \text{Cpp_Type_}):$ $\text{subterm}(t, \text{ptm_typ}') \supset$ $\text{no_pointers_to_bitfield}?(t) \wedge$ $\text{no_pointers_to_references}?(t) \wedge$ $\text{no_cv_references}?(t) \wedge$ $\text{no_reference_to_reference}?(t) \wedge$ $\text{no_reference_to_bitfields}?(t) \wedge$ $\text{no_pointer_to_member_to_reference}?(t) \wedge$ $\text{no_pointer_to_member_to_cv_void}?(t) \wedge$ $\text{no_cv_void_parameter}?(t) \wedge$ $\text{no_array_of_references}?(t) \wedge$ $\text{no_array_of_cv_void}?(t) \wedge$ $\text{no_array_of_function}?(t) \wedge$ $\text{no_array_of_abstract_class}?(t) \wedge$ $\text{no_pointer_or_ref_to_incomplete_array_parameter}?(t) \wedge$ $\text{no_array_return_type}?(t) \wedge$ $\text{no_function_return_type}?(t) \wedge$ $\text{bitfield_underlying_integral_or_enum_type}?(t) \wedge$ $\text{cv_array}?(t) \wedge$ $\text{enum_underlying_integral}?(t) \wedge$ $\text{enum_constants}?(t) \wedge$ $\text{const_volatile}?(t) \wedge$ $\text{const_stutter}?(t) \wedge$ $\text{volatile_stutter}?(t) \wedge$ $\text{no_cv_class}?(t) \wedge$ $\text{no_cv_union}?(t) \wedge$ $\text{no_cv_function}?(t) \wedge$ $\text{no_reference_to_void}?(t) \wedge$ $\text{no_cv_void}?(t) \wedge \text{no_array_of_bitfields}?(t)$
{-4}	$\text{cv}(\text{pointer_to_member}?(\text{ptm_typ}'))$
{-5}	$\text{range_ptm}(\text{cv_base}(\text{ptm_typ}'))(\text{ofs}')$
{1}	$\text{no_pointers_to_bitfield}?(t') \wedge$ $\text{no_pointers_to_references}?(t') \wedge$ $\text{no_cv_references}?(t') \wedge$ $\text{no_reference_to_reference}?(t') \wedge$ $\text{no_reference_to_bitfields}?(t') \wedge$ $\text{no_pointer_to_member_to_reference}?(t') \wedge$ $\text{no_pointer_to_member_to_cv_void}?(t') \wedge$ $\text{no_cv_void_parameter}?(t') \wedge$ $\text{no_array_of_references}?(t') \wedge$ $\text{no_array_of_cv_void}?(t') \wedge$ $\text{no_array_of_function}?(t') \wedge$ $\text{no_array_of_abstract_class}?(t') \wedge$ $\text{no_pointer_or_ref_to_incomplete_array_parameter}?(t') \wedge$ $\text{no_array_return_type}?(t') \wedge$ $\text{no_function_return_type}?(t') \wedge$ $\text{bitfield_underlying_integral_or_enum_type}?(t') \wedge$ $\text{cv_array}?(t') \wedge$ $\text{enum_underlying_integral}?(t') \wedge$ $\text{enum_constants}?(t') \wedge$ $\text{const_volatile}?(t') \wedge$ $\text{const_stutter}?(t') \wedge$ $\text{volatile_stutter}?(t') \wedge$ $\text{no_cv_class}?(t') \wedge$ $\text{no_cv_union}?(t') \wedge$ $\text{no_cv_function}?(t') \wedge$ $\text{no_reference_to_void}?(t') \wedge$ $\text{no_cv_void}?(t') \wedge \text{no_array_of_bitfields}?(t')$

C Proof scripts

Hiding formulas: -3,

Replacing using formula -1,

Installing automatic rewrites from: no_pointers_to_bitfield? no_pointers_to_references?
no_cv_references? no_reference_to_reference? no_reference_to_bitfields? no_pointer_to_member_to_reference?
no_pointer_to_member_to_cv_void? no_cv_void_parameter? no_array_of_references? no_array_of_bitfields?
no_array_of_cv_void? no_array_of_function? no_array_of_abstract_class? no_pointer_or_ref_to_incomplete_array_par
no_array_return_type? no_function_return_type? bitfield_underlying_integral_or_enum_type? cv_array?
enum_underlying_integral? enum_constants? const_volatile? const_stutter? volatile_stutter?
no_cv_class? no_cv_union? no_cv_function? no_reference_to_void? no_cv_void?

Adding type constraints for `typ(cv_base(ptm_typ!1))`,

Simplifying, rewriting, and recording with decision procedures,

Keeping (-1 -3) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `ptm_arrow_TCC1.1.2.1`.

ptm_arrow_TCC1.1.2.2:

{-1}	subterm(t' , typ(cv_base(ptm_typ')))
{-2}	pointer_to_member?(cv_base(ptm_typ'))
{-3}	$\forall (t: \text{Cpp_Type_}):$ subterm(t , ptm_typ') \supset no_pointers_to_bitfield?(t) \wedge no_pointers_to_references?(t) \wedge no_cv_references?(t) \wedge no_reference_to_reference?(t) \wedge no_reference_to_bitfields?(t) \wedge no_pointer_to_member_to_reference?(t) \wedge no_pointer_to_member_to_cv_void?(t) \wedge no_cv_void_parameter?(t) \wedge no_array_of_references?(t) \wedge no_array_of_cv_void?(t) \wedge no_array_of_function?(t) \wedge no_array_of_abstract_class?(t) \wedge no_pointer_or_ref_to_incomplete_array_parameter?(t) \wedge no_array_return_type?(t) \wedge no_function_return_type?(t) \wedge bitfield_underlying_integral_or_enum_type?(t) \wedge cv_array?(t) \wedge enum_underlying_integral?(t) \wedge enum_constants?(t) \wedge const_volatile?(t) \wedge const_stutter?(t) \wedge volatile_stutter?(t) \wedge no_cv_class?(t) \wedge no_cv_union?(t) \wedge no_cv_function?(t) \wedge no_reference_to_void?(t) \wedge no_cv_void?(t) \wedge no_array_of_bitfields?(t)
{-4}	cv(pointer_to_member?)(ptm_typ')
{-5}	range_ptm(cv_base(ptm_typ'))(ofs')
{1}	no_pointers_to_bitfield?(t') \wedge no_pointers_to_references?(t') \wedge no_cv_references?(t') \wedge no_reference_to_reference?(t') \wedge no_reference_to_bitfields?(t') \wedge no_pointer_to_member_to_reference?(t') \wedge no_pointer_to_member_to_cv_void?(t') \wedge no_cv_void_parameter?(t') \wedge no_array_of_references?(t') \wedge no_array_of_cv_void?(t') \wedge no_array_of_function?(t') \wedge no_array_of_abstract_class?(t') \wedge no_pointer_or_ref_to_incomplete_array_parameter?(t') \wedge no_array_return_type?(t') \wedge no_function_return_type?(t') \wedge bitfield_underlying_integral_or_enum_type?(t') \wedge cv_array?(t') \wedge enum_underlying_integral?(t') \wedge enum_constants?(t') \wedge const_volatile?(t') \wedge const_stutter?(t') \wedge volatile_stutter?(t') \wedge no_cv_class?(t') \wedge no_cv_union?(t') \wedge no_cv_function?(t') \wedge no_reference_to_void?(t') \wedge no_cv_void?(t') \wedge no_array_of_bitfields?(t')

C Proof scripts

Instantiating the top quantifier in -3 with the terms: (t!1),

Splitting conjunctions,

we get 2 subgoals:

ptm_arrow_TCC1.1.2.2.1:

{-1}	$\begin{aligned} & \text{no_pointers_to_bitfield?}(t') \wedge \\ & \text{no_pointers_to_references?}(t') \wedge \\ & \text{no_cv_references?}(t') \wedge \\ & \text{no_reference_to_reference?}(t') \wedge \\ & \text{no_reference_to_bitfields?}(t') \wedge \\ & \text{no_pointer_to_member_to_reference?}(t') \wedge \\ & \text{no_pointer_to_member_to_cv_void?}(t') \wedge \\ & \text{no_cv_void_parameter?}(t') \wedge \\ & \text{no_array_of_references?}(t') \wedge \\ & \text{no_array_of_cv_void?}(t') \wedge \\ & \text{no_array_of_function?}(t') \wedge \\ & \text{no_array_of_abstract_class?}(t') \wedge \\ & \text{no_pointer_or_ref_to_incomplete_array_parameter?}(t') \wedge \\ & \text{no_array_return_type?}(t') \wedge \\ & \text{no_function_return_type?}(t') \wedge \\ & \text{bitfield_underlying_integral_or_enum_type?}(t') \wedge \\ & \text{cv_array?}(t') \wedge \\ & \text{enum_underlying_integral?}(t') \wedge \\ & \text{enum_constants?}(t') \wedge \\ & \text{const_volatile?}(t') \wedge \\ & \text{const_stutter?}(t') \wedge \\ & \text{volatile_stutter?}(t') \wedge \\ & \text{no_cv_class?}(t') \wedge \\ & \text{no_cv_union?}(t') \wedge \\ & \text{no_cv_function?}(t') \wedge \\ & \text{no_reference_to_void?}(t') \wedge \\ & \text{no_cv_void?}(t') \wedge \text{no_array_of_bitfields?}(t') \end{aligned}$
{-2}	$\text{subterm}(t', \text{typ}(\text{cv_base}(\text{ptm_typ})))$
{-3}	$\text{pointer_to_member?}(\text{cv_base}(\text{ptm_typ}'))$
{-4}	$\text{cv}(\text{pointer_to_member?})(\text{ptm_typ}')$
{-5}	$\text{range_ptm}(\text{cv_base}(\text{ptm_typ}'))(\text{ofs}')$
{1}	$\begin{aligned} & \text{no_pointers_to_bitfield?}(t') \wedge \\ & \text{no_pointers_to_references?}(t') \wedge \\ & \text{no_cv_references?}(t') \wedge \\ & \text{no_reference_to_reference?}(t') \wedge \\ & \text{no_reference_to_bitfields?}(t') \wedge \\ & \text{no_pointer_to_member_to_reference?}(t') \wedge \\ & \text{no_pointer_to_member_to_cv_void?}(t') \wedge \\ & \text{no_cv_void_parameter?}(t') \wedge \\ & \text{no_array_of_references?}(t') \wedge \\ & \text{no_array_of_cv_void?}(t') \wedge \\ & \text{no_array_of_function?}(t') \wedge \\ & \text{no_array_of_abstract_class?}(t') \wedge \\ & \text{no_pointer_or_ref_to_incomplete_array_parameter?}(t') \wedge \\ & \text{no_array_return_type?}(t') \wedge \\ & \text{no_function_return_type?}(t') \wedge \\ & \text{bitfield_underlying_integral_or_enum_type?}(t') \wedge \\ & \text{cv_array?}(t') \wedge \\ & \text{enum_underlying_integral?}(t') \wedge \\ & \text{enum_constants?}(t') \wedge \\ & \text{const_volatile?}(t') \wedge \\ & \text{const_stutter?}(t') \wedge \\ & \text{volatile_stutter?}(t') \wedge \\ & \text{no_cv_class?}(t') \wedge \\ & \text{no_cv_union?}(t') \wedge \\ & \text{no_cv_function?}(t') \wedge \\ & \text{no_reference_to_void?}(t') \wedge \\ & \text{no_cv_void?}(t') \wedge \text{no_array_of_bitfields?}(t') \end{aligned}$

which is trivially true.

This completes the proof of `ptm_arrow_TCC1.1.2.2.1`.

`ptm_arrow_TCC1.1.2.2.2`:

<pre> {-1} subterm(t', typ(cv_base(ptm_typ'))) {-2} pointer_to_member?(cv_base(ptm_typ')) {-3} cv(pointer_to_member?)(ptm_typ') {-4} range_ptm(cv_base(ptm_typ'))(ofs') </pre>	<hr style="border: 0.5px solid black;"/> <pre> {1} subterm(t', ptm_typ') {2} no_pointers_to_bitfield?(t') ∧ no_pointers_to_references?(t') ∧ no_cv_references?(t') ∧ no_reference_to_reference?(t') ∧ no_reference_to_bitfields?(t') ∧ no_pointer_to_member_to_reference?(t') ∧ no_pointer_to_member_to_cv_void?(t') ∧ no_cv_void_parameter?(t') ∧ no_array_of_references?(t') ∧ no_array_of_cv_void?(t') ∧ no_array_of_function?(t') ∧ no_array_of_abstract_class?(t') ∧ no_pointer_or_ref_to_incomplete_array_parameter?(t') ∧ no_array_return_type?(t') ∧ no_function_return_type?(t') ∧ bitfield_underlying_integral_or_enum_type?(t') ∧ cv_array?(t') ∧ enum_underlying_integral?(t') ∧ enum_constants?(t') ∧ const_volatile?(t') ∧ const_stutter?(t') ∧ volatile_stutter?(t') ∧ no_cv_class?(t') ∧ no_cv_union?(t') ∧ no_cv_function?(t') ∧ no_reference_to_void?(t') ∧ no_cv_void?(t') ∧ no_array_of_bitfields?(t') </pre>
--	---

Hiding formulas: 2,

Using lemma `subterm_cv_base`,

Adding type constraints for `typ(cv_base(ptm_typ!1))`,

Simplifying, rewriting, and recording with decision procedures,

Expanding the definition of `subterm`,

Replacing using formula -3,

Using lemma `subterm_transitive`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Expanding the definition of `subterm`,

which is trivially true.

This completes the proof of `ptm_arrow_TCC1.1.2.2.2`.

ptm_arrow_TCC1.1.3:

{-1}	pointer_to_member?(cv_base(ptm_typ'))
{-2}	Cpp_Type?(ptm_typ')
{-3}	cv(pointer_to_member?)(ptm_typ')
{-4}	range_ptm(cv_base(ptm_typ'))(ofs')
{1}	array?(cv_base(ptm_typ')) \vee pointer?(cv_base(ptm_typ')) \vee reference?(cv_base(ptm_typ')) \vee bitfield?(cv_base(ptm_typ')) \vee enum?(cv_base(ptm_typ')) \vee pointer_to_member?(cv_base(ptm_typ')) \vee const?(cv_base(ptm_typ')) \vee volatile?(cv_base(ptm_typ'))
{2}	Cpp_Type?(pointer(typ(cv_base(ptm_typ')))) \wedge cv(extend[Cpp_Type_, Cpp_Type, bool, FALSE] ($\lambda (t: \text{Cpp_Type}): \text{pointer?}(t) \wedge (\text{class?}(\text{typ}(t)) \vee \text{union?}(\text{typ}(t)))$)) (pointer(typ(cv_base(ptm_typ')))))

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 This completes the proof of ptm_arrow_TCC1.1.3.

ptm_arrow_TCC1.2:

{-1}	Cpp_Type?(ptm_typ')
{-2}	cv(pointer_to_member?)(ptm_typ')
{-3}	range_ptm(cv_base(ptm_typ'))(ofs')
{1}	(pointer_to_member? \subseteq interpreted?)
{2}	Cpp_Type?(pointer(typ(cv_base(ptm_typ')))) \wedge cv(extend[Cpp_Type_, Cpp_Type, bool, FALSE] ($\lambda (t: \text{Cpp_Type}): \text{pointer?}(t) \wedge (\text{class?}(\text{typ}(t)) \vee \text{union?}(\text{typ}(t)))$)) (pointer(typ(cv_base(ptm_typ')))))

Keeping (1) and hiding *,
 Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of ptm_arrow_TCC1.2.
 Q.E.D.

C.185 Proofs for Posnat_induction (vfiasco-prelude.pvs)

C.185.1 Posnat_induction.posnat_induction

Terse proof for posnat_induction.

posnat_induction:

{1}	$\forall (p: \text{pred}[\text{posnat}]):$ $(p(1) \wedge (\forall j: p(j) \supset p(j+1))) \supset (\forall i: p(i))$
-----	--

For the top quantifier in 1, we introduce Skolem constants: (p'),
 Applying disjunctive simplification to flatten sequent,
 Inducting on i on formula 1,
 we get 3 subgoals:

C Proof scripts

posnat_induction.1:

{-1}	$p'(1)$
{-2}	$\forall j: p'(j) \supset p'(j+1)$
<hr/>	
{1}	$i' > 0$
{2}	$p'(i')$

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of posnat_induction.1.

posnat_induction.2:

{-1}	$p'(1)$
{-2}	$\forall j: p'(j) \supset p'(j+1)$
<hr/>	
{1}	$0 > 0 \supset p'(0)$

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of posnat_induction.2.

posnat_induction.3:

{-1}	$p'(1)$
{-2}	$\forall j: p'(j) \supset p'(j+1)$
<hr/>	
{1}	$\forall j: (j > 0 \supset p'(j)) \supset j+1 > 0 \supset p'(j+1)$

Repeatedly Skolemizing and flattening,
Instantiating quantified variables,
we get 2 subgoals:

posnat_induction.3.1:

{-1}	$j' \geq 0$
{-2}	$j' > 0 \supset p'(j')$
{-3}	$j' + 1 > 0$
{-4}	$p'(1)$
{-5}	$p'(j') \supset p'(j'+1)$
<hr/>	
{1}	$p'(j'+1)$

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of posnat_induction.3.1.

posnat_induction.3.2:

{-1}	$j' \geq 0$
{-2}	$j' > 0 \supset p'(j')$
{-3}	$j' + 1 > 0$
{-4}	$p'(1)$
<hr/>	
{1}	$j' > 0$
{2}	$p'(j'+1)$

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of posnat_induction.3.2.

Q.E.D.

C.186 Proofs for PostfixExpressions (expressions.pvs)

C.186.1 PostfixExpressions.postinc_TCC1

Terse proof for postinc_TCC1.

postinc_TCC1:

{1} $\forall ((\text{typ: Cpp_Subtype}(v(\text{non_bool_integral?})))): \text{cv}(\text{non_bool_integral_enum?})(\text{typ})$
--

Repeatedly Skolemizing and flattening,
 Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of postinc_TCC1.
 Q.E.D.

C.186.2 PostfixExpressions.postinc_float_TCC1

Terse proof for postinc_float_TCC1.

postinc_float_TCC1:

{1} $\forall ((\text{typ: Cpp_Subtype}(v(\text{floating_point?})))): \text{cv}(\text{floating_point?})(\text{typ})$
--

Repeatedly Skolemizing and flattening,
 Hiding formulas: -1,
 Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of postinc_float_TCC1.
 Q.E.D.

C.186.3 PostfixExpressions.postinc_float_TCC2

Terse proof for postinc_float_TCC2.

postinc_float_TCC2:

{1} $\forall ((\text{typ: Cpp_Subtype}(v(\text{floating_point?})))): \text{floating_point?}(\text{cv_base}(\text{typ}))$
--

Repeatedly Skolemizing and flattening,
 Rewriting using cv_base_result, matching in *,
 we get 2 subgoals:

postinc_float_TCC2.1:

{-1} Cpp_Type?(typ')
{-2} v(floating_point?)(typ')
{1} cv(floating_point?)(typ')
{2} floating_point?(cv_base(typ'))

Using lemma postinc_float_TCC1,
 This completes the proof of postinc_float_TCC2.1.

postinc_float_TCC2.2:

{-1} Cpp_Type?(typ')
{-2} v(floating_point?)(typ')
{1} (floating_point? \subseteq interpreted?)
{2} floating_point?(cv_base(typ'))

Keeping (1) and hiding *,
 Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of postinc_float_TCC2.2.
 Q.E.D.

C.186.4 PostfixExpressions.postinc_float_TCC3

Terse proof for postinc_float_TCC3.

postinc_float_TCC3:

$$\{1\} \quad \forall ((\text{typ}: \text{Cpp_Subtype}(v(\text{floating_point?}))), \\ n: (\text{range_floating_point}(\text{cv_base}(\text{typ})))): \\ \text{cv}(\text{floating_point?})(\text{cv_base}(\text{typ}))$$

Repeatedly Skolemizing and flattening,
Keeping (-2 1) and hiding *,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of postinc_float_TCC3.
Q.E.D.

C.187 Proofs for PrimaryExpressions (expressions.pvs)

This theory contains no provable formal statements.

C.188 Proofs for Pte_type (paging-data.pvs)

This theory contains no provable formal statements.

C.189 Proofs for Pte_type_adt (Pte_type_adt.pvs)

C.189.1 Pte_type_adt.Pte_TCC1

Terse proof for Pte_TCC1.

Pte_TCC1:

$$\{1\} \quad \forall (a: \text{Memory_Address_4G}): \text{offset}(a) \geq 0$$

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of Pte_TCC1.
Q.E.D.

C.190 Proofs for Pte_type_adt_reduce (Pte_type_adt.pvs)

This theory contains no provable formal statements.

C.191 Proofs for Random_Device (random_device.pvs)

C.191.1 Random_Device.base_TCC1

Terse proof for base_TCC1.

base_TCC1:

{1} $\exists (x: (\{m: \text{Memory_Address_4G} \mid m + \text{size} \leq \text{max_linear}\}))$: TRUE

Instantiating the top quantifier in 1 with the terms: (Mem(0)),
we get 2 subgoals:

base_TCC1.1:

{1} TRUE

which is trivially true.

This completes the proof of base_TCC1.1.

base_TCC1.2:

{1} $\text{Mem}?(\text{Mem}(0)' \text{type_of}) \wedge$
 $(0 \leq \text{Mem}(0)' \text{offset} \wedge \text{Mem}(0)' \text{offset} < \text{max_linear_offset}) \wedge$
 $\text{Mem}(0) + \text{size} \leq \text{max_linear}$

Using lemma dt_uint_size,

Expanding the definition of max_linear,

Expanding the definition of Mem,

Expanding the definition of +,

Expanding the definition of size,

Expanding the definition of <=,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of base_TCC1.2.

Q.E.D.

C.191.2 Random_Device.rnd_seed_TCC1

Terse proof for rnd_seed_TCC1.

rnd_seed_TCC1:

{1} $0 \leq (\text{base} + 0 \times \text{size}(\text{uidt}(\text{dt_uint})))' \text{offset} \wedge$
 $(\text{base} + 0 \times \text{size}(\text{uidt}(\text{dt_uint})))' \text{offset} < \text{max_linear_offset}$

Installing automatic rewrites from: + < ertimes_ax

Adding type constraints for base,

Expanding the definition of size,

Using lemma dt_uint_size,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of rnd_seed_TCC1.

Q.E.D.

C.191.3 Random_Device.mem_cnt_TCC1

Terse proof for mem_cnt_TCC1.

mem_cnt_TCC1:

{1} $0 \leq (\text{base} + 1 \times \text{size}(\text{uidt}(\text{dt_uint})))' \text{offset} \wedge$
 $(\text{base} + 1 \times \text{size}(\text{uidt}(\text{dt_uint})))' \text{offset} < \text{max_linear_offset}$

C Proof scripts

Using lemma `dt_uint_size`,
Adding type constraints for `base`,
Installing automatic rewrites from: `+ <= < max_linear size Mem ertimes_ax`
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `mem_cnt_TCC1`.
Q.E.D.

C.191.4 Random_Device.rnd_val_TCC1

Terse proof for `rnd_val_TCC1`.

`rnd_val_TCC1`:

$$\{1\} \quad 0 \leq (\text{base} + 2 \times \text{size}(\text{uidt}(\text{dt_uint})))' \text{offset} \wedge \\ (\text{base} + 2 \times \text{size}(\text{uidt}(\text{dt_uint})))' \text{offset} < \text{max_linear_offset}$$

Using lemma `dt_uint_size`,
Adding type constraints for `base`,
Installing automatic rewrites from: `+ <= < max_linear size Mem ertimes_ax`
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `rnd_val_TCC1`.
Q.E.D.

C.191.5 Random_Device.increase_access_count_TCC1

Terse proof for `increase_access_count_TCC1`.

`increase_access_count_TCC1`:

$$\{1\} \quad \forall (s: \text{Random_device_memory}, n: \text{nat}): \\ s' \text{expansion}' \text{counter_offset} \leq n + \text{access_count}(s)$$

Repeatedly Skolemizing and flattening,
Expanding the definition of `access_count`,
Adding type constraints for `s!1'expansion'counter_offset`,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `increase_access_count_TCC1`.
Q.E.D.

C.191.6 Random_Device.clear_counter_TCC1

Terse proof for `clear_counter_TCC1`.

`clear_counter_TCC1`:

$$\{1\} \quad \forall (s: \text{Random_device_memory}): \text{access_count}(s) \leq s' \text{expansion}' \text{memory_access_count}$$

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `clear_counter_TCC1`.
Q.E.D.

C.191.7 Random_Device.random_TCC1

Terse proof for random_TCC1.

random_TCC1:

$\{1\} \quad \exists (x_1: \text{[[nat, nat]} \rightarrow \{l: \text{list[Byte]} \mid \text{length[Byte]}(l) = \text{size}(\text{uidt}(\text{dt_uint}))\}]):$ <p style="text-align: center;">TRUE</p>
--

Instantiating the top quantifier in 1 with the terms: $(\lambda (n_1, n_2: \text{nat}): \text{choose}(\lambda (l: \text{list[Byte]}): \text{length}(l) = \text{size}(\text{uidt}(\text{dt_uint}))))$,

we get 2 subgoals:

random_TCC1.1:

$\{1\} \quad \text{TRUE}$

which is trivially true.

This completes the proof of random_TCC1.1.

random_TCC1.2:

$\{1\} \quad \text{nonempty?}[\text{list[Byte]}]$ $(\lambda (l: \text{list[Byte]}): \text{length[Byte]}(l) = \text{size}(\text{uidt}(\text{dt_uint})))$
--

Expanding the definition of nonempty?,

Expanding the definition of empty?,

Expanding the definition of member,

Rewriting using forall_not, matching in *,

Using lemma list_all_length[Byte],

This completes the proof of random_TCC1.2.

Q.E.D.

C.191.8 Random_Device.read_mem_cnt_TCC1

Terse proof for read_mem_cnt_TCC1.

read_mem_cnt_TCC1:

$\{1\} \quad \forall (a: \text{Address}, \text{bl}: \text{list[Byte]}):$ $\text{cons?}(\text{bl}) \wedge a = \text{mem_cnt} \supset (\forall (s: \text{Random_device_memory}): \text{non_bool_integral?}(\text{uint}))$
--

Repeatedly Skolemizing and flattening,

Keeping (1) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of read_mem_cnt_TCC1.

Q.E.D.

C.191.9 Random_Device.read_mem_cnt_TCC2

Terse proof for read_mem_cnt_TCC2.

read_mem_cnt_TCC2:

$$\frac{\{1\} \quad \forall (a: \text{Address}, \text{bl}: \text{list}[\text{Byte}]): \text{cons}?(bl) \wedge a = \text{mem_cnt} \supset (\forall (s: \text{Random_device_memory}): \text{max_value}(\text{uint}) \geq 0 \wedge \text{max_value}(\text{uint}) > 0)}{\quad}$$

Repeatedly Skolemizing and flattening,

Keeping (1) and hiding *,

Expanding the definition of max_value,

Expanding the definition of range_integral,

Using lemma max_uint,

Using lemma max_uint_bits,

Using lemma pos_expt_gt,

Case splitting on $\text{extend}[\text{real}, \text{int}, \text{bool}, \text{FALSE}] (\text{extend}[\text{int}, \text{nat}, \text{bool}, \text{FALSE}] (\text{range_uint})) = \text{extend}[\text{real}, \text{nat}, \text{bool}, \text{FALSE}] (\text{range_uint})$,

we get 2 subgoals:

read_mem_cnt_TCC2.1:

$$\frac{\begin{array}{l} \{-1\} \quad \text{extend}[\text{real}, \text{int}, \text{bool}, \text{FALSE}] (\text{extend}[\text{int}, \text{nat}, \text{bool}, \text{FALSE}] (\text{range_uint})) = \\ \quad \text{extend}[\text{real}, \text{nat}, \text{bool}, \text{FALSE}] (\text{range_uint}) \\ \{-2\} \quad \text{max_value_bits_uint} < 2 \wedge \text{max_value_bits_uint} \\ \{-3\} \quad \text{max_value_bits_uint} \geq 16 \\ \{-4\} \quad \text{max}(\text{range_uint}) \geq (2 \wedge \text{max_value_bits_uint}) - 1 \end{array}}{\{1\} \quad \text{max}(\text{extend}[\text{real}, \text{int}, \text{bool}, \text{FALSE}] (\text{extend}[\text{int}, \text{nat}, \text{bool}, \text{FALSE}] (\text{range_uint}))) \geq 0 \wedge \text{max}(\text{extend}[\text{real}, \text{int}, \text{bool}, \text{FALSE}] (\text{extend}[\text{int}, \text{nat}, \text{bool}, \text{FALSE}] (\text{range_uint}))) > 0}$$

Replacing using formula -1,

Using lemma max_stable_under_extension[real, nat, ≤],

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of read_mem_cnt_TCC2.1.

read_mem_cnt_TCC2.2:

$$\frac{\begin{array}{l} \{-1\} \quad \text{max_value_bits_uint} < 2 \wedge \text{max_value_bits_uint} \\ \{-2\} \quad \text{max_value_bits_uint} \geq 16 \\ \{-3\} \quad \text{max}(\text{range_uint}) \geq (2 \wedge \text{max_value_bits_uint}) - 1 \end{array}}{\begin{array}{l} \{1\} \quad \text{extend}[\text{real}, \text{int}, \text{bool}, \text{FALSE}] (\text{extend}[\text{int}, \text{nat}, \text{bool}, \text{FALSE}] (\text{range_uint})) = \\ \quad \text{extend}[\text{real}, \text{nat}, \text{bool}, \text{FALSE}] (\text{range_uint}) \\ \{2\} \quad \text{max}(\text{extend}[\text{real}, \text{int}, \text{bool}, \text{FALSE}] (\text{extend}[\text{int}, \text{nat}, \text{bool}, \text{FALSE}] (\text{range_uint}))) \geq 0 \wedge \\ \quad \text{max}(\text{extend}[\text{real}, \text{int}, \text{bool}, \text{FALSE}] (\text{extend}[\text{int}, \text{nat}, \text{bool}, \text{FALSE}] (\text{range_uint}))) > 0 \end{array}}$$

Keeping (1) and hiding *,

Applying decompose-equality,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of read_mem_cnt_TCC2.2.

Q.E.D.

C.191.10 Random_Device.read_mem_cnt_TCC3

Terse proof for read_mem_cnt_TCC3.

read_mem_cnt_TCC3:

$$\frac{\{1\} \quad \forall (a: \text{Address}, \text{bl}: \text{list}[\text{Byte}]): \\ \text{cons?}(\text{bl}) \wedge a = \text{mem_cnt} \supset \\ (\forall (s: \text{Random_device_memory}): \\ (\text{range_uint})(\text{rem}(\text{max_value}(\text{uint})) \\ (\text{access_count}(s) - \text{counter_offset}(s))))}{}$$

Repeatedly Skolemizing and flattening,

Keeping (1) and hiding *,

Adding type constraints for $\text{rem}(\text{max_value}(\text{uint}))(\text{access_count}(s!1) - \text{counter_offset}(s!1))$,

we get 2 subgoals:

read_mem_cnt_TCC3.1:

$$\frac{\begin{array}{l} \{-1\} \quad \text{rem}(\text{max_value}(\text{uint}))(\text{access_count}(s') - \text{counter_offset}(s')) < \text{max_value}(\text{uint}) \\ \{-2\} \quad \exists q: \\ \quad \text{access_count}(s') - \text{counter_offset}(s') = \\ \quad \text{rem}(\text{max_value}(\text{uint}))(\text{access_count}(s') - \text{counter_offset}(s')) + \text{max_value}(\text{uint}) \times q \end{array}}{\{1\} \quad (\text{range_uint})(\text{rem}(\text{max_value}(\text{uint}))(\text{access_count}(s') - \text{counter_offset}(s')))}{}$$

Using lemma binary_range_uint,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Keeping (-1 2) and hiding *,

Expanding the definition of max_value,

Expanding the definition of range_integral,

Case splitting on $\text{extend}[\text{real}, \text{int}, \text{bool}, \text{FALSE}] (\text{extend}[\text{int}, \text{nat}, \text{bool}, \text{FALSE}] (\text{range_uint})) = \text{extend}[\text{real}, \text{nat}, \text{bool}, \text{FALSE}] (\text{range_uint})$,

we get 2 subgoals:

read_mem_cnt_TCC3.1.1:

$$\frac{\begin{array}{l} \{-1\} \quad \text{extend}[\text{real}, \text{int}, \text{bool}, \text{FALSE}] (\text{extend}[\text{int}, \text{nat}, \text{bool}, \text{FALSE}] (\text{range_uint})) = \\ \quad \text{extend}[\text{real}, \text{nat}, \text{bool}, \text{FALSE}] (\text{range_uint}) \\ \{-2\} \quad \text{rem}(\text{max}(\text{extend}[\text{real}, \text{int}, \text{bool}, \text{FALSE}] (\text{extend}[\text{int}, \text{nat}, \text{bool}, \text{FALSE}] (\text{range_uint})))) \\ \quad (\text{access_count}(s') - \text{counter_offset}(s')) \\ \quad < \text{max}(\text{extend}[\text{real}, \text{int}, \text{bool}, \text{FALSE}] (\text{extend}[\text{int}, \text{nat}, \text{bool}, \text{FALSE}] (\text{range_uint})))) \end{array}}{\{1\} \quad \text{rem}(\text{max}(\text{extend}[\text{real}, \text{int}, \text{bool}, \text{FALSE}] (\text{extend}[\text{int}, \text{nat}, \text{bool}, \text{FALSE}] (\text{range_uint})))) \\ (\text{access_count}(s') - \text{counter_offset}(s')) \\ \leq \text{max}(\text{range_uint})}{}$$

Replacing using formula -1,

Using lemma max_stable_under_extension[real, nat, ≤],

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of read_mem_cnt_TCC3.1.1.

read_mem_cnt_TCC3.1.2:

$$\frac{\{-1\} \quad \text{rem}(\text{max}(\text{extend}[\text{real}, \text{int}, \text{bool}, \text{FALSE}] (\text{extend}[\text{int}, \text{nat}, \text{bool}, \text{FALSE}] (\text{range_uint})))) \\ (\text{access_count}(s') - \text{counter_offset}(s')) \\ < \text{max}(\text{extend}[\text{real}, \text{int}, \text{bool}, \text{FALSE}] (\text{extend}[\text{int}, \text{nat}, \text{bool}, \text{FALSE}] (\text{range_uint}))))}{\{1\} \quad \text{extend}[\text{real}, \text{int}, \text{bool}, \text{FALSE}] (\text{extend}[\text{int}, \text{nat}, \text{bool}, \text{FALSE}] (\text{range_uint})) = \\ \text{extend}[\text{real}, \text{nat}, \text{bool}, \text{FALSE}] (\text{range_uint})} \\ \{2\} \quad \text{rem}(\text{max}(\text{extend}[\text{real}, \text{int}, \text{bool}, \text{FALSE}] (\text{extend}[\text{int}, \text{nat}, \text{bool}, \text{FALSE}] (\text{range_uint})))) \\ (\text{access_count}(s') - \text{counter_offset}(s')) \\ \leq \text{max}(\text{range_uint})$$

Keeping (1) and hiding *,

C Proof scripts

Applying decompose-equality,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `read_mem_cnt_TCC3.1.2`.
`read_mem_cnt_TCC3.2`:

$\{1\}$ $\text{max_value}(\text{uint}) \geq 0$ $\{2\}$ $(\text{range_uint})(\text{rem}(\text{max_value}(\text{uint}))(\text{access_count}(s') - \text{counter_offset}(s')))$
--

Using lemma `read_mem_cnt_TCC2`,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `read_mem_cnt_TCC3.2`.
Q.E.D.

C.191.11 Random_Device.read_seed_TCC1

Terse proof for `read_seed_TCC1`.

`read_seed_TCC1`:

$\{1\}$ $\forall (a: \text{Address}, \text{bl}: \text{list}[\text{Byte}]):$ $\text{cons}?(bl) \wedge a = \text{rnd_seed} \supset (\forall (s: \text{Random_device_memory}): \text{non_bool_integral}?(uint))$

Repeatedly Skolemizing and flattening,
Expanding the definition of `non_bool_integral?`,
Expanding the definition of `unsigned_integer?`,
which is trivially true.
This completes the proof of `read_seed_TCC1`.
Q.E.D.

C.191.12 Random_Device.read_seed_TCC2

Terse proof for `read_seed_TCC2`.

`read_seed_TCC2`:

$\{1\}$ $\forall (a: \text{Address}, \text{bl}: \text{list}[\text{Byte}]):$ $\text{cons}?(bl) \wedge a = \text{rnd_seed} \supset$ $(\forall (s: \text{Random_device_memory}): \text{max_value}(\text{uint}) \geq 0 \wedge \text{max_value}(\text{uint}) > 0)$
--

Repeatedly Skolemizing and flattening,
Keeping (1) and hiding *,
Using lemma `max_uint`,
Using lemma `max_uint_bits`,
Using lemma `pos_expt_gt`,
Expanding the definition of `max_value`,
Expanding the definition of `range_integral`,
Case splitting on `extend[real, int, bool, FALSE] (extend[int, nat, bool, FALSE](range_uint)) = extend[real, nat, bool, FALSE](range_uint)`,
we get 2 subgoals:

read_seed_TCC2.1:

$$\begin{array}{l}
 \{-1\} \quad \text{extend}[\text{real}, \text{int}, \text{bool}, \text{FALSE}](\text{extend}[\text{int}, \text{nat}, \text{bool}, \text{FALSE}](\text{range_uint})) = \\
 \quad \text{extend}[\text{real}, \text{nat}, \text{bool}, \text{FALSE}](\text{range_uint}) \\
 \{-2\} \quad \text{max_value_bits_uint} < 2 \wedge \text{max_value_bits_uint} \\
 \{-3\} \quad \text{max_value_bits_uint} \geq 16 \\
 \{-4\} \quad \text{max}(\text{range_uint}) \geq (2 \wedge \text{max_value_bits_uint}) - 1 \\
 \hline
 \{1\} \quad \text{max}(\text{extend}[\text{real}, \text{int}, \text{bool}, \text{FALSE}](\text{extend}[\text{int}, \text{nat}, \text{bool}, \text{FALSE}](\text{range_uint}))) \geq 0 \wedge \\
 \quad \text{max}(\text{extend}[\text{real}, \text{int}, \text{bool}, \text{FALSE}](\text{extend}[\text{int}, \text{nat}, \text{bool}, \text{FALSE}](\text{range_uint}))) > 0
 \end{array}$$

Replacing using formula -1,

Using lemma max_stable_under_extension[real, nat, ≤],

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of read_seed_TCC2.1.

read_seed_TCC2.2:

$$\begin{array}{l}
 \{-1\} \quad \text{max_value_bits_uint} < 2 \wedge \text{max_value_bits_uint} \\
 \{-2\} \quad \text{max_value_bits_uint} \geq 16 \\
 \{-3\} \quad \text{max}(\text{range_uint}) \geq (2 \wedge \text{max_value_bits_uint}) - 1 \\
 \hline
 \{1\} \quad \text{extend}[\text{real}, \text{int}, \text{bool}, \text{FALSE}](\text{extend}[\text{int}, \text{nat}, \text{bool}, \text{FALSE}](\text{range_uint})) = \\
 \quad \text{extend}[\text{real}, \text{nat}, \text{bool}, \text{FALSE}](\text{range_uint}) \\
 \{2\} \quad \text{max}(\text{extend}[\text{real}, \text{int}, \text{bool}, \text{FALSE}](\text{extend}[\text{int}, \text{nat}, \text{bool}, \text{FALSE}](\text{range_uint}))) \geq 0 \wedge \\
 \quad \text{max}(\text{extend}[\text{real}, \text{int}, \text{bool}, \text{FALSE}](\text{extend}[\text{int}, \text{nat}, \text{bool}, \text{FALSE}](\text{range_uint}))) > 0
 \end{array}$$

Keeping (1) and hiding *,

Applying decompose-equality,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of read_seed_TCC2.2.

Q.E.D.

C.191.13 Random_Device.read_seed_TCC3

Terse proof for read_seed_TCC3.

read_seed_TCC3:

$$\begin{array}{l}
 \{1\} \quad \forall (a: \text{Address}, \text{bl}: \text{list}[\text{Byte}]): \\
 \quad \text{cons}?(bl) \wedge a = \text{rnd_seed} \supset \\
 \quad (\forall (s: \text{Random_device_memory}): (\text{range_uint})(\text{rem}(\text{max_value}(\text{uint}))(\text{seed}(s))))
 \end{array}$$

Repeatedly Skolemizing and flattening,

Using lemma read_seed_TCC2,

Using lemma binary_range_uint,

we get 2 subgoals:

C Proof scripts

read_seed_TCC3.1:

{-1}	range_uint(rem(max_value(uint))(seed(s'))) \equiv rem(max_value(uint))(seed(s')) \leq max(range_uint)
{-2}	cons?(bl') \wedge a' = rnd_seed \supset (\forall (s: Random_device_memory): max_value(uint) \geq 0 \wedge max_value(uint) $>$ 0)
{-3}	every(λ (x: number): number_field_pred(x) \wedge real_pred(x) \wedge rational_pred(x) \wedge integer_pred(x) \wedge x \geq 0 \wedge x $<$ max_byte)
	(bl')
{-4}	cons?(bl')
{-5}	a' = rnd_seed
{1}	(range_uint)(rem(max_value(uint))(seed(s')))

Adding type constraints for rem(max_value(uint))(seed(s!1)),

we get 2 subgoals:

read_seed_TCC3.1.1:

{-1}	rem(max_value(uint))(seed(s')) $<$ max_value(uint)
{-2}	\exists q: seed(s') = rem(max_value(uint))(seed(s')) + max_value(uint) \times q
{-3}	range_uint(rem(max_value(uint))(seed(s'))) \equiv rem(max_value(uint))(seed(s')) \leq max(range_uint)
{-4}	cons?(bl') \wedge a' = rnd_seed \supset (\forall (s: Random_device_memory): max_value(uint) \geq 0 \wedge max_value(uint) $>$ 0)
{-5}	every(λ (x: number): number_field_pred(x) \wedge real_pred(x) \wedge rational_pred(x) \wedge integer_pred(x) \wedge x \geq 0 \wedge x $<$ max_byte)
	(bl')
{-6}	cons?(bl')
{-7}	a' = rnd_seed
{1}	(range_uint)(rem(max_value(uint))(seed(s')))

Expanding the definition of max_value,

Expanding the definition of range_integral,

Case splitting on extend[real, int, bool, FALSE] (extend[int, nat, bool, FALSE](range_uint)) = extend[real, nat, bool, FALSE](range_uint),

we get 2 subgoals:

read_seed_TCC3.1.1.1.1:

$$\begin{array}{l}
 \{-1\} \text{ extend}[\text{real}, \text{int}, \text{bool}, \text{FALSE}](\text{extend}[\text{int}, \text{nat}, \text{bool}, \text{FALSE}](\text{range_uint})) = \\
 \quad \text{extend}[\text{real}, \text{nat}, \text{bool}, \text{FALSE}](\text{range_uint}) \\
 \{-2\} \text{ rem}(\max(\text{extend}[\text{real}, \text{int}, \text{bool}, \text{FALSE}](\text{extend}[\text{int}, \text{nat}, \text{bool}, \text{FALSE}](\text{range_uint})))) \\
 \quad (\text{seed}(s')) \\
 \quad < \max(\text{extend}[\text{real}, \text{int}, \text{bool}, \text{FALSE}](\text{extend}[\text{int}, \text{nat}, \text{bool}, \text{FALSE}](\text{range_uint}))) \\
 \{-3\} \exists q: \\
 \quad \text{seed}(s') = \\
 \quad \text{rem}(\max(\text{extend}[\text{real}, \text{int}, \text{bool}, \text{FALSE}](\text{extend}[\text{int}, \text{nat}, \text{bool}, \text{FALSE}](\text{range_uint}))))(\text{seed}(s')) + \max \\
 \{-4\} \text{ range_uint}(\text{rem}(\max(\text{extend}[\text{real}, \text{int}, \text{bool}, \text{FALSE}] \\
 \quad (\text{extend}[\text{int}, \text{nat}, \text{bool}, \text{FALSE}](\text{range_uint})))) \\
 \quad (\text{seed}(s')))) \\
 \quad \equiv \\
 \quad \text{rem}(\max(\text{extend}[\text{real}, \text{int}, \text{bool}, \text{FALSE}](\text{extend}[\text{int}, \text{nat}, \text{bool}, \text{FALSE}](\text{range_uint})))) \\
 \quad (\text{seed}(s')) \\
 \quad \leq \max(\text{range_uint}) \\
 \{-5\} \text{ cons?}(\text{bl}') \wedge a' = \text{rnd_seed} \supset \\
 \quad (\forall (s: \text{Random_device_memory}): \\
 \quad \max(\text{extend}[\text{real}, \text{int}, \text{bool}, \text{FALSE}](\text{extend}[\text{int}, \text{nat}, \text{bool}, \text{FALSE}](\text{range_uint}))) \geq 0 \\
 \quad \wedge \\
 \quad \max(\text{extend}[\text{real}, \text{int}, \text{bool}, \text{FALSE}](\text{extend}[\text{int}, \text{nat}, \text{bool}, \text{FALSE}](\text{range_uint}))) > \\
 \quad 0) \\
 \{-6\} \text{ every}(\lambda (x: \text{number}): \\
 \quad \text{number_field_pred}(x) \wedge \\
 \quad \text{real_pred}(x) \wedge \\
 \quad \text{rational_pred}(x) \wedge \text{integer_pred}(x) \wedge x \geq 0 \wedge x < \text{max_byte}) \\
 \quad (\text{bl}') \\
 \{-7\} \text{ cons?}(\text{bl}') \\
 \{-8\} a' = \text{rnd_seed} \\
 \hline
 \{-1\} (\text{range_uint})(\text{rem}(\max(\text{extend}[\text{real}, \text{int}, \text{bool}, \text{FALSE}] \\
 \quad (\text{extend}[\text{int}, \text{nat}, \text{bool}, \text{FALSE}](\text{range_uint})))) \\
 \quad (\text{seed}(s'))))
 \end{array}$$

Using lemma `max_stable_under_extension[real, nat, ≤]`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `read_seed_TCC3.1.1.1`.

read_seed_TCC3.1.1.2:

<p>{-1} $\text{rem}(\text{max}(\text{extend}[\text{real}, \text{int}, \text{bool}, \text{FALSE}](\text{extend}[\text{int}, \text{nat}, \text{bool}, \text{FALSE}](\text{range_uint}))))$ $(\text{seed}(s'))$ $< \text{max}(\text{extend}[\text{real}, \text{int}, \text{bool}, \text{FALSE}](\text{extend}[\text{int}, \text{nat}, \text{bool}, \text{FALSE}](\text{range_uint})))$</p> <p>{-2} $\exists q:$ $\text{seed}(s') =$ $\text{rem}(\text{max}(\text{extend}[\text{real}, \text{int}, \text{bool}, \text{FALSE}](\text{extend}[\text{int}, \text{nat}, \text{bool}, \text{FALSE}](\text{range_uint}))))(\text{seed}(s'))$</p> <p>{-3} $\text{range_uint}(\text{rem}(\text{max}(\text{extend}[\text{real}, \text{int}, \text{bool}, \text{FALSE}](\text{extend}[\text{int}, \text{nat}, \text{bool}, \text{FALSE}](\text{range_uint}))))$ $(\text{seed}(s'))$ \equiv $\text{rem}(\text{max}(\text{extend}[\text{real}, \text{int}, \text{bool}, \text{FALSE}](\text{extend}[\text{int}, \text{nat}, \text{bool}, \text{FALSE}](\text{range_uint}))))$ $(\text{seed}(s'))$ $\leq \text{max}(\text{range_uint})$</p> <p>{-4} $\text{cons?}(\text{bl}') \wedge a' = \text{rnd_seed} \supset$ $(\forall (s: \text{Random_device_memory}):$ $\text{max}(\text{extend}[\text{real}, \text{int}, \text{bool}, \text{FALSE}](\text{extend}[\text{int}, \text{nat}, \text{bool}, \text{FALSE}](\text{range_uint}))) \geq 0$ \wedge $\text{max}(\text{extend}[\text{real}, \text{int}, \text{bool}, \text{FALSE}](\text{extend}[\text{int}, \text{nat}, \text{bool}, \text{FALSE}](\text{range_uint}))) >$ $0)$</p> <p>{-5} $\text{every}(\lambda (x: \text{number}):$ $\text{number_field_pred}(x) \wedge$ $\text{real_pred}(x) \wedge$ $\text{rational_pred}(x) \wedge \text{integer_pred}(x) \wedge x \geq 0 \wedge x < \text{max_byte})$ (bl')</p> <p>{-6} $\text{cons?}(\text{bl}')$</p> <p>{-7} $a' = \text{rnd_seed}$</p>	<hr style="border: 0.5px solid black;"/> <p>{1} $\text{extend}[\text{real}, \text{int}, \text{bool}, \text{FALSE}](\text{extend}[\text{int}, \text{nat}, \text{bool}, \text{FALSE}](\text{range_uint})) =$ $\text{extend}[\text{real}, \text{nat}, \text{bool}, \text{FALSE}](\text{range_uint})$</p> <p>{2} $(\text{range_uint})(\text{rem}(\text{max}(\text{extend}[\text{real}, \text{int}, \text{bool}, \text{FALSE}](\text{extend}[\text{int}, \text{nat}, \text{bool}, \text{FALSE}](\text{range_uint}))))$ $(\text{seed}(s'))$</p>
--	---

Keeping (1) and hiding *,

Applying decompose-equality,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of read_seed_TCC3.1.1.2.

read_seed_TCC3.1.2:

<p>{-1} $\text{range_uint}(\text{rem}(\text{max_value}(\text{uint}))(\text{seed}(s'))) \equiv$ $\text{rem}(\text{max_value}(\text{uint}))(\text{seed}(s')) \leq \text{max}(\text{range_uint})$</p> <p>{-2} $\text{cons?}(\text{bl}') \wedge a' = \text{rnd_seed} \supset$ $(\forall (s: \text{Random_device_memory}): \text{max_value}(\text{uint}) \geq 0 \wedge \text{max_value}(\text{uint}) > 0)$</p> <p>{-3} $\text{every}(\lambda (x: \text{number}):$ $\text{number_field_pred}(x) \wedge$ $\text{real_pred}(x) \wedge$ $\text{rational_pred}(x) \wedge \text{integer_pred}(x) \wedge x \geq 0 \wedge x < \text{max_byte})$ (bl')</p> <p>{-4} $\text{cons?}(\text{bl}')$</p> <p>{-5} $a' = \text{rnd_seed}$</p>	<hr style="border: 0.5px solid black;"/> <p>{1} $\text{max_value}(\text{uint}) \geq 0$</p> <p>{2} $(\text{range_uint})(\text{rem}(\text{max_value}(\text{uint}))(\text{seed}(s')))$</p>
--	--

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `read_seed_TCC3.1.2`.

`read_seed_TCC3.2`:

<div style="display: flex; flex-direction: column; gap: 5px;"> <div style="display: flex; align-items: flex-start;"> {-1} $\text{cons?}(bl') \wedge a' = \text{rnd_seed} \supset$ </div> <div style="margin-left: 20px;"> $(\forall (s: \text{Random_device_memory}): \text{max_value}(\text{uint}) \geq 0 \wedge \text{max_value}(\text{uint}) > 0)$ </div> <div style="display: flex; align-items: flex-start;"> {-2} $\text{every}(\lambda (x: \text{number}):$ </div> <div style="margin-left: 20px;"> $\text{number_field_pred}(x) \wedge$ $\text{real_pred}(x) \wedge$ $\text{rational_pred}(x) \wedge \text{integer_pred}(x) \wedge x \geq 0 \wedge x < \text{max_byte})$ </div> <div style="margin-left: 20px;"> (bl') </div> <div style="display: flex; align-items: flex-start;"> {-3} $\text{cons?}(bl')$ </div> <div style="display: flex; align-items: flex-start;"> {-4} $a' = \text{rnd_seed}$ </div> </div>	<hr style="border: 0.5px solid black;"/> <div style="display: flex; flex-direction: column; gap: 5px;"> <div style="display: flex; align-items: flex-start;"> {1} $\text{max_value}(\text{uint}) \geq 0$ </div> <div style="display: flex; align-items: flex-start;"> {2} $(\text{range_uint})(\text{rem}(\text{max_value}(\text{uint}))(\text{seed}(s')))$ </div> </div>
--	--

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `read_seed_TCC3.2`.

Q.E.D.

C.192 Proofs for Read_After_Write_Fails (datatype_model.pvs)

C.192.1 Read_After_Write_Fails.plain_mem_write_list_states_TCC1

Terse proof for `plain_mem_write_list_states_TCC1`.

`plain_mem_write_list_states_TCC1`:

<div style="display: flex; flex-direction: column; gap: 5px;"> <div style="display: flex; align-items: flex-start;"> {1} $\forall (a: \text{Address}, pm: \text{Plain_Memory}[\text{State}], s: \text{State}, \text{size}: \text{nat}, bl: \text{list}[\text{Byte}]):$ </div> <div style="margin-left: 20px;"> $\text{size} = \text{length}(bl) \wedge$ $\text{plain_memory?}(pm) \wedge$ $pm \text{'states}(s) \wedge$ $(\text{address_block}(a, \text{size}) \subseteq pm \text{'rw_addr}) \wedge$ $\text{OK?}(\text{memory_write_list}(pm \text{'mem})(a, bl)(s))$ </div> <div style="margin-left: 20px;"> \supset </div> <div style="margin-left: 20px;"> $\text{OK?}[\text{State}, \text{Unit}](\text{memory_write_list}[\text{State}](pm \text{'mem})(a, bl)(s)) \vee$ $\text{Exception?}[\text{State}, \text{Unit}](\text{memory_write_list}[\text{State}](pm \text{'mem})(a, bl)(s))$ </div> </div>	<hr style="border: 0.5px solid black;"/>
--	--

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `plain_mem_write_list_states_TCC1`.

Q.E.D.

C.192.2 Read_After_Write_Fails.plain_mem_write_list_states

Terse proof for `plain_mem_write_list_states`.

C Proof scripts

`plain_mem_write_list_states:`

<pre>{1} ∃ (a: Address, pm: Plain_Memory[State], s: State, size: nat, bl: list[Byte]): size = length(bl) ∧ plain_memory?(pm) ∧ pm' states(s) ∧ (address_block(a, size) ⊆ pm'rw_addr) ∧ OK?(memory_write_list(pm' mem)(a, bl)(s)) ⊃ pm' states(state(memory_write_list(pm' mem)(a, bl)(s)))</pre>

Repeatedly Skolemizing and flattening,

Using lemma `transformer_invariant_write_list[State]`,

Replacing using formula -4,

Using lemma `plain_memory_side_effect_content_unchanged_write_block`,

Replacing using formula -4,

Replacing using formula -3,

Replacing using formula -6,

Replacing using formula -1,

Rewriting using `add_as_union`, matching in *,

Rewriting using `transformer_invariant_union_transformers`, matching in *,

we get 3 subgoals:

`plain_mem_write_list_states.1:`

<pre>{-1} side_effect_content_unchanged(address_block(a', length(bl')), pm' states, memory_write_side_effect(pm' mem)) {-2} result_pred(pm' states)(expr_2_super(memory_write_list(pm' mem)(a', bl'))(s')) {-3} size' = length(bl') {-4} plain_memory?(pm') {-5} pm' states(s') {-6} (address_block(a', length(bl')) ⊆ pm'rw_addr) {-7} OK?(memory_write_list(pm' mem)(a', bl')(s'))</pre> <hr/> <pre>{1} pm' states(state(memory_write_list(pm' mem)(a', bl')(s')))</pre>

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `plain_mem_write_list_states.1`.

`plain_mem_write_list_states.2:`

<pre>{-1} side_effect_content_unchanged(address_block(a', length(bl')), pm' states, memory_write_side_effect(pm' mem)) {-2} transformer_invariant?(pm' states, (memory_write_transformers(pm' mem, address_block(a', length(bl')) ⊃ result_pred(pm' states)(expr_2_super(memory_write_list(pm' mem)(a', bl'))(s')))) {-3} size' = length(bl') {-4} plain_memory?(pm') {-5} pm' states(s') {-6} (address_block(a', length(bl')) ⊆ pm'rw_addr) {-7} OK?(memory_write_list(pm' mem)(a', bl')(s'))</pre> <hr/> <pre>{1} transformer_invariant?(pm' states, memory_write_transformers(pm' mem, address_block(a', length(bl')) ⊃ result_pred(pm' states)(expr_2_super(memory_write_list(pm' mem)(a', bl'))(s')))) {2} pm' states(state(memory_write_list(pm' mem)(a', bl')(s')))</pre>

Using lemma `plain_memory_invariant_write_block`,

Replacing using formula -5,

Replacing using formula -7,

which is trivially true.

This completes the proof of plain_mem_write_list_states.2.

plain_mem_write_list_states.3:

{-1}	side_effect_content_unchanged(address_block(a', length(bl')), pm' 'states, memory_write_side_effect(pm' 'mem))
{-2}	transformer_invariant?(pm' 'states, (memory_write_transformers(pm' 'mem, address_block(a', length(bl'))) ∪ singleton ⊃ result_pred(pm' 'states)(expr_2_super(memory_write_list(pm' 'mem)(a', bl'))(s'))
{-3}	size' = length(bl')
{-4}	plain_memory?(pm')
{-5}	pm' 'states(s')
{-6}	(address_block(a', length(bl'))) ⊆ pm' 'rw_addr
{-7}	OK?(memory_write_list(pm' 'mem)(a', bl')(s'))
{1}	transformer_invariant?(pm' 'states, singleton(expr_2_super(memory_write_side_effect(pm' 'mem) (a', bl', FALSE))))
{2}	pm' 'states(state(memory_write_list(pm' 'mem)(a', bl')(s'))

Using lemma plain_memory_transformer_invariant_write_side_effects,

Replacing using formula -5,

Replacing using formula -7,

Applying transformer_invariant_mono_transformers where transformers_1 gets singleton(expr_2_super(memory_write_side_effect(pm!1, bl!1, FALSE))), transformers_2 gets memory_write_side_effect_super_transformers(pm!1'mem, address_block(a!1, length(bl!1))),

Instantiating quantified variables,

Replacing using formula -2,

Hiding formulas: -2, -3, -4, 2,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Hiding formulas: 2,

Expanding the definition of subset?,

Expanding the definition of memory_write_side_effect_super_transformers,

Expanding the definition of expr_2_super,

Expanding the definition of singleton,

Expanding the definition of member,

Repeatedly Skolemizing and flattening,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of plain_mem_write_list_states.3.

Q.E.D.

C.192.3 Read_After_Write_Fails.plain_mem_write_list_read_list_ok

Terse proof for plain_mem_write_list_read_list_ok.

plain_mem_write_list_read_list_ok:

$\{1\} \quad \forall (a: \text{Address}, pm: \text{Plain_Memory}[\text{State}], s: \text{State}, \text{size}: \text{nat}, \text{bl}: \text{list}[\text{Byte}]):$ $\text{size} = \text{length}(\text{bl}) \wedge$ $\text{plain_memory?}(pm) \wedge$ $pm' \text{states}(s) \wedge$ $(\text{address_block}(a, \text{size}) \subseteq pm' \text{rw_addr}) \wedge$ $\text{OK?}(\text{memory_write_list}(pm' \text{mem})(a, \text{bl})(s))$ \supset $\text{OK?}(\text{memory_read_list}(pm' \text{mem})(a, \text{size})$ $(\text{state}(\text{memory_write_list}(pm' \text{mem})(a, \text{bl})(s))))$

Repeatedly Skolemizing and flattening,
Using lemma memory_read_list_ok[State],
Using lemma plain_memory_transformer_invariant_read_list_block,
Replacing using formula -4,
Applying subset_bigger_union_right where a gets address_block(a!1, size!1), b gets pm!1'ro_addr, c gets pm!1'rw_addr,
Replacing using formula -7,
Replacing using formula -1,
Replacing using formula -2,
Using lemma plain_memory_transformers_ok_read_list_block,
Replacing using formula -6,
Replacing using formula -2,
Replacing using formula -1,
Using lemma plain_mem_write_list_states,
Replacing using formula -6,
Replacing using formula -7,
Replacing using formula -8,
Replacing using formula -9,
Replacing using formula -10,
Replacing using formula -1,
which is trivially true.
This completes the proof of plain_mem_write_list_read_list_ok.
Q.E.D.

C.192.4 Read_After_Write_Fails.Fatal_Result

Terse proof for Fatal_Result.

Fatal_Result:

$\{1\} \quad \forall (a: \text{Address}, pm: \text{Plain_Memory}[\text{State}], s: \text{State}):$ $\text{plain_memory?}(pm) \wedge pm' \text{states}(s) \wedge \text{in_blessed_memory?}(\text{pod}_1, a, pm' \text{rw_addr}) \supset$ $\text{Fatal?}((\text{write_data}(pm, \text{pod}_1)(a, \text{unit}) \text{ ## } \text{read_data}(pm, \text{pod}_2)(a))(s))$
--

Repeatedly Skolemizing and flattening,
Installing automatic rewrites from: write_data in_blessed_memory? pod_1 pod_2 length read_data uidt_1 uidt_2 ##
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Using lemma memory_write_list_ok[State],
Using lemma plain_memory_transformer_invariant_write_list_block,
Using lemma plain_memory_transformers_ok_write_list_block,

Using lemma plain_memory_side_effect_content_unchanged_write_block,
 Replacing using formula -5,
 Replacing using formula -6,
 Expanding the definition of length,
 Expanding the definition of length,
 Replacing using formula -1,
 Replacing using formula -2,
 Replacing using formula -3,
 Using lemma plain_mem_write_list_read_list_ok,
 Using lemma plain_mem_write_list_read_list,
 Expanding the definition of ok_lift,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 Replacing using formula -6,
 Keeping (-10) and hiding *,
 Using lemma list_cons_eta,
 Using lemma list_cons_eta,
 Replacing using formula -2,
 Replacing using formula -1,
 Keeping (-3) and hiding *,
 Applying decompose-equality,
 This completes the proof of Fatal_Result.
 Q.E.D.

C.193 Proofs for Read_After_Write_Fails_2 (datatype_model.pvs)

C.193.1

Read_After_Write_Fails_2.plain_mem_write_list_states_TCC1

Terse proof for plain_mem_write_list_states_TCC1.

plain_mem_write_list_states_TCC1:

$\{1\} \quad \forall (a: \text{Address}, pm: \text{Plain_Memory}[\text{State}], s: \text{State}, \text{size}: \text{nat}, bl: \text{list}[\text{Byte}]):$ $\text{size} = \text{length}(bl) \wedge$ $\text{plain_memory?}(pm) \wedge$ $pm' \text{states}(s) \wedge$ $(\text{address_block}(a, \text{size}) \subseteq pm' \text{rw_addr}) \wedge$ $\text{OK?}(\text{memory_write_list}(pm' \text{mem})(a, bl)(s))$ \supset $\text{OK?}[\text{State}, \text{Unit}](\text{memory_write_list}[\text{State}](pm' \text{mem})(a, bl)(s)) \vee$ $\text{Exception?}[\text{State}, \text{Unit}](\text{memory_write_list}[\text{State}](pm' \text{mem})(a, bl)(s))$
--

Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of plain_mem_write_list_states_TCC1.
 Q.E.D.

C.193.2 Read_After_Write_Fails_2.plain_mem_write_list_states

Terse proof for plain_mem_write_list_states.

C Proof scripts

plain_mem_write_list_states:

{1}	$\begin{aligned} &\forall (a: \text{Address}, pm: \text{Plain_Memory}[\text{State}], s: \text{State}, \text{size}: \text{nat}, bl: \text{list}[\text{Byte}]): \\ &\text{size} = \text{length}(bl) \wedge \\ &\text{plain_memory?}(pm) \wedge \\ &pm' \text{states}(s) \wedge \\ &(\text{address_block}(a, \text{size}) \subseteq pm' \text{rw_addr}) \wedge \\ &\text{OK?}(\text{memory_write_list}(pm' \text{mem})(a, bl)(s)) \\ &\supset pm' \text{states}(\text{state}(\text{memory_write_list}(pm' \text{mem})(a, bl)(s))) \end{aligned}$
-----	---

Repeatedly Skolemizing and flattening,

Using lemma transformer_invariant_write_list[State],

Replacing using formula -4,

Using lemma plain_memory_side_effect_content_unchanged_write_block,

Replacing using formula -4,

Replacing using formula -3,

Replacing using formula -6,

Replacing using formula -1,

Rewriting using add_as_union, matching in *,

Rewriting using transformer_invariant_union_transformers, matching in *,

we get 3 subgoals:

plain_mem_write_list_states.1:

{-1}	side_effect_content_unchanged(address_block(a', length(bl')), pm' states, memory_write_side_effect(pm' mem))
{-2}	result_pred(pm' states)(expr_2_super(memory_write_list(pm' mem)(a', bl'))(s'))
{-3}	size' = length(bl')
{-4}	plain_memory?(pm')
{-5}	pm' states(s')
{-6}	(address_block(a', length(bl')) \subseteq pm' rw_addr)
{-7}	OK?(memory_write_list(pm' mem)(a', bl')(s'))
{1}	pm' states(state(memory_write_list(pm' mem)(a', bl')(s')))

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of plain_mem_write_list_states.1.

plain_mem_write_list_states.2:

{-1}	side_effect_content_unchanged(address_block(a', length(bl')), pm' states, memory_write_side_effect(pm' mem))
{-2}	transformer_invariant?(pm' states, (memory_write_transformers(pm' mem, address_block(a', length(bl')) \supset result_pred(pm' states)(expr_2_super(memory_write_list(pm' mem)(a', bl'))(s')))
{-3}	size' = length(bl')
{-4}	plain_memory?(pm')
{-5}	pm' states(s')
{-6}	(address_block(a', length(bl')) \subseteq pm' rw_addr)
{-7}	OK?(memory_write_list(pm' mem)(a', bl')(s'))
{1}	transformer_invariant?(pm' states, memory_write_transformers(pm' mem, address_block(a', length(bl'))))
{2}	pm' states(state(memory_write_list(pm' mem)(a', bl')(s')))

Using lemma plain_memory_invariant_write_block,

Replacing using formula -5,

Replacing using formula -7,

which is trivially true.

This completes the proof of plain_mem_write_list_states.2.

plain_mem_write_list_states.3:

{-1}	side_effect_content_unchanged(address_block(a', length(bl')), pm' 'states, memory_write_side_effect(pm' 'mem))
{-2}	transformer_invariant?(pm' 'states, (memory_write_transformers(pm' 'mem, address_block(a', length(bl'))) ∪ singleton ⊃ result_pred(pm' 'states)(expr_2_super(memory_write_list(pm' 'mem)(a', bl'))(s'))
{-3}	size' = length(bl')
{-4}	plain_memory?(pm')
{-5}	pm' 'states(s')
{-6}	(address_block(a', length(bl'))) ⊆ pm' 'rw_addr)
{-7}	OK?(memory_write_list(pm' 'mem)(a', bl')(s'))
{1}	transformer_invariant?(pm' 'states, singleton(expr_2_super(memory_write_side_effect(pm' 'mem) (a', bl', FALSE))))
{2}	pm' 'states(state(memory_write_list(pm' 'mem)(a', bl')(s')))

Using lemma plain_memory_transformer_invariant_write_side_effects,

Replacing using formula -5,

Replacing using formula -7,

Applying transformer_invariant_mono_transformers where transformers_1 gets singleton(expr_2_super(memory_write_side_effect(pm!1, bl!1, FALSE))), transformers_2 gets memory_write_side_effect_super_transformers(pm!1'mem, address_block(a!1, length(bl!1))),

Instantiating quantified variables,

Replacing using formula -2,

Hiding formulas: -2, -3, -4, 2,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Hiding formulas: 2,

Expanding the definition of subset?,

Expanding the definition of memory_write_side_effect_super_transformers,

Expanding the definition of expr_2_super,

Expanding the definition of singleton,

Expanding the definition of member,

Repeatedly Skolemizing and flattening,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of plain_mem_write_list_states.3.

Q.E.D.

C.193.3 Read_After_Write_Fails_2.plain_mem_write_list_read_list_ok

Terse proof for plain_mem_write_list_read_list_ok.

plain_mem_write_list_read_list_ok:

$\{1\} \quad \forall (a: \text{Address}, pm: \text{Plain_Memory}[\text{State}], s: \text{State}, size: \text{nat}, bl: \text{list}[\text{Byte}]):$ $size = \text{length}(bl) \wedge$ $\text{plain_memory?}(pm) \wedge$ $pm' \text{states}(s) \wedge$ $(\text{address_block}(a, size) \subseteq pm' \text{rw_addr}) \wedge$ $\text{OK?}(\text{memory_write_list}(pm' \text{mem})(a, bl)(s))$ \supset $\text{OK?}(\text{memory_read_list}(pm' \text{mem})(a, size)$ $(\text{state}(\text{memory_write_list}(pm' \text{mem})(a, bl)(s))))$

Repeatedly Skolemizing and flattening,
Using lemma memory_read_list_ok[State],
Using lemma plain_memory_transformer_invariant_read_list_block,
Replacing using formula -4,
Applying subset_bigger_union_right where a gets address_block(a!1, size!1), b gets pm!1'ro_addr, c gets pm!1'rw_addr,
Replacing using formula -7,
Replacing using formula -1,
Replacing using formula -2,
Using lemma plain_memory_transformers_ok_read_list_block,
Replacing using formula -6,
Replacing using formula -2,
Replacing using formula -1,
Using lemma plain_mem_write_list_states,
Replacing using formula -6,
Replacing using formula -7,
Replacing using formula -8,
Replacing using formula -9,
Replacing using formula -10,
Replacing using formula -1,
which is trivially true.
This completes the proof of plain_mem_write_list_read_list_ok.
Q.E.D.

C.193.4 Read_After_Write_Fails_2.Fatal_Result

Terse proof for Fatal_Result.

Fatal_Result:

$\{1\} \quad \forall (a: \text{Address}, pm: \text{Plain_Memory}[\text{State}], s: \text{State}):$ $\text{plain_memory?}(pm) \wedge pm' \text{states}(s) \wedge \text{in_blessed_memory?}(\text{pod_bool}, a, pm' \text{rw_addr}) \supset$ $\text{Fatal?}((\text{write_data}(pm, \text{pod_bool})(a, \text{TRUE}) \#\# \text{read_data}(pm, \text{pod_Byte})(a))(s))$

Repeatedly Skolemizing and flattening,
Expanding the definition of write_data,
Expanding the definition of write_data,
Expanding the definition of in_blessed_memory?,
Expanding the definition of pod_bool,
Expanding the definition of ##,
Expanding the definition of ##,

Using lemma `memory_write_list_ok`[State],
Using lemma `plain_memory_transformer_invariant_write_list_block`,
Expanding the definition of `uidt_bool`,
Expanding the definition of `length`,
Expanding the definition of `length`,
Expanding the definition of `length`,
Replacing using formula -3,
Replacing using formula -5,
Replacing using formula -1,
Using lemma `plain_memory_transformers_ok_write_list_block`,
Replacing using formula -4,
Replacing using formula -6,
Replacing using formula -1,
Using lemma `plain_memory_side_effect_content_unchanged_write_block`,
Replacing using formula -5,
Replacing using formula -7,
Replacing using formula -1,
Replacing using formula -6,
Simplifying, rewriting, and recording with decision procedures,
Expanding the definition of `read_data`,
Expanding the definition of `read_data`,
Expanding the definition of `pod_Byte`,
Expanding the definition of `uidt_Byte`,
Expanding the definition of `##`,
Using lemma `plain_mem_write_list_read_list_ok`,
Expanding the definition of `length`,
Expanding the definition of `length`,
Expanding the definition of `length`,
Simplifying, rewriting, and recording with decision procedures,
Using lemma `plain_mem_write_list_read_list`,
Expanding the definition of `length`,
Expanding the definition of `length`,
Expanding the definition of `length`,
Replacing using formula -1,
Expanding the definition of `ok_lift`,
which is trivially true.
This completes the proof of `Fatal_Result`.
Q.E.D.

C.194 Proofs for Register_Id (constants.pvs)

This theory contains no provable formal statements.

C.195 Proofs for Register_Id_adt (Register_Id_adt.pvs)

This theory contains no provable formal statements.

C.196 Proofs for Register_Id_adt_reduce (Register_Id_adt.pvs)

This theory contains no provable formal statements.

C.197 Proofs for Rewrite_Test (plain_memory_rewrites.pvs)

C.197.1 Rewrite_Test.rewrite_test

Terse proof for `rewrite_test`.

`rewrite_test:`

<pre> {1} ∃ (addr1, addr2: Address, data1: Data1, data2: Data2, dt1: (interpreted_data_type?[Data1] dt2: (interpreted_data_type?[Data2]), pm: Plain_Memory[State], s: State): plain_memory?(pm) ∧ in_blessed_memory?(dt1, addr1, pm'rw_addr) ∧ in_blessed_memory?(dt2, addr2, pm'rw_addr) ∧ blocks_disjoint?(addr1, size(uidt(dt1)), addr2, size(uidt(dt2))) ∧ blocks_disjoint?(addr2, size(uidt(dt2)), addr1, size(uidt(dt1))) ∧ pm'states(s) ⊃ OK?((write_data(pm, dt1)(addr1, data1) ## write_data(pm, dt2)(addr2, data2) ## read_data(pm, dt1)(addr1) ## write_data(pm, dt2)(addr2, data2) ## read_data(pm, dt2)(addr2) ## read_data(pm, dt1)(addr1) (s) </pre>
--

Installing automatic rewrites from: `plain_memory_write_ok_single plain_memory_write_ok_q_expr plain_memory_write_ok_q_stmt in_blessed_memory_rw_ro pm_q_prop_ok pm_q_prop_single_read pm_q_prop_single_write pm_q_prop_read_write_single pm_q_prop_read_write_other_single pm_q_prop_read_read_single plain_memory_read_write_other_single_data plain_memory_read_read plain_memory_read_write_res pm_q_prop_read_ok_expr pm_q_prop_read_ok_stmt pm_q_prop_read_pm_q_prop_expr pm_q_prop_read_pm_q_prop_stmt pm_q_prop_read_write_q_expr pm_q_prop_read_write_q_stmt plain_memory_read_write_q_data_expr plain_memory_read_write_q_data_stmt pm_q_prop_read_write_other_q_expr pm_q_prop_read_write_other_q_stmt pm_q_prop_read_read_q_expr pm_q_prop_read_read_q_stmt plain_memory_read_write_other_q_data_expr plain_memory_read_write_other_q_data_stmt plain_memory_read_read_q_data_expr plain_memory_read_read_q_data_stmt pm_q_prop_ok_result_single pm_q_prop_ok_result_q_expr pm_q_prop_ok_result_q_stmt pm_q_prop_read_ok_result_single pm_q_prop_read_ok_result_q_expr pm_q_prop_read_ok_result_q_stmt pm_q_prop_e2s pm_q_prop_stmt_e2s pm_q_prop_read_e2s pm_q_prop_read_stmt_e2s`

Repeatedly Skolemizing and flattening,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `rewrite_test`.

Q.E.D.

C.197.2 Rewrite_Test.rewrite_test_data_TCC1

Terse proof for `rewrite_test_data_TCC1`.

rewrite_test_data_TCC1:

```

{1}  ∃ (addr1, addr2: Address, data1: Data1, data2: Data2, dt1: (interpreted_data_type?[Data1]),
      dt2: (interpreted_data_type?[Data2]), pm: Plain_Memory[State], s: State):
  pm'states(s) ∧
  blocks_disjoint?(addr2, size(uidt(dt2)), addr1, size(uidt(dt1))) ∧
  blocks_disjoint?(addr1, size(uidt(dt1)), addr2, size(uidt(dt2))) ∧
  in_blessed_memory?(dt2, addr2, pm'rw_addr) ∧
  in_blessed_memory?(dt1, addr1, pm'rw_addr) ∧ plain_memory?(pm)
  ⊃
  OK?[State, Data1]
    ((##[State, Data2, Data1]
      (##[State, Unit, Data2]
        (##[State, Data1, Unit]
          (##[State, Unit, Data1]
            (##[State, Unit, Unit]
              (write_data[State, Data1](pm, dt1)(addr1, data1),
                write_data[State, Data2](pm, dt2)(addr2, data2)),
                read_data[State, Data1](pm, dt1)(addr1)),
                write_data[State, Data2](pm, dt2)(addr2, data2)),
                read_data[State, Data2](pm, dt2)(addr2)),
                read_data[State, Data1](pm, dt1)(addr1)))
            (s))
    )

```

Repeatedly Skolemizing and flattening,
 Rewriting using rewrite_test, matching in *,
 This completes the proof of rewrite_test_data_TCC1.
 Q.E.D.

C.197.3 Rewrite_Test.rewrite_test_data

Terse proof for rewrite_test_data.

rewrite_test_data:

```

{1}  ∃ (addr1, addr2: Address, data1: Data1, data2: Data2, dt1: (interpreted_data_type?[Data1]),
      dt2: (interpreted_data_type?[Data2]), pm: Plain_Memory[State], s: State):
  plain_memory?(pm) ∧
  in_blessed_memory?(dt1, addr1, pm'rw_addr) ∧
  in_blessed_memory?(dt2, addr2, pm'rw_addr) ∧
  blocks_disjoint?(addr1, size(uidt(dt1)), addr2, size(uidt(dt2))) ∧
  blocks_disjoint?(addr2, size(uidt(dt2)), addr1, size(uidt(dt1))) ∧ pm'states(s)
  ⊃
  data((write_data(pm, dt1)(addr1, data1) ## write_data(pm, dt2)(addr2, data2) ##
        read_data(pm, dt1)(addr1)
        ## write_data(pm, dt2)(addr2, data2)
        ## read_data(pm, dt2)(addr2)
        ## read_data(pm, dt1)(addr1))
        (s))
  = data1

```

Installing automatic rewrites from: plain_memory_write_ok_single plain_memory_write_ok_q_expr

C Proof scripts

plain_memory_write_ok_q_stmt in_blessed_memory_rw_ro pm_q_prop_ok pm_q_prop_single_read
pm_q_prop_single_write pm_q_prop_read_write_single pm_q_prop_read_write_other_single pm_q_prop_read_read_sing
plain_memory_read_write_other_single_data plain_memory_read_read plain_memory_read_write_res
pm_q_prop_read_ok_expr pm_q_prop_read_ok_stmt pm_q_prop_read_pm_q_prop_expr pm_q_prop_read_pm_q_prop_s
pm_q_prop_read_write_q_expr pm_q_prop_read_write_q_stmt plain_memory_read_write_q_data_expr
plain_memory_read_write_q_data_stmt pm_q_prop_read_write_other_q_expr pm_q_prop_read_write_other_q_stmt
pm_q_prop_read_read_q_expr pm_q_prop_read_read_q_stmt plain_memory_read_write_other_q_data_expr
plain_memory_read_write_other_q_data_stmt plain_memory_read_read_q_data_expr plain_memory_read_read_q_data
pm_q_prop_ok_result_single pm_q_prop_ok_result_q_expr pm_q_prop_ok_result_q_stmt pm_q_prop_read_ok_result_sir
pm_q_prop_read_ok_result_q_expr pm_q_prop_read_ok_result_q_stmt pm_q_prop_e2s pm_q_prop_stmt_e2s
pm_q_prop_read_e2s pm_q_prop_read_stmt_e2s

Repeatedly Skolemizing and flattening,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `rewrite_test_data`.

Q.E.D.

C.197.4 Rewrite_Test.rewrite_test_data_long_TCC1

Terse proof for `rewrite_test_data_long_TCC1`.

C Proof scripts

Repeatedly Skolemizing and flattening,

Installing automatic rewrites from: plain_memory_write_ok_single plain_memory_write_ok_q_expr
plain_memory_write_ok_q_stmt in_blessed_memory_rw_ro pm_q_prop_ok pm_q_prop_single_read
pm_q_prop_single_write pm_q_prop_read_write_single pm_q_prop_read_write_other_single pm_q_prop_read_read_sing
plain_memory_read_write_other_single_data plain_memory_read_read plain_memory_read_write_res
pm_q_prop_read_ok_expr pm_q_prop_read_ok_stmt pm_q_prop_read_pm_q_prop_expr pm_q_prop_read_pm_q_prop_st
pm_q_prop_read_write_q_expr pm_q_prop_read_write_q_stmt plain_memory_read_write_q_data_expr
plain_memory_read_write_q_data_stmt pm_q_prop_read_write_other_q_expr pm_q_prop_read_write_other_q_stmt
pm_q_prop_read_read_q_expr pm_q_prop_read_read_q_stmt plain_memory_read_write_other_q_data_expr
plain_memory_read_write_other_q_data_stmt plain_memory_read_read_q_data_expr plain_memory_read_read_q_data
pm_q_prop_ok_result_single pm_q_prop_ok_result_q_expr pm_q_prop_ok_result_q_stmt pm_q_prop_read_ok_result_sir
pm_q_prop_read_ok_result_q_expr pm_q_prop_read_ok_result_q_stmt pm_q_prop_e2s pm_q_prop_stmt_e2s
pm_q_prop_read_e2s pm_q_prop_read_stmt_e2s

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `rewrite_test_data_long_TCC1`.

Q.E.D.

C.197.5 Rewrite_Test.rewrite_test_data_long

Terse proof for `rewrite_test_data_long`.

rewrite_test_data_long:

```

{1}  ∃ (addr1, addr2: Address, data1: Data1, data2: Data2, dt1: (interpreted_data_type?[Data1]),
      dt2: (interpreted_data_type?[Data2]), pm: Plain_Memory[State], s: State):
  plain_memory?(pm) ∧
  in_blessed_memory?(dt1, addr1, pm'rw_addr) ∧
  in_blessed_memory?(dt2, addr2, pm'rw_addr) ∧
  blocks_disjoint?(addr1, size(uidt(dt1)), addr2, size(uidt(dt2))) ∧
  blocks_disjoint?(addr2, size(uidt(dt2)), addr1, size(uidt(dt1))) ∧ pm'states(s)
  ⊃
  data((write_data(pm, dt1)(addr1, data1) ## write_data(pm, dt2)(addr2, data2) ##
      read_data(pm, dt1)(addr1)
      ## write_data(pm, dt2)(addr2, data2)
      ## read_data(pm, dt2)(addr2)
      ## write_data(pm, dt2)(addr2, data2)
      ## read_data(pm, dt1)(addr1)
      ## write_data(pm, dt2)(addr2, data2)
      ## read_data(pm, dt2)(addr2)
      ## write_data(pm, dt2)(addr2, data2)
      ## read_data(pm, dt1)(addr1)
      ## write_data(pm, dt2)(addr2, data2)
      ## read_data(pm, dt2)(addr2)
      ## write_data(pm, dt2)(addr2, data2)
      ## read_data(pm, dt1)(addr1)
      ## write_data(pm, dt2)(addr2, data2)
      ## read_data(pm, dt2)(addr2)
      ## write_data(pm, dt2)(addr2, data2)
      ## read_data(pm, dt1)(addr1)
      ## write_data(pm, dt2)(addr2, data2)
      ## read_data(pm, dt2)(addr2)
      ## write_data(pm, dt2)(addr2, data2)
      ## read_data(pm, dt1)(addr1)
      ## write_data(pm, dt2)(addr2, data2)
      ## read_data(pm, dt2)(addr2)
      ## write_data(pm, dt2)(addr2, data2)
      ## read_data(pm, dt1)(addr1)
      ## write_data(pm, dt2)(addr2, data2)
      ## read_data(pm, dt2)(addr2)
      ## write_data(pm, dt2)(addr2, data2)
      ## read_data(pm, dt1)(addr1)
      ## write_data(pm, dt2)(addr2, data2)
      ## read_data(pm, dt2)(addr2)
      ## read_data(pm, dt1)(addr1))
      (s))
  = data1

```

Installing automatic rewrites from: plain_memory_write_ok_single plain_memory_write_ok_q_expr
plain_memory_write_ok_q_stmt in_blessed_memory_rw_ro pm_q_prop_ok pm_q_prop_single_read
pm_q_prop_single_write pm_q_prop_read_write_single pm_q_prop_read_write_other_single pm_q_prop_read_read_single
plain_memory_read_write_other_single_data plain_memory_read_read plain_memory_read_write_res
pm_q_prop_read_ok_expr pm_q_prop_read_ok_stmt pm_q_prop_read_pm_q_prop_expr pm_q_prop_read_pm_q_prop_stmt
pm_q_prop_read_write_q_expr pm_q_prop_read_write_q_stmt plain_memory_read_write_q_data_expr

plain_memory_read_write_q_data_stmt pm_q_prop_read_write_other_q_expr pm_q_prop_read_write_other_q_stmt
 pm_q_prop_read_read_q_expr pm_q_prop_read_read_q_stmt plain_memory_read_write_other_q_data_expr
 plain_memory_read_write_other_q_data_stmt plain_memory_read_read_q_data_expr plain_memory_read_read_q_data_stmt
 pm_q_prop_ok_result_single pm_q_prop_ok_result_q_expr pm_q_prop_ok_result_q_stmt pm_q_prop_read_ok_result_single
 pm_q_prop_read_ok_result_q_expr pm_q_prop_read_ok_result_q_stmt pm_q_prop_e2s pm_q_prop_stmt_e2s
 pm_q_prop_read_e2s pm_q_prop_read_stmt_e2s

Repeatedly Skolemizing and flattening,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `rewrite_test_data_long`.

Q.E.D.

C.197.6 Rewrite_Test.rewrite_test_data_eval_if_ok_ok

Terse proof for `rewrite_test_data_eval_if_ok_ok`.

`rewrite_test_data_eval_if_ok_ok`:

{1}	\forall (addr1, addr2: Address, data1: Data1, data2: Data2, dt1: (interpreted_data_type?[Data1]) dt2: (interpreted_data_type?[Data2]), pm: Plain_Memory[State], s: State): plain_memory?(pm) \wedge in_blessed_memory?(dt1, addr1, pm'rw_addr) \wedge in_blessed_memory?(dt2, addr2, pm'rw_addr) \wedge blocks_disjoint?(addr1, size(uidt(dt1)), addr2, size(uidt(dt2))) \wedge blocks_disjoint?(addr2, size(uidt(dt2)), addr1, size(uidt(dt1))) \wedge pm'states(s) \supset OK?((write_data(pm, dt1)(addr1, data1) ## write_data(pm, dt2)(addr2, data2) ## read_data(pm, dt1)(addr1) ## write_data(pm, dt2)(addr2, data2) ## read_data(pm, dt2)(addr2) ## (λ (d: Data2): read_data(pm, dt1)(addr1))) (s))
-----	--

Installing automatic rewrites from: plain_memory_write_ok_single plain_memory_write_ok_q_expr
 plain_memory_write_ok_q_stmt in_blessed_memory_rw_ro pm_q_prop_ok pm_q_prop_single_read
 pm_q_prop_single_write pm_q_prop_read_write_single pm_q_prop_read_write_other_single pm_q_prop_read_read_single
 plain_memory_read_write_other_single_data plain_memory_read_read plain_memory_read_write_res
 pm_q_prop_read_ok_expr pm_q_prop_read_ok_stmt pm_q_prop_read_pm_q_prop_expr pm_q_prop_read_pm_q_prop_stmt
 pm_q_prop_read_write_q_expr pm_q_prop_read_write_q_stmt plain_memory_read_write_q_data_expr
 plain_memory_read_write_q_data_stmt pm_q_prop_read_write_other_q_expr pm_q_prop_read_write_other_q_stmt
 pm_q_prop_read_read_q_expr pm_q_prop_read_read_q_stmt plain_memory_read_write_other_q_data_expr
 plain_memory_read_write_other_q_data_stmt plain_memory_read_read_q_data_expr plain_memory_read_read_q_data_stmt
 pm_q_prop_ok_result_single pm_q_prop_ok_result_q_expr pm_q_prop_ok_result_q_stmt pm_q_prop_read_ok_result_single
 pm_q_prop_read_ok_result_q_expr pm_q_prop_read_ok_result_q_stmt pm_q_prop_e2s pm_q_prop_stmt_e2s
 pm_q_prop_read_e2s pm_q_prop_read_stmt_e2s

Repeatedly Skolemizing and flattening,

Installing automatic rewrites from: comp_eval_if_ok_fexpr ok_result_q_ok ok_result_ok
 ok_result_q_data different_registers_blocks_disjoint

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `rewrite_test_data_eval_if_ok_ok`.

Q.E.D.

C.197.7 Rewrite_Test.rewrite_test_data_eval_if_ok_TCC1

Terse proof for rewrite_test_data_eval_if_ok_TCC1.

rewrite_test_data_eval_if_ok_TCC1:

```

{1}  ∇ (addr1, addr2: Address, data1: Data1, data2: Data2, dt1: (interpreted_data_type?[Data1]),
      dt2: (interpreted_data_type?[Data2]), pm: Plain_Memory[State], s: State):
  pm'states(s) ∧
  blocks_disjoint?(addr2, size(uidt(dt2)), addr1, size(uidt(dt1))) ∧
  blocks_disjoint?(addr1, size(uidt(dt1)), addr2, size(uidt(dt2))) ∧
  in_blessed_memory?(dt2, addr2, pm'rw_addr) ∧
  in_blessed_memory?(dt1, addr1, pm'rw_addr) ∧ plain_memory?(pm)
⊃
OK?[State, Data1]
  ((##[State, Data2, Data1]
    (##[State, Unit, Data2]
      (##[State, Data1, Unit]
        (##[State, Unit, Data1]
          (##[State, Unit, Unit]
            (write_data[State, Data1](pm, dt1)(addr1, data1),
              write_data[State, Data2](pm, dt2)(addr2, data2)),
            read_data[State, Data1](pm, dt1)(addr1),
            write_data[State, Data2](pm, dt2)(addr2, data2)),
            read_data[State, Data2](pm, dt2)(addr2)),
          λ (d: Data2): read_data[State, Data1](pm, dt1)(addr1)))
      (s))
  )

```

Repeatedly Skolemizing and flattening,

Rewriting using rewrite_test_data_eval_if_ok_ok, matching in *,

This completes the proof of rewrite_test_data_eval_if_ok_TCC1.

Q.E.D.

C.197.8 Rewrite_Test.rewrite_test_data_eval_if_ok

Terse proof for rewrite_test_data_eval_if_ok.

rewrite_test_data_eval_if_ok:

```

{1}  ∇ (addr1, addr2: Address, data1: Data1, data2: Data2, dt1: (interpreted_data_type?[Data1]),
      dt2: (interpreted_data_type?[Data2]), pm: Plain_Memory[State], s: State):
  plain_memory?(pm) ∧
  in_blessed_memory?(dt1, addr1, pm'rw_addr) ∧
  in_blessed_memory?(dt2, addr2, pm'rw_addr) ∧
  blocks_disjoint?(addr1, size(uidt(dt1)), addr2, size(uidt(dt2))) ∧
  blocks_disjoint?(addr2, size(uidt(dt2)), addr1, size(uidt(dt1))) ∧ pm'states(s)
⊃
data((write_data(pm, dt1)(addr1, data1) ## write_data(pm, dt2)(addr2, data2) ##
      read_data(pm, dt1)(addr1)
      ## write_data(pm, dt2)(addr2, data2)
      ## read_data(pm, dt2)(addr2)
      ## (λ (d: Data2): read_data(pm, dt1)(addr1)))
      (s))
  = data1

```

Installing automatic rewrites from: plain_memory_write_ok_single plain_memory_write_ok_q_expr plain_memory_write_ok_q_stmt in_blessed_memory_rw_ro pm_q_prop_ok pm_q_prop_single_read pm_q_prop_single_write pm_q_prop_read_write_single pm_q_prop_read_write_other_single pm_q_prop_read_read_single plain_memory_read_write_other_single_data plain_memory_read_read plain_memory_read_write_res pm_q_prop_read_ok_expr pm_q_prop_read_ok_stmt pm_q_prop_read_pm_q_prop_expr pm_q_prop_read_pm_q_prop_stmt pm_q_prop_read_write_q_expr pm_q_prop_read_write_q_stmt plain_memory_read_write_q_data_expr plain_memory_read_write_q_data_stmt pm_q_prop_read_write_other_q_expr pm_q_prop_read_write_other_q_stmt pm_q_prop_read_read_q_expr pm_q_prop_read_read_q_stmt plain_memory_read_write_other_q_data_expr plain_memory_read_write_other_q_data_stmt plain_memory_read_read_q_data_expr plain_memory_read_read_q_data_stmt pm_q_prop_ok_result_single pm_q_prop_ok_result_q_expr pm_q_prop_ok_result_q_stmt pm_q_prop_read_ok_result_single pm_q_prop_read_ok_result_q_expr pm_q_prop_read_ok_result_q_stmt pm_q_prop_e2s pm_q_prop_stmt_e2s pm_q_prop_read_e2s pm_q_prop_read_stmt_e2s

Repeatedly Skolemizing and flattening,

Installing automatic rewrites from: comp_eval_if_ok_fexpr ok_result_q_ok ok_result_ok ok_result_q_data different_registers_blocks_disjoint

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `rewrite_test_data_eval_if_ok`.

Q.E.D.

C.197.9 Rewrite_Test.rewrite_test_data_eval_if_ok_ok_result_ok

Terse proof for `rewrite_test_data_eval_if_ok_ok_result_ok`.

`rewrite_test_data_eval_if_ok_ok_result_ok`:

<pre>{1} ∃ (addr1, addr2: Address, data1: Data1, data2: Data2, dt1: (interpreted_data_type? [Data1] dt2: (interpreted_data_type? [Data2]), pm: Plain_Memory [State], s: State): plain_memory?(pm) ∧ in_blessed_memory?(dt1, addr1, pm'rw_addr) ∧ in_blessed_memory?(dt2, addr2, pm'rw_addr) ∧ blocks_disjoint?(addr1, size(uidt(dt1)), addr2, size(uidt(dt2))) ∧ blocks_disjoint?(addr2, size(uidt(dt2)), addr1, size(uidt(dt1))) ∧ pm'states(s) ⊃ OK?[State, Data2] ((write_data(pm, dt1)(addr1, data1) ## write_data(pm, dt2)(addr2, data2) ## read_data(pm, dt1)(addr1) ## write_data(pm, dt2)(addr2, data2) ## read_data(pm, dt2)(addr2) ## (λ (d: Data2): ok_result(d)) (s))</pre>
--

Installing automatic rewrites from: plain_memory_write_ok_single plain_memory_write_ok_q_expr plain_memory_write_ok_q_stmt in_blessed_memory_rw_ro pm_q_prop_ok pm_q_prop_single_read pm_q_prop_single_write pm_q_prop_read_write_single pm_q_prop_read_write_other_single pm_q_prop_read_read_single plain_memory_read_write_other_single_data plain_memory_read_read plain_memory_read_write_res pm_q_prop_read_ok_expr pm_q_prop_read_ok_stmt pm_q_prop_read_pm_q_prop_expr pm_q_prop_read_pm_q_prop_stmt pm_q_prop_read_write_q_expr pm_q_prop_read_write_q_stmt plain_memory_read_write_q_data_expr plain_memory_read_write_q_data_stmt pm_q_prop_read_write_other_q_expr pm_q_prop_read_write_other_q_stmt pm_q_prop_read_read_q_expr pm_q_prop_read_read_q_stmt plain_memory_read_write_other_q_data_expr plain_memory_read_write_other_q_data_stmt plain_memory_read_read_q_data_expr plain_memory_read_read_q_data_stmt pm_q_prop_ok_result_single pm_q_prop_ok_result_q_expr pm_q_prop_ok_result_q_stmt pm_q_prop_read_ok_result_single pm_q_prop_read_ok_result_q_expr pm_q_prop_read_ok_result_q_stmt pm_q_prop_e2s pm_q_prop_stmt_e2s pm_q_prop_read_e2s pm_q_prop_read_stmt_e2s

Repeatedly Skolemizing and flattening,

Installing automatic rewrites from: comp_eval_if_ok_fexpr ok_result_q_ok ok_result_ok
ok_result_q_data different_registers_blocks_disjoint

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of rewrite_test_data_eval_if_ok_ok_result_ok.

Q.E.D.

C.197.10 Rewrite_Test.rewrite_test_data_eval_if_ok_ok_result_TCC1

Terse proof for rewrite_test_data_eval_if_ok_ok_result_TCC1.

rewrite_test_data_eval_if_ok_ok_result_TCC1:

<pre> {1} ∃ (addr1, addr2: Address, data1: Data1, data2: Data2, dt1: (interpreted_data_type?[Data1]), dt2: (interpreted_data_type?[Data2]), pm: Plain_Memory[State], s: State): pm'states(s) ∧ blocks_disjoint?(addr2, size(uidt(dt2)), addr1, size(uidt(dt1))) ∧ blocks_disjoint?(addr1, size(uidt(dt1)), addr2, size(uidt(dt2))) ∧ in_blessed_memory?(dt2, addr2, pm'rw_addr) ∧ in_blessed_memory?(dt1, addr1, pm'rw_addr) ∧ plain_memory?(pm) ⊃ OK?[State, Data2] ((##[State, Data2, Data2] (##[State, Unit, Data2] (##[State, Data1, Unit] (##[State, Unit, Data1] (##[State, Unit, Unit] (write_data[State, Data1](pm, dt1)(addr1, data1), write_data[State, Data2](pm, dt2)(addr2, data2)), read_data[State, Data1](pm, dt1)(addr1), write_data[State, Data2](pm, dt2)(addr2, data2)), read_data[State, Data2](pm, dt2)(addr2)), λ (d: Data2): ok_result[State, Data2](d))) (s)) </pre>
--

Repeatedly Skolemizing and flattening,

Rewriting using rewrite_test_data_eval_if_ok_ok_result_ok, matching in *,

This completes the proof of rewrite_test_data_eval_if_ok_ok_result_TCC1.

Q.E.D.

C.197.11 Rewrite_Test.rewrite_test_data_eval_if_ok_ok_result

Terse proof for rewrite_test_data_eval_if_ok_ok_result.

rewrite_test_data_eval_if_ok_ok_result:

{1}	$\forall (\text{addr1}, \text{addr2}: \text{Address}, \text{data1}: \text{Data1}, \text{data2}: \text{Data2}, \text{dt1}: (\text{interpreted_data_type?}[\text{Data1}])$ $\text{dt2}: (\text{interpreted_data_type?}[\text{Data2}]), \text{pm}: \text{Plain_Memory}[\text{State}], \text{s}: \text{State}):$ $\text{plain_memory?}(\text{pm}) \wedge$ $\text{in_blessed_memory?}(\text{dt1}, \text{addr1}, \text{pm}'\text{rw_addr}) \wedge$ $\text{in_blessed_memory?}(\text{dt2}, \text{addr2}, \text{pm}'\text{rw_addr}) \wedge$ $\text{blocks_disjoint?}(\text{addr1}, \text{size}(\text{uidt}(\text{dt1})), \text{addr2}, \text{size}(\text{uidt}(\text{dt2}))) \wedge$ $\text{blocks_disjoint?}(\text{addr2}, \text{size}(\text{uidt}(\text{dt2})), \text{addr1}, \text{size}(\text{uidt}(\text{dt1}))) \wedge \text{pm}'\text{states}(\text{s})$ \supset $\text{data}((\text{write_data}(\text{pm}, \text{dt1})(\text{addr1}, \text{data1}) \text{ ## } \text{write_data}(\text{pm}, \text{dt2})(\text{addr2}, \text{data2}) \text{ ##}$ $\text{read_data}(\text{pm}, \text{dt1})(\text{addr1})$ $\text{ ## } \text{write_data}(\text{pm}, \text{dt2})(\text{addr2}, \text{data2})$ $\text{ ## } \text{read_data}(\text{pm}, \text{dt2})(\text{addr2})$ $\text{ ## } (\lambda (d: \text{Data2}): \text{ok_result}(d))$ $(\text{s}))$ $= \text{data2}$
-----	--

Installing automatic rewrites from: plain_memory_write_ok_single plain_memory_write_ok_q_expr
plain_memory_write_ok_q_stmt in_blessed_memory_rw_ro pm_q_prop_ok pm_q_prop_single_read
pm_q_prop_single_write pm_q_prop_read_write_single pm_q_prop_read_write_other_single pm_q_prop_read_read_sing
plain_memory_read_write_other_single_data plain_memory_read_read plain_memory_read_write_res
pm_q_prop_read_ok_expr pm_q_prop_read_ok_stmt pm_q_prop_read_pm_q_prop_expr pm_q_prop_read_pm_q_prop_st
pm_q_prop_read_write_q_expr pm_q_prop_read_write_q_stmt plain_memory_read_write_q_data_expr
plain_memory_read_write_q_data_stmt pm_q_prop_read_write_other_q_expr pm_q_prop_read_write_other_q_stmt
pm_q_prop_read_read_q_expr pm_q_prop_read_read_q_stmt plain_memory_read_write_other_q_data_expr
plain_memory_read_write_other_q_data_stmt plain_memory_read_read_q_data_expr plain_memory_read_read_q_data
pm_q_prop_ok_result_single pm_q_prop_ok_result_q_expr pm_q_prop_ok_result_q_stmt pm_q_prop_read_ok_result_sir
pm_q_prop_read_ok_result_q_expr pm_q_prop_read_ok_result_q_stmt pm_q_prop_e2s pm_q_prop_stmt_e2s
pm_q_prop_read_e2s pm_q_prop_read_stmt_e2s

Repeatedly Skolemizing and flattening,

Installing automatic rewrites from: comp_eval_if_ok_fexpr ok_result_q_ok ok_result_ok
ok_result_q_data different_registers_blocks_disjoint

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of rewrite_test_data_eval_if_ok_ok_result.

Q.E.D.

C.198 Proofs for Search_Example (search-example.pvs)

C.198.1 Search_Example.value_TCC1

Terse proof for value_TCC1.

value_TCC1:

{1}	$\exists (x: (\text{range}(\text{uint}))) : \text{TRUE}$
-----	--

Instantiating the top quantifier in 1 with the terms: (0),

we get 2 subgoals:

value_TCC1.1:

{1}	TRUE
-----	------

which is trivially true.

This completes the proof of value_TCC1.1.

value_TCC1.2:

{1} range(uint)(0)

Trying repeated skolemization, instantiation, and if-lifting,

Rewriting using binary_range_uint, matching in *,

Using lemma max_uint,

Using lemma max_uint_bits,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of value_TCC1.2.

Q.E.D.

C.198.2 Search_Example.Search_Pre_TCC1

Terse proof for Search_Pre_TCC1.

Search_Pre_TCC1:

{1} Cpp_Type?(uint) \wedge cv(non_bool_integral_enum?)(uint)

Applying propositional simplification,

we get 2 subgoals:

Search_Pre_TCC1.1:

{1} Cpp_Type?(uint)

Expanding the definition of Cpp_Type?,

Repeatedly Skolemizing and flattening,

Expanding the definition of subterm,

Installing automatic rewrites from: no_pointers_to_bitfield? no_pointers_to_references?
no_cv_references? no_reference_to_reference? no_reference_to_bitfields? no_pointer_to_member_to_reference?
no_pointer_to_member_to_cv_void? no_cv_void_parameter? no_array_of_references? no_array_of_cv_void?
no_array_of_function? no_array_of_abstract_class? no_pointer_or_ref_to_incomplete_array_parameter?
no_array_return_type? no_function_return_type? bitfield_underlying_integral_or_enum_type? cv_array?
enum_underlying_integral? enum_constants? const_volatile? const_stutter? volatile_stutter?
no_cv_class? no_cv_union? no_cv_function? no_reference_to_void? no_cv_void? no_array_of_bitfields?

Replacing using formula -1,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of Search_Pre_TCC1.1.

Search_Pre_TCC1.2:

{1} cv(non_bool_integral_enum?)(uint)

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of Search_Pre_TCC1.2.

Q.E.D.

C.198.3 Search_Example.Search_Pre_TCC2

Terse proof for Search_Pre_TCC2.

C Proof scripts

Search_Pre_TCC2:

$$\{1\} \quad \forall (s: \text{State}):$$
$$\text{pm}'\text{states}(s) \wedge \text{plain_memory}'(\text{pm}) \supset$$
$$\text{Cpp_Type}'(\text{pointer}(\text{uint})) \wedge \text{cv}(\text{pointer}')(\text{pointer}(\text{uint}))$$

Repeatedly Skolemizing and flattening,

Keeping (1) and hiding *,

Applying propositional simplification,

we get 2 subgoals:

Search_Pre_TCC2.1:

$$\{1\} \quad \text{Cpp_Type}'(\text{pointer}(\text{uint}))$$

Expanding the definition of Cpp_Type?,

Expanding the definition of subterm,

Repeatedly Skolemizing and flattening,

Expanding the definition of subterm,

Installing automatic rewrites from: no_pointers_to_bitfield? no_pointers_to_references?
no_cv_references? no_reference_to_reference? no_reference_to_bitfields? no_pointer_to_member_to_reference?
no_pointer_to_member_to_cv_void? no_cv_void_parameter? no_array_of_references? no_array_of_cv_void?
no_array_of_function? no_array_of_abstract_class? no_pointer_or_ref_to_incomplete_array_parameter?
no_array_return_type? no_function_return_type? bitfield_underlying_integral_or_enum_type? cv_array?
enum_underlying_integral? enum_constants? const_volatile? const_stutter? volatile_stutter?
no_cv_class? no_cv_union? no_cv_function? no_reference_to_void? no_cv_void? no_array_of_bitfields?

Splitting conjunctions,

we get 2 subgoals:

Search_Pre_TCC2.1.1.1:

{-1}	$t' = \text{pointer}(\text{uint})$
{1}	$\begin{aligned} & \text{no_pointers_to_bitfield?}(t') \wedge \\ & \text{no_pointers_to_references?}(t') \wedge \\ & \text{no_cv_references?}(t') \wedge \\ & \text{no_reference_to_reference?}(t') \wedge \\ & \text{no_reference_to_bitfields?}(t') \wedge \\ & \text{no_pointer_to_member_to_reference?}(t') \wedge \\ & \text{no_pointer_to_member_to_cv_void?}(t') \wedge \\ & \text{no_cv_void_parameter?}(t') \wedge \\ & \text{no_array_of_references?}(t') \wedge \\ & \text{no_array_of_cv_void?}(t') \wedge \\ & \text{no_array_of_function?}(t') \wedge \\ & \text{no_array_of_abstract_class?}(t') \wedge \\ & \text{no_pointer_or_ref_to_incomplete_array_parameter?}(t') \wedge \\ & \text{no_array_return_type?}(t') \wedge \\ & \text{no_function_return_type?}(t') \wedge \\ & \text{bitfield_underlying_integral_or_enum_type?}(t') \wedge \\ & \text{cv_array?}(t') \wedge \\ & \text{enum_underlying_integral?}(t') \wedge \\ & \text{enum_constants?}(t') \wedge \\ & \text{const_volatile?}(t') \wedge \\ & \text{const_stutter?}(t') \wedge \\ & \text{volatile_stutter?}(t') \wedge \\ & \text{no_cv_class?}(t') \wedge \\ & \text{no_cv_union?}(t') \wedge \\ & \text{no_cv_function?}(t') \wedge \\ & \text{no_reference_to_void?}(t') \wedge \\ & \text{no_cv_void?}(t') \wedge \text{no_array_of_bitfields?}(t') \end{aligned}$

Replacing using formula -1,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Keeping (-2) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of Search_Pre_TCC2.1.1.1.

Search_Pre_TCC2.1.2:

{-1}	$t' = \text{uint}$
{1}	$ \begin{aligned} & \text{no_pointers_to_bitfield?}(t') \wedge \\ & \text{no_pointers_to_references?}(t') \wedge \\ & \text{no_cv_references?}(t') \wedge \\ & \text{no_reference_to_reference?}(t') \wedge \\ & \text{no_reference_to_bitfields?}(t') \wedge \\ & \text{no_pointer_to_member_to_reference?}(t') \wedge \\ & \text{no_pointer_to_member_to_cv_void?}(t') \wedge \\ & \text{no_cv_void_parameter?}(t') \wedge \\ & \text{no_array_of_references?}(t') \wedge \\ & \text{no_array_of_cv_void?}(t') \wedge \\ & \text{no_array_of_function?}(t') \wedge \\ & \text{no_array_of_abstract_class?}(t') \wedge \\ & \text{no_pointer_or_ref_to_incomplete_array_parameter?}(t') \wedge \\ & \text{no_array_return_type?}(t') \wedge \\ & \text{no_function_return_type?}(t') \wedge \\ & \text{bitfield_underlying_integral_or_enum_type?}(t') \wedge \\ & \text{cv_array?}(t') \wedge \\ & \text{enum_underlying_integral?}(t') \wedge \\ & \text{enum_constants?}(t') \wedge \\ & \text{const_volatile?}(t') \wedge \\ & \text{const_stutter?}(t') \wedge \\ & \text{volatile_stutter?}(t') \wedge \\ & \text{no_cv_class?}(t') \wedge \\ & \text{no_cv_union?}(t') \wedge \\ & \text{no_cv_function?}(t') \wedge \\ & \text{no_reference_to_void?}(t') \wedge \\ & \text{no_cv_void?}(t') \wedge \text{no_array_of_bitfields?}(t') \end{aligned} $

Replacing using formula -1,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 This completes the proof of Search_Pre_TCC2.1.2.

Search_Pre_TCC2.2:

{1}	$\text{cv}(\text{pointer?})(\text{pointer}(\text{uint}))$
-----	---

Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of Search_Pre_TCC2.2.
 Q.E.D.

C.198.4 Search_Example.Search_Pre_TCC3

Terse proof for Search_Pre_TCC3.

Search_Pre_TCC3:

{1}	$\forall (s: \text{State}): \text{pm}'\text{states}(s) \wedge \text{plain_memory?}(\text{pm}) \supset \text{Cpp_Type?}(\text{pointer}(\text{uint}))$
-----	--

Repeatedly Skolemizing and flattening,
 Using lemma Search_Pre_TCC2,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 This completes the proof of Search_Pre_TCC3.

Q.E.D.

C.198.5 Search_Example.Search_Pre_TCC4

Terse proof for Search_Pre_TCC4.

Search_Pre_TCC4:

$$\frac{}{\{1\} \quad \forall (s: \text{State}): \text{pm}'\text{states}(s) \wedge \text{plain_memory?}(\text{pm}) \supset \text{pointer?}(\text{cv_base}(\text{pointer}(\text{uint})))}$$

Repeatedly Skolemizing and flattening,

Rewriting using cv_base_result, matching in *,

we get 2 subgoals:

Search_Pre_TCC4.1:

$$\frac{\begin{array}{l} \{-1\} \quad \text{pm}'\text{states}(s') \\ \{-2\} \quad \text{plain_memory?}(\text{pm}) \end{array}}{\begin{array}{l} \{1\} \quad \text{cv}(\text{pointer?})(\text{pointer}(\text{uint})) \\ \{2\} \quad \text{pointer?}(\text{cv_base}(\text{pointer}(\text{uint}))) \end{array}}$$

Using lemma Search_Pre_TCC2,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of Search_Pre_TCC4.1.

Search_Pre_TCC4.2:

$$\frac{\begin{array}{l} \{-1\} \quad \text{pm}'\text{states}(s') \\ \{-2\} \quad \text{plain_memory?}(\text{pm}) \end{array}}{\begin{array}{l} \{1\} \quad (\text{pointer?} \subseteq \text{interpreted?}) \\ \{2\} \quad \text{pointer?}(\text{cv_base}(\text{pointer}(\text{uint}))) \end{array}}$$

Keeping (1) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of Search_Pre_TCC4.2.

Q.E.D.

C.198.6 Search_Example.Search_Pre_TCC5

Terse proof for Search_Pre_TCC5.

Search_Pre_TCC5:

```

{1}  ∀ (s: State):
      blocks_disjoint?(current, size_of(pointer(uint)), last, size_of(pointer(uint))) ∧
      blocks_disjoint?(last, size_of(pointer(uint)), current, size_of(pointer(uint))) ∧
      blocks_disjoint?(current, size_of(pointer(uint)), first, size_of(pointer(uint))) ∧
      blocks_disjoint?(first, size_of(pointer(uint)), current, size_of(pointer(uint))) ∧
      in_blessed_memory?(dt_cv_pointer(pointer(uint)), current, pm'rw_addr) ∧
      valid_in_mem(pm, dt_cv_pointer(pointer(uint)))(last)(s) ∧
      valid_in_mem(pm, dt_cv_pointer(pointer(uint)))(first)(s) ∧
      in_blessed_memory?(dt_cv_pointer(pointer(uint)), last,
                        (pm'ro_addr ∪ pm'rw_addr))
      ∧
      in_blessed_memory?(dt_cv_pointer(pointer(uint)), first,
                        (pm'ro_addr ∪ pm'rw_addr))
      ∧ pm'states(s) ∧ plain_memory?(pm)
    ⊃
    OK?[State, ((range_pointer(cv_base(pointer(uint)))))]
      (read_data[State, ((range_pointer(cv_base(pointer(uint)))))]
        (pm[State], dt_cv_pointer(pointer(uint)))(first)(s))

```

Repeatedly Skolemizing and flattening,
 Rewriting using plain_memory_read_data_ok, matching in *,
 This completes the proof of Search_Pre_TCC5.
 Q.E.D.

C.198.7 Search_Example.Search_Pre_TCC6

Terse proof for Search_Pre_TCC6.

Search_Pre_TCC6:

```

{1}  ∀ (s: State):
      blocks_disjoint?(current, size_of(pointer(uint)), last, size_of(pointer(uint))) ∧
      blocks_disjoint?(last, size_of(pointer(uint)), current, size_of(pointer(uint))) ∧
      blocks_disjoint?(current, size_of(pointer(uint)), first, size_of(pointer(uint))) ∧
      blocks_disjoint?(first, size_of(pointer(uint)), current, size_of(pointer(uint))) ∧
      in_blessed_memory?(dt_cv_pointer(pointer(uint)), current, pm'rw_addr) ∧
      valid_in_mem(pm, dt_cv_pointer(pointer(uint)))(last)(s) ∧
      valid_in_mem(pm, dt_cv_pointer(pointer(uint)))(first)(s) ∧
      in_blessed_memory?(dt_cv_pointer(pointer(uint)), last,
                        (pm'ro_addr ∪ pm'rw_addr))
      ∧
      in_blessed_memory?(dt_cv_pointer(pointer(uint)), first,
                        (pm'ro_addr ∪ pm'rw_addr))
      ∧ pm'states(s) ∧ plain_memory?(pm)
    ⊃
    (∀ (first_value: ((range_pointer(cv_base(pointer(uint)))))):
      first_value = data(read_data(pm, dt_cv_pointer(pointer(uint)))(first)(s)) ⊃
      OK?[State, ((range_pointer(cv_base(pointer(uint)))))]
        (read_data[State, ((range_pointer(cv_base(pointer(uint)))))]
          (pm[State], dt_cv_pointer(pointer(uint)))(last)(s))

```

Repeatedly Skolemizing and flattening,

Rewriting using plain_memory_read_data_ok, matching in *,

This completes the proof of Search_Pre_TCC6.

Q.E.D.

C.198.8 Search_Example.Search_Pre_TCC7

Terse proof for Search_Pre_TCC7.

Search_Pre_TCC7:

<pre> {1} ∀ (s: State): blocks_disjoint?(current, size_of(pointer(uint)), last, size_of(pointer(uint))) ∧ blocks_disjoint?(last, size_of(pointer(uint)), current, size_of(pointer(uint))) ∧ blocks_disjoint?(current, size_of(pointer(uint)), first, size_of(pointer(uint))) ∧ blocks_disjoint?(first, size_of(pointer(uint)), current, size_of(pointer(uint))) ∧ in_blessed_memory?(dt_cv_pointer(pointer(uint)), current, pm'rw_addr) ∧ valid_in_mem(pm, dt_cv_pointer(pointer(uint)))(last)(s) ∧ valid_in_mem(pm, dt_cv_pointer(pointer(uint)))(first)(s) ∧ in_blessed_memory?(dt_cv_pointer(pointer(uint)), last, (pm'ro_addr ∪ pm'rw_addr)) ∧ in_blessed_memory?(dt_cv_pointer(pointer(uint)), first, (pm'ro_addr ∪ pm'rw_addr)) ∧ pm'states(s) ∧ plain_memory?(pm) ⊃ (∀ (first_value: ((range_pointer(cv_base(pointer(uint)))))): first_value = data(read_data(pm, dt_cv_pointer(pointer(uint)))(first)(s)) ⊃ (∀ (last_value: ((range_pointer(cv_base(pointer(uint)))))): not_null?(pointer(uint))(last_value) ∧ not_null?(pointer(uint))(first_value) ∧ last_value = data(read_data(pm, dt_cv_pointer(pointer(uint)))(last)(s)) ⊃ (∀ (i: below[N]): Cpp_Type?(uint)))) </pre>

Repeatedly Skolemizing and flattening,

Using lemma Search_Pre_TCC1,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of Search_Pre_TCC7.

Q.E.D.

C.198.9 Search_Example.Search_Pre_TCC8

Terse proof for Search_Pre_TCC8.

Search_Pre_TCC8:

$$\begin{aligned}
 & \{1\} \quad \forall (s: \text{State}): \\
 & \quad \text{blocks_disjoint?}(\text{current}, \text{size_of}(\text{pointer}(\text{uint})), \text{last}, \text{size_of}(\text{pointer}(\text{uint}))) \wedge \\
 & \quad \text{blocks_disjoint?}(\text{last}, \text{size_of}(\text{pointer}(\text{uint})), \text{current}, \text{size_of}(\text{pointer}(\text{uint}))) \wedge \\
 & \quad \text{blocks_disjoint?}(\text{current}, \text{size_of}(\text{pointer}(\text{uint})), \text{first}, \text{size_of}(\text{pointer}(\text{uint}))) \wedge \\
 & \quad \text{blocks_disjoint?}(\text{first}, \text{size_of}(\text{pointer}(\text{uint})), \text{current}, \text{size_of}(\text{pointer}(\text{uint}))) \wedge \\
 & \quad \text{in_blessed_memory?}(\text{dt_cv_pointer}(\text{pointer}(\text{uint})), \text{current}, \text{pm}'\text{rw_addr}) \wedge \\
 & \quad \text{valid_in_mem}(\text{pm}, \text{dt_cv_pointer}(\text{pointer}(\text{uint}))) (\text{last})(s) \wedge \\
 & \quad \text{valid_in_mem}(\text{pm}, \text{dt_cv_pointer}(\text{pointer}(\text{uint}))) (\text{first})(s) \wedge \\
 & \quad \text{in_blessed_memory?}(\text{dt_cv_pointer}(\text{pointer}(\text{uint})), \text{last}, \\
 & \quad \quad (\text{pm}'\text{ro_addr} \cup \text{pm}'\text{rw_addr})) \\
 & \quad \wedge \\
 & \quad \text{in_blessed_memory?}(\text{dt_cv_pointer}(\text{pointer}(\text{uint})), \text{first}, \\
 & \quad \quad (\text{pm}'\text{ro_addr} \cup \text{pm}'\text{rw_addr})) \\
 & \quad \wedge \text{pm}'\text{states}(s) \wedge \text{plain_memory?}(\text{pm}) \\
 & \quad \supset \\
 & \quad (\forall (\text{first_value}: ((\text{range_pointer}(\text{cv_base}(\text{pointer}(\text{uint})))))): \\
 & \quad \quad \text{first_value} = \text{data}(\text{read_data}(\text{pm}, \text{dt_cv_pointer}(\text{pointer}(\text{uint}))) (\text{first})(s)) \supset \\
 & \quad \quad (\forall (\text{last_value}: ((\text{range_pointer}(\text{cv_base}(\text{pointer}(\text{uint})))))): \\
 & \quad \quad \quad \text{not_null?}(\text{pointer}(\text{uint})) (\text{last_value}) \wedge \\
 & \quad \quad \quad \text{not_null?}(\text{pointer}(\text{uint})) (\text{first_value}) \wedge \\
 & \quad \quad \quad \text{last_value} = \text{data}(\text{read_data}(\text{pm}, \text{dt_cv_pointer}(\text{pointer}(\text{uint}))) (\text{last})(s)) \\
 & \quad \quad \quad \supset \\
 & \quad \quad (\forall (i: \text{below}[N]): \\
 & \quad \quad \quad \text{up?}[\text{Memory_Address}] \\
 & \quad \quad \quad \quad (\text{address_of}(\text{pointer}(\text{uint})) \\
 & \quad \quad \quad \quad \quad (\text{add}(\text{pointer}(\text{uint})) \\
 & \quad \quad \quad \quad \quad \quad (\text{first_value}, i \times \text{size_of}(\text{uint}))))))
 \end{aligned}$$

Repeatedly Skolemizing and flattening,

Rewriting using `address_of_spec`, matching in `*`,

Keeping (-17 1) and hiding `*`,

Using lemma `add_spec`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `Search_Pre_TCC8`.

Q.E.D.

C.198.10 Search_Example.Search_Pre_TCC9

Terse proof for `Search_Pre_TCC9`.

Search_Pre_TCC9:

```

{1}  ∀ (s: State):
    blocks_disjoint?(current, size_of(pointer(uint)), last, size_of(pointer(uint))) ∧
    blocks_disjoint?(last, size_of(pointer(uint)), current, size_of(pointer(uint))) ∧
    blocks_disjoint?(current, size_of(pointer(uint)), first, size_of(pointer(uint))) ∧
    blocks_disjoint?(first, size_of(pointer(uint)), current, size_of(pointer(uint))) ∧
    in_blessed_memory?(dt_cv_pointer(pointer(uint)), current, pm'rw_addr) ∧
    valid_in_mem(pm, dt_cv_pointer(pointer(uint)))(last)(s) ∧
    valid_in_mem(pm, dt_cv_pointer(pointer(uint)))(first)(s) ∧
    in_blessed_memory?(dt_cv_pointer(pointer(uint)), last,
        (pm'ro_addr ∪ pm'rw_addr))
    ∧
    in_blessed_memory?(dt_cv_pointer(pointer(uint)), first,
        (pm'ro_addr ∪ pm'rw_addr))
    ∧ pm'states(s) ∧ plain_memory?(pm)
    ⊃
    (∀ (first_value: ((range_pointer(cv_base(pointer(uint)))))):
        first_value = data(read_data(pm, dt_cv_pointer(pointer(uint)))(first)(s)) ⊃
        (∀ (last_value: ((range_pointer(cv_base(pointer(uint)))))):
            (∀ (i: below[N]):
                valid_in_mem(pm, dt(uint))
                    (down(address_of(pointer(uint))
                        (add(pointer(uint))
                            (first_value,
                                i × size_of(uint))))))
                    (s))
            ∧
            (∀ (i: below[N]):
                blocks_disjoint?(current, size_of(pointer(uint)),
                    down(address_of(pointer(uint))
                        (add(pointer(uint))
                            (first_value,
                                i × size_of(uint))))),
                    size_of(uint)))
            ∧
            (∀ (i: below[N]):
                blocks_disjoint?(down(address_of(pointer(uint))
                    (add(pointer(uint))
                        (first_value,
                            i × size_of(uint))))),
                    size_of(uint), current, size_of(pointer(uint))))
            ∧
            (∀ (i: below[N]):
                in_blessed_memory?[(range(uint))]
                    (dt(uint),
                        down(address_of(pointer(uint))
                            (add(pointer(uint))
                                (first_value, i × size_of(uint))))),
                    (pm'ro_addr ∪ pm'rw_addr)))
            ∧
            not_null?(pointer(uint))(last_value) ∧
            not_null?(pointer(uint))(first_value) ∧
            last_value =
                data(read_data(pm, dt_cv_pointer(pointer(uint)))(last)(s))
            ⊃ Cpp_Type?(uint)))

```

C Proof scripts

Repeatedly Skolemizing and flattening,

Using lemma `Search_Pre_TCC1`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `Search_Pre_TCC9`.

Q.E.D.

C.198.11 `Search_Example.search_TCC1`

Terse proof for `search_TCC1`.

`search_TCC1`:

$$\frac{}{\{1\} \text{ Cpp_Type?}(pointer(uint)) \wedge cv(pointer?)(pointer(uint))}$$

Applying propositional simplification,

we get 2 subgoals:

`search_TCC1.1`:

$$\frac{}{\{1\} \text{ Cpp_Type?}(pointer(uint))}$$

Expanding the definition of `Cpp_Type?`,

Repeatedly Skolemizing and flattening,

Expanding the definition of subterm,

Expanding the definition of subterm,

Installing automatic rewrites from: `no_pointers_to_bitfield?` `no_pointers_to_references?`
`no_cv_references?` `no_reference_to_reference?` `no_reference_to_bitfields?` `no_pointer_to_member_to_reference?`
`no_pointer_to_member_to_cv_void?` `no_cv_void_parameter?` `no_array_of_references?` `no_array_of_cv_void?`
`no_array_of_function?` `no_array_of_abstract_class?` `no_pointer_or_ref_to_incomplete_array_parameter?`
`no_array_return_type?` `no_function_return_type?` `bitfield_underlying_integral_or_enum_type?` `cv_array?`
`enum_underlying_integral?` `enum_constants?` `const_volatile?` `const_stutter?` `volatile_stutter?`
`no_cv_class?` `no_cv_union?` `no_cv_function?` `no_reference_to_void?` `no_cv_void?` `no_array_of_bitfields?`

Splitting conjunctions,

we get 2 subgoals:

search_TCC1.1.1.1:

{-1}	$t' = \text{pointer}(\text{uint})$
{1}	$\begin{aligned} & \text{no_pointers_to_bitfield?}(t') \wedge \\ & \text{no_pointers_to_references?}(t') \wedge \\ & \text{no_cv_references?}(t') \wedge \\ & \text{no_reference_to_reference?}(t') \wedge \\ & \text{no_reference_to_bitfields?}(t') \wedge \\ & \text{no_pointer_to_member_to_reference?}(t') \wedge \\ & \text{no_pointer_to_member_to_cv_void?}(t') \wedge \\ & \text{no_cv_void_parameter?}(t') \wedge \\ & \text{no_array_of_references?}(t') \wedge \\ & \text{no_array_of_cv_void?}(t') \wedge \\ & \text{no_array_of_function?}(t') \wedge \\ & \text{no_array_of_abstract_class?}(t') \wedge \\ & \text{no_pointer_or_ref_to_incomplete_array_parameter?}(t') \wedge \\ & \text{no_array_return_type?}(t') \wedge \\ & \text{no_function_return_type?}(t') \wedge \\ & \text{bitfield_underlying_integral_or_enum_type?}(t') \wedge \\ & \text{cv_array?}(t') \wedge \\ & \text{enum_underlying_integral?}(t') \wedge \\ & \text{enum_constants?}(t') \wedge \\ & \text{const_volatile?}(t') \wedge \\ & \text{const_stutter?}(t') \wedge \\ & \text{volatile_stutter?}(t') \wedge \\ & \text{no_cv_class?}(t') \wedge \\ & \text{no_cv_union?}(t') \wedge \\ & \text{no_cv_function?}(t') \wedge \\ & \text{no_reference_to_void?}(t') \wedge \\ & \text{no_cv_void?}(t') \wedge \text{no_array_of_bitfields?}(t') \end{aligned}$

Replacing using formula -1,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Keeping (-2) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of search_TCC1.1.1.1.

search_TCC1.1.2:

{-1}	$t' = \text{uint}$
{1}	$\text{no_pointers_to_bitfield?}(t') \wedge$ $\text{no_pointers_to_references?}(t') \wedge$ $\text{no_cv_references?}(t') \wedge$ $\text{no_reference_to_reference?}(t') \wedge$ $\text{no_reference_to_bitfields?}(t') \wedge$ $\text{no_pointer_to_member_to_reference?}(t') \wedge$ $\text{no_pointer_to_member_to_cv_void?}(t') \wedge$ $\text{no_cv_void_parameter?}(t') \wedge$ $\text{no_array_of_references?}(t') \wedge$ $\text{no_array_of_cv_void?}(t') \wedge$ $\text{no_array_of_function?}(t') \wedge$ $\text{no_array_of_abstract_class?}(t') \wedge$ $\text{no_pointer_or_ref_to_incomplete_array_parameter?}(t') \wedge$ $\text{no_array_return_type?}(t') \wedge$ $\text{no_function_return_type?}(t') \wedge$ $\text{bitfield_underlying_integral_or_enum_type?}(t') \wedge$ $\text{cv_array?}(t') \wedge$ $\text{enum_underlying_integral?}(t') \wedge$ $\text{enum_constants?}(t') \wedge$ $\text{const_volatile?}(t') \wedge$ $\text{const_stutter?}(t') \wedge$ $\text{volatile_stutter?}(t') \wedge$ $\text{no_cv_class?}(t') \wedge$ $\text{no_cv_union?}(t') \wedge$ $\text{no_cv_function?}(t') \wedge$ $\text{no_reference_to_void?}(t') \wedge$ $\text{no_cv_void?}(t') \wedge \text{no_array_of_bitfields?}(t')$

Replacing using formula -1,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 This completes the proof of search_TCC1.1.2.

search_TCC1.2:

{1}	$\text{cv}(\text{pointer?})(\text{pointer}(\text{uint}))$
-----	---

Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of search_TCC1.2.
 Q.E.D.

C.198.12 Search_Example.search_TCC2

Terse proof for search_TCC2.

search_TCC2:

{1}	$\text{Cpp_Type?}(\text{pointer}(\text{uint}))$
-----	--

Using lemma search_TCC1,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 This completes the proof of search_TCC2.
 Q.E.D.

C.198.13 Search_Example.search_TCC3

Terse proof for `search_TCC3`.

`search_TCC3`:

{1} `pointer?(cv_base(pointer(uint)))`

Rewriting using `cv_base_result`, matching in `*`,
we get 2 subgoals:

`search_TCC3.1`:

{1} `cv(pointer?)(pointer(uint))`
{2} `pointer?(cv_base(pointer(uint)))`

Using lemma `search_TCC1`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `search_TCC3.1`.

`search_TCC3.2`:

{1} `(pointer? \subseteq interpreted?)`
{2} `pointer?(cv_base(pointer(uint)))`

Hiding formulas: 2,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `search_TCC3.2`.

Q.E.D.

C.198.14 Search_Example.search_TCC4

Terse proof for `search_TCC4`.

`search_TCC4`:

{1} `Cpp_Type?(pointer(uint)) \wedge v(pointer?)(pointer(uint))`

Applying propositional simplification,

we get 2 subgoals:

`search_TCC4.1`:

{1} `Cpp_Type?(pointer(uint))`

Installing automatic rewrites from: `no_pointers_to_bitfield?` `no_pointers_to_references?`
`no_cv_references?` `no_reference_to_reference?` `no_reference_to_bitfields?` `no_pointer_to_member_to_reference?`
`no_pointer_to_member_to_cv_void?` `no_cv_void_parameter?` `no_array_of_references?` `no_array_of_cv_void?`
`no_array_of_function?` `no_array_of_abstract_class?` `no_pointer_or_ref_to_incomplete_array_parameter?`
`no_array_return_type?` `no_function_return_type?` `bitfield_underlying_integral_or_enum_type?` `cv_array?`
`enum_underlying_integral?` `enum_constants?` `const_volatile?` `const_stutter?` `volatile_stutter?`
`no_cv_class?` `no_cv_union?` `no_cv_function?` `no_reference_to_void?` `no_cv_void?` `no_array_of_bitfields?`

Expanding the definition of `Cpp_Type?`,

Repeatedly Skolemizing and flattening,

Expanding the definition of subterm,

Expanding the definition of subterm,

Splitting conjunctions,

C Proof scripts

we get 2 subgoals:

search_TCC4.1.1:

{-1}	$t' = \text{pointer}(\text{uint})$
{1}	$\text{no_pointers_to_bitfield?}(t') \wedge$ $\text{no_pointers_to_references?}(t') \wedge$ $\text{no_cv_references?}(t') \wedge$ $\text{no_reference_to_reference?}(t') \wedge$ $\text{no_reference_to_bitfields?}(t') \wedge$ $\text{no_pointer_to_member_to_reference?}(t') \wedge$ $\text{no_pointer_to_member_to_cv_void?}(t') \wedge$ $\text{no_cv_void_parameter?}(t') \wedge$ $\text{no_array_of_references?}(t') \wedge$ $\text{no_array_of_cv_void?}(t') \wedge$ $\text{no_array_of_function?}(t') \wedge$ $\text{no_array_of_abstract_class?}(t') \wedge$ $\text{no_pointer_or_ref_to_incomplete_array_parameter?}(t') \wedge$ $\text{no_array_return_type?}(t') \wedge$ $\text{no_function_return_type?}(t') \wedge$ $\text{bitfield_underlying_integral_or_enum_type?}(t') \wedge$ $\text{cv_array?}(t') \wedge$ $\text{enum_underlying_integral?}(t') \wedge$ $\text{enum_constants?}(t') \wedge$ $\text{const_volatile?}(t') \wedge$ $\text{const_stutter?}(t') \wedge$ $\text{volatile_stutter?}(t') \wedge$ $\text{no_cv_class?}(t') \wedge$ $\text{no_cv_union?}(t') \wedge$ $\text{no_cv_function?}(t') \wedge$ $\text{no_reference_to_void?}(t') \wedge$ $\text{no_cv_void?}(t') \wedge \text{no_array_of_bitfields?}(t')$

Replacing using formula -1,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of search_TCC4.1.1.

search_TCC4.1.2:

{-1}	$t' = \text{uint}$
{1}	$\text{no_pointers_to_bitfield?}(t') \wedge$ $\text{no_pointers_to_references?}(t') \wedge$ $\text{no_cv_references?}(t') \wedge$ $\text{no_reference_to_reference?}(t') \wedge$ $\text{no_reference_to_bitfields?}(t') \wedge$ $\text{no_pointer_to_member_to_reference?}(t') \wedge$ $\text{no_pointer_to_member_to_cv_void?}(t') \wedge$ $\text{no_cv_void_parameter?}(t') \wedge$ $\text{no_array_of_references?}(t') \wedge$ $\text{no_array_of_cv_void?}(t') \wedge$ $\text{no_array_of_function?}(t') \wedge$ $\text{no_array_of_abstract_class?}(t') \wedge$ $\text{no_pointer_or_ref_to_incomplete_array_parameter?}(t') \wedge$ $\text{no_array_return_type?}(t') \wedge$ $\text{no_function_return_type?}(t') \wedge$ $\text{bitfield_underlying_integral_or_enum_type?}(t') \wedge$ $\text{cv_array?}(t') \wedge$ $\text{enum_underlying_integral?}(t') \wedge$ $\text{enum_constants?}(t') \wedge$ $\text{const_volatile?}(t') \wedge$ $\text{const_stutter?}(t') \wedge$ $\text{volatile_stutter?}(t') \wedge$ $\text{no_cv_class?}(t') \wedge$ $\text{no_cv_union?}(t') \wedge$ $\text{no_cv_function?}(t') \wedge$ $\text{no_reference_to_void?}(t') \wedge$ $\text{no_cv_void?}(t') \wedge \text{no_array_of_bitfields?}(t')$

Replacing using formula -1,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of search_TCC4.1.2.

search_TCC4.2:

{1}	$v(\text{pointer?})(\text{pointer}(\text{uint}))$
-----	---

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of search_TCC4.2.

Q.E.D.

C.198.15 Search_Example.deref_current_TCC1

Terse proof for deref_current_TCC1.

deref_current_TCC1:

{1}	$\forall (s: \text{State}):$ $\text{read_current}(s) \supset$ $\text{OK?}[\text{State}, ((\text{range_pointer}(\text{cv_base}(\text{pointer}(\text{uint})))))]$ $(\text{read_data}[\text{State}, ((\text{range_pointer}(\text{cv_base}(\text{pointer}(\text{uint}))))])$ $(\text{pm}[\text{State}], \text{dt_cv_pointer}(\text{pointer}(\text{uint})))(\text{current})(s)$
-----	--

Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of `deref_current_TCC1`.
 Q.E.D.

C.198.16 Search_Example.deref_current_TCC2

Terse proof for `deref_current_TCC2`.

`deref_current_TCC2`:

```
{1}  ∀ (s: State):
      read_current(s) ⊃
      (∀ (current_value: ((range_pointer(cv_base(pointer(uint)))))):
        not_null?(pointer(uint))(current_value) ∧
        current_value = data(read_data(pm, dt_cv_pointer(pointer(uint)))(current)(s))
        ⊃ up?[Memory_Address](address_of(pointer(uint))(current_value)))
```

Repeatedly Skolemizing and flattening,
 Rewriting using `address_of_spec`, matching in `*`,
 This completes the proof of `deref_current_TCC2`.
 Q.E.D.

C.198.17 Search_Example.read_array_TCC1

Terse proof for `read_array_TCC1`.

`read_array_TCC1`:

```
{1}  ∀ (s: State):
      read_first(s) ⊃
      OK?[State, ((range_pointer(cv_base(pointer(uint)))))]
      (read_data[State, ((range_pointer(cv_base(pointer(uint)))))]
       (pm[State], dt_cv_pointer(pointer(uint)))(first)(s))
```

Repeatedly Skolemizing and flattening,
 Expanding the definition of `read_first`,
 which is trivially true.
 This completes the proof of `read_array_TCC1`.
 Q.E.D.

C.198.18 Search_Example.read_array_TCC2

Terse proof for `read_array_TCC2`.

`read_array_TCC2`:

```
{1}  ∀ (s: State):
      read_first(s) ⊃
      (∀ (first_value: ((range_pointer(cv_base(pointer(uint)))))):
        not_null?(pointer(uint))(first_value) ∧
        first_value = data(read_data(pm, dt_cv_pointer(pointer(uint)))(first)(s))
        ⊃ (∀ (i: below[N]): Cpp_Type?(uint)))
```

Repeatedly Skolemizing and flattening,

Using lemma Search_Pre_TCC1,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 This completes the proof of read_array_TCC2.
 Q.E.D.

C.198.19 Search_Example.read_array_TCC3

Terse proof for read_array_TCC3.

read_array_TCC3:

<pre> {1} ∀ (s: State): read_first(s) ⊃ (∀ (first_value: ((range_pointer(cv_base(pointer(uint)))))): not_null?(pointer(uint))(first_value) ∧ first_value = data(read_data(pm, dt_cv_pointer(pointer(uint)))(first)(s)) ⊃ (∀ (i: below[N]): up?[Memory_Address] (address_of(pointer(uint)) (add(pointer(uint)) (first_value, i × size_of(uint)))))) </pre>
--

Repeatedly Skolemizing and flattening,
 Rewriting using address_of_spec, matching in *,
 Using lemma add_spec,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 This completes the proof of read_array_TCC3.
 Q.E.D.

C.198.20 Search_Example.not_found_TCC1

Terse proof for not_found_TCC1.

not_found_TCC1:

<pre> {1} ∀ (s: State): read_array(s) ∧ read_last(s) ∧ read_current(s) ∧ read_first(s) ⊃ (∀ (first_value: ((range_pointer(cv_base(pointer(uint)))))): first_value = data(read_data(pm, dt_cv_pointer(pointer(uint)))(first)(s)) ⊃ (∀ (i: below[N]): Cpp_Type?(uint))) </pre>

Repeatedly Skolemizing and flattening,
 Using lemma Search_Pre_TCC1,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 This completes the proof of not_found_TCC1.
 Q.E.D.

C.198.21 Search_Example.not_found_TCC2

Terse proof for not_found_TCC2.

not_found_TCC2:

```
{1}  ∀ (s: State):
      read_array(s) ∧ read_last(s) ∧ read_current(s) ∧ read_first(s) ⊃
      (∀ (first_value: ((range_pointer(cv_base(pointer(uint)))))):
        first_value = data(read_data(pm, dt_cv_pointer(pointer(uint)))(first)(s)) ⊃
        (∀ (i: below[N]):
          up?[Memory_Address]
            (address_of(pointer(uint))
              (add(pointer(uint))
                (first_value, i × size_of(uint)))))))
```

Repeatedly Skolemizing and flattening,
 Rewriting using address_of_spec, matching in *,
 Using lemma add_spec,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 Expanding the definition of read_array,
 which is trivially true.
 This completes the proof of not_found_TCC2.
 Q.E.D.

C.198.22 Search_Example.not_found_TCC3

Terse proof for not_found_TCC3.

not_found_TCC3:

```
{1}  ∀ (s: State):
      read_array(s) ∧ read_last(s) ∧ read_current(s) ∧ read_first(s) ⊃
      (∀ (first_value: ((range_pointer(cv_base(pointer(uint)))))):
        first_value = data(read_data(pm, dt_cv_pointer(pointer(uint)))(first)(s)) ⊃
        (∀ (i: below[N]):
          OK?[State, ((range(uint)))]
            (read_data[State, ((range(uint)))]
              (pm[State], dt(uint))
                (down[Memory_Address]
                  (address_of(pointer(uint))
                    (add(pointer(uint))
                      (first_value, i × size_of(uint)))))))
            (s))))
```

Repeatedly Skolemizing and flattening,
 Expanding the definition of read_array,
 Replacing using formula -7,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 Instantiating quantified variables,
 This completes the proof of not_found_TCC3.
 Q.E.D.

C.198.23 Search_Example.not_found_TCC4

Terse proof for not_found_TCC4.

not_found_TCC4:

```

{1}  ∀ (s: State):
      read_first(s) ∧
      read_current(s) ∧
      read_last(s) ∧
      read_array(s) ∧
      (LET first_value = data(read_data(pm, dt_cv_pointer(pointer(uint)))(first)(s))
        IN
        ∀ (i: below[N]):
          (data(read_data(pm, dt(uint))
                    (down(address_of(pointer(uint))
                          (add(pointer(uint))
                              (first_value,
                                i × size_of(uint))))))
            (s))
            ≠ value))
      ⊃
      OK?[State, ((range_pointer(cv_base(pointer(uint)))))]
        (read_data[State, ((range_pointer(cv_base(pointer(uint)))))]
          (pm[State], dt_cv_pointer(pointer(uint)))(last)(s))

```

Repeatedly Skolemizing and flattening,
 Expanding the definition of read_last,
 which is trivially true.
 This completes the proof of not_found_TCC4.
 Q.E.D.

C.198.24 Search_Example.found_TCC1

Terse proof for found_TCC1.

found_TCC1:

```

{1}  ∀ (s: State):
      deref_current(s) ∧
      read_array(s) ∧ read_last(s) ∧ read_current(s) ∧ read_first(s)
      ⊃
      OK?[State, ((range_pointer(cv_base(pointer(uint)))))]
        (read_data[State, ((range_pointer(cv_base(pointer(uint)))))]
          (pm[State], dt_cv_pointer(pointer(uint)))(last)(s))

```

Repeatedly Skolemizing and flattening,
 Expanding the definition of read_last,
 which is trivially true.
 This completes the proof of found_TCC1.
 Q.E.D.

C.198.25 Search_Example.found_TCC2

Terse proof for found_TCC2.

found_TCC2:

<pre> {1} ∀ (s: State): read_first(s) ∧ read_current(s) ∧ read_last(s) ∧ read_array(s) ∧ deref_current(s) ∧ data(read_data(pm, dt_cv_pointer(pointer(uint)))(last)(s)) ≠ data(read_data(pm, dt_cv_pointer(pointer(uint)))(current)(s)) ⊃ up?[Memory_Address] (address_of(pointer(uint)) (data[State, ((range_pointer(cv_base(pointer(uint))))]) (read_data[State, ((range_pointer(cv_base(pointer(uint))))]) (pm[State], dt_cv_pointer(pointer(uint)))(current) (s))))) </pre>

Repeatedly Skolemizing and flattening,

Expanding the definition of deref_current,

Applying disjunctive simplification to flatten sequent,

Rewriting using address_of_spec, matching in *,

Expanding the definition of cv_base,

which is trivially true.

This completes the proof of found_TCC2.

Q.E.D.

C.198.26 Search_Example.found_TCC3

Terse proof for found_TCC3.

found_TCC3:

<pre> {1} ∀ (s: State): read_first(s) ∧ read_current(s) ∧ read_last(s) ∧ read_array(s) ∧ deref_current(s) ∧ data(read_data(pm, dt_cv_pointer(pointer(uint)))(last)(s)) ≠ data(read_data(pm, dt_cv_pointer(pointer(uint)))(current)(s)) ⊃ OK?[State, ((range(uint)))] (read_data[State, ((range(uint)))] (pm[State], dt(uint)) (down[Memory_Address] (address_of(pointer(uint)) (data[State, ((range_pointer(cv_base(pointer(uint))))]) (read_data [State, ((range_pointer(cv_base(pointer(uint))))]) (pm[State], dt_cv_pointer(pointer(uint)) (current)(s)))))) (s)) </pre>
--

Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of found_TCC3.
 Q.E.D.

C.198.27 Search_Example.uchar_range_values_TCC1

Terse proof for uchar_range_values_TCC1.

uchar_range_values_TCC1:

<pre> {1} ∀ (n: nat): n ≤ 255 ⊃ Cpp_Type?(uchar) ∧ cv(non_bool_integral_enum?)(uchar) </pre>

Repeatedly Skolemizing and flattening,
 Applying propositional simplification,
 we get 2 subgoals:

uchar_range_values_TCC1.1:

<pre> {-1} n' ≥ 0 {-2} n' ≤ 255 {1} Cpp_Type?(uchar) </pre>

Expanding the definition of Cpp_Type?,
 Repeatedly Skolemizing and flattening,
 Expanding the definition of subterm,
 Replacing using formula -1,

Installing automatic rewrites from: no_pointers_to_bitfield? no_pointers_to_references?
 no_cv_references? no_reference_to_reference? no_reference_to_bitfields? no_pointer_to_member_to_reference?
 no_pointer_to_member_to_cv_void? no_cv_void_parameter? no_array_of_references? no_array_of_cv_void?
 no_array_of_function? no_array_of_abstract_class? no_pointer_or_ref_to_incomplete_array_parameter?
 no_array_return_type? no_function_return_type? bitfield_underlying_integral_or_enum_type? cv_array?

enum_underlying_integral? enum_constants? const_volatile? const_stutter? volatile_stutter?
no_cv_class? no_cv_union? no_cv_function? no_reference_to_void? no_cv_void? no_array_of_bitfields?

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of uchar_range_values_TCC1.1.

uchar_range_values_TCC1.2:

{-1}	$n' \geq 0$
{-2}	$n' \leq 255$
{1}	$cv(\text{non_bool_integral_enum?})(\text{uchar})$

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of uchar_range_values_TCC1.2.
Q.E.D.

C.198.28 Search_Example.uchar_range_values

Terse proof for uchar_range_values.

uchar_range_values:

{1}	$\forall (n: \text{nat}): n \leq 255 \supset \text{range}(\text{uchar})(n)$
-----	---

Repeatedly Skolemizing and flattening,
Expanding the definition of range,
Expanding the definition of range_integral,
Expanding the definition of extend,
Using lemma binary_range_uchar,
Using lemma max_uchar,
Using lemma min_max_value_bits_uchar,
Trying repeated skolemization, instantiation, and if-lifting,
Using lemma expt_ge1,
Expanding the definition of ^,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of uchar_range_values.
Q.E.D.

C.198.29 Search_Example.search_terminates

Terse proof for search_terminates.

search_terminates:

{1}	$\text{Valid}(\text{Search_Pre}, \text{search}, \lambda (s: \text{State}): \text{TRUE})$
-----	---

Installing automatic rewrites from: valid Valid Search_Pre search
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Repeatedly Skolemizing and flattening,
Installing automatic rewrites from: mod_range size_of uidt er_plus_ax er_minus_ax er_div_ax
er_times_ax er_neg_ax address_of_spec address_of_spec2 add_spec add_spec2 same_array_spec
same_array_spec2 check_bounds_spec extended_real_superset_of_number_fields
Installing automatic rewrites from: literal id member postinc postdec deref arrow unary_minus
unary_minus_unsigned unary_not unary_bitnot preinc predec UnaryExpressions.sizeof ptm_member
ptm_arrow arrow times div div_float mod plus plus_ptr postinc_ptr preinc_ptr subscript minus minus_ptr
postdec_ptr predec_ptr minus_ptr_ptr cmp lt gt le ge lt_ptr gt_ptr le_ptr ge_ptr cmp_pointer eq_bool
eq not_equal not_equal_bool eq_ptr not_equal_ptr bitand bitxor bitor and_exp or_exp conditional_expr

assign assign_times assign_div assign_mod assign_div_float assign_plus assign_plus_ptr assign_minus assign_minus_ptr assign_bitand assign_bitxor assign_bitor comma

Installing automatic rewrites from: plain_memory_write_ok_single plain_memory_read_data_ok plain_memory_write_ok_q_expr plain_memory_write_ok_q_stmt in_blessed_memory_rw_ro pm_q_prop_ok pm_q_prop_single_read pm_q_prop_single_write pm_q_prop_read_write_single pm_q_prop_read_write_other_single pm_q_prop_read_read_single plain_memory_read_write_other_single_data plain_memory_read_read plain_memory_read_write_res pm_q_prop_read_ok_expr pm_q_prop_read_ok_stmt pm_q_prop_read_pm_q_prop_expr pm_q_prop_read_pm_q_prop_stmt pm_q_prop_read_write_q_expr pm_q_prop_read_write_q_stmt plain_memory_read_write_q_data_expr plain_memory_read_write_q_data_stmt pm_q_prop_read_write_other_q_expr pm_q_prop_read_write_other_q_stmt pm_q_prop_read_read_q_expr pm_q_prop_read_read_q_stmt plain_memory_read_write_other_q_data_expr plain_memory_read_write_other_q_data_stmt plain_memory_read_read_q_data_expr plain_memory_read_read_q_data_stmt pm_q_prop_ok_result_single pm_q_prop_ok_result_q_expr pm_q_prop_ok_result_q_stmt pm_q_prop_read_ok_result_single pm_q_prop_read_ok_result_q_expr pm_q_prop_read_ok_result_q_stmt pm_q_prop_e2s pm_q_prop_stmt_e2s pm_q_prop_read_e2s pm_q_prop_read_stmt_e2s pm_q_prop_read_ok_pm_q_prop_read_expr pm_q_prop_read_ok_pm_q_prop_read_stmt

Installing automatic rewrites from: if_else switch member break_catch_break break_break_stmt break_break_catch_continue break_break_catch_default break_break_lift_stmt break_break_case break_break_default break_break_stmt_break_break break_stmt_break break_stmt_catch_continue break_stmt_catch_default break_stmt_lift break_stmt_lift_case break_stmt_default continue_catch_continue continue_break_stmt continue_break_catch_continue continue_break_catch_default continue_break_lift_stmt continue_break_case continue_break_default continue_continue stmt_continue_continue continue_stmt_continue continue_stmt_catch_continue continue_stmt_catch_default continue_stmt_lift continue_stmt_case continue_stmt_default switch_stmt stmt_switch_stmt switch_case_taken stmt_switch_case_taken switch_case_not_taken stmt_switch_case_not_taken switch_default stmt_switch_default switch_catch_break stmt_switch_catch_break switch_catch_continue stmt_switch_catch_continue switch_catch_default stmt_switch_catch_default default_stmt stmt_default_stmt default_case stmt_default_case default_default stmt_default_default default_catch_break stmt_default_catch_break default_catch_continue stmt_default_catch_continue default_catch_default stmt_default_catch_default return_void_result return_ex_result stmt_remove_catch_default stmt_remove_case stmt_remove_default stmt_remove_catch_continue stmt_remove_catch_break break_break_stmt_expr break_break_catch_continue_expr break_break_catch_default_expr break_break_lift_stmt_expr break_break_case_expr break_break_default_expr continue_break_stmt_expr continue_break_catch_continue_expr continue_break_catch_default_expr continue_break_lift_stmt_expr continue_break_case_expr continue_break_default_expr

Installing automatic rewrites from: skip_ok skip_state skip_elimination stmt_skip_elimination stmt_ok_lift stmt_ok_lift_expr e2s_ok stmt_e2s_ok e2s_state stmt_e2s_state ok_result_fexpr ok_result_fstmt e2s_merge stmt_e2s_merge e2s_expr stmt_e2s_expr

Installing automatic rewrites from: while_unroll stmt_while_unroll iterate_while_ok while_hang while_unroll_expr while_unroll_lexpr stmt_while_unroll_expr stmt_while_unroll_lexpr while_unroll_0 do_while_unroll do_while_inv_unroll for_unroll for_inv_unroll while_to_while_no_cb while_invariant?! while_variant?! while_inv_rewrite_termination_result! while_inv_rewrite_data_ok! while_inv_rewrite_data_break! while_inv_rewrite_data_return! stmt_while_inv_rewrite_termination_result! stmt_while_inv_rewrite_data_ok! stmt_while_inv_rewrite_data_break! stmt_while_inv_rewrite_data_return! pm_q_prop_while! pm_q_prop_stmt_while!

Installing automatic rewrites from: composition_assoc_rewrite_stmt_ok forward_s_c_s_of_s_expr comp_eval_if_ok_fstmt stmt_eval_if_ok_fstmt comp_eval_if_ok_fstmt_cstmt stmt_eval_if_ok_fstmt_cstmt composition_assoc_rewrite_1 composition_assoc_rewrite_2 composition_assoc_rewrite_3 composition_assoc_rewrite_4 composition_assoc_rewrite_5 composition_assoc_rewrite_6 composition_assoc_rewrite_7 composition_assoc_rewrite_8 composition_assoc_rewrite_expr_1 composition_assoc_rewrite_expr_2 composition_assoc_rewrite_expr_3 composition_assoc_rewrite_expr_4 composition_assoc_rewrite_expr_5 composition_assoc_rewrite_expr_6 composition_assoc_rewrite_stmt_ok composition_assoc_rewrite_stmt_ok_expr composition_assoc_rewrite_stmt_ok_expr_lexpr comp_eval_if_ok_fstmt comp_eval_if_ok_fstmt_expr comp_eval_if_ok_fstmt_cstmt_expr stmt_eval_if_ok_fstmt_expr stmt_eval_if_ok_fstmt_cstmt_expr

C Proof scripts

comp_eval_if_ok_fexpr_expr comp_eval_if_ok_fexpr stmt_eval_if_ok_fexpr expr_eval_if_ok_fexpr composition_assoc_expression_rewrite composition_assoc_expression_rewrite_stmt stmt_eval_if_ok_fexpr_expr expr_eval_if_ok_fexpr_expr ok_result_fexpr ok_result_fstmt composition_assoc_rewrite_stmt_ok_cstmt

Installing automatic rewrites from: allocate_stack deallocate_stack with_new_stackvar

Installing automatic rewrites from: ok_result_ok ok_result_data ok_result_q_ok ok_result_q_state ok_result_q_data ok_result_elimination_1 ok_result_elimination_2 ok_result_state

Installing automatic rewrites from: l2r integral_to_bool pointer_to_bool bool2int

Installing automatic rewrites from: comp_simple_expr_simple_expr_ok comp_simple_expr_simple_expr_data comp_simple_expr_simple_expr_state comp_simple_stmt_simple_expr_ok comp_simple_stmt_simple_expr_data comp_simple_stmt_simple_expr_state

Installing automatic rewrites from: floating_point?

Installing automatic rewrites from: cv_base_result2 cv_base_result_c cv_base_result_v cv_base_result_cv cv c v

Installing automatic rewrites from: uchar_range_values

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `search_terminates`.

Q.E.D.

C.198.30 Search_Example.search_read_first

Terse proof for `search_read_first`.

`search_read_first`:

{1} valid(Search_Pre, search, read_first)

Installing automatic rewrites from: valid Valid Search_Pre search read_first search_terminates

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Repeatedly Skolemizing and flattening,

Installing automatic rewrites from: mod_range size_of uidt er_plus_ax er_minus_ax er_div_ax er_times_ax er_neg_ax address_of_spec address_of_spec2 add_spec add_spec2 same_array_spec same_array_spec2 check_bounds_spec extended_real_superset_of_number_fields

Installing automatic rewrites from: literal id member postinc postdec deref arrow unary_minus unary_minus_unsigned unary_not unary_bitnot preinc predec UnaryExpressions.sizeof ptm_member ptm_arrow arrow times div div_float mod plus plus_ptr postinc_ptr preinc_ptr subscript minus minus_ptr postdec_ptr predec_ptr minus_ptr_ptr cmp lt gt le ge lt_ptr gt_ptr le_ptr ge_ptr cmp_pointer eq_bool eq not_equal not_equal_bool eq_ptr not_equal_ptr bitand bitxor bitor and_exp or_exp conditional_expr assign assign_times assign_div assign_mod assign_div_float assign_plus assign_plus_ptr assign_minus assign_minus_ptr assign_bitand assign_bitxor assign_bitor comma

Installing automatic rewrites from: plain_memory_write_ok_single plain_memory_read_data_ok plain_memory_write_ok_q_expr plain_memory_write_ok_q_stmt in_blessed_memory_rw_ro pm_q_prop_ok pm_q_prop_single_read pm_q_prop_single_write pm_q_prop_read_write_single pm_q_prop_read_write_other_single pm_q_prop_read_read_single plain_memory_read_write_other_single_data plain_memory_read_read plain_memory_read_write_res pm_q_prop_read_ok_expr pm_q_prop_read_ok_stmt pm_q_prop_read_pm_q_prop_expr pm_q_prop_read_pm_q_prop_stmt pm_q_prop_read_write_q_expr pm_q_prop_read_write_q_stmt plain_memory_read_write_q_data_expr plain_memory_read_write_q_data_stmt pm_q_prop_read_write_other_q_expr pm_q_prop_read_write_other_q_stmt pm_q_prop_read_read_q_expr pm_q_prop_read_read_q_stmt plain_memory_read_write_other_q_data_expr plain_memory_read_write_other_q_data_stmt plain_memory_read_read plain_memory_read_read_q_data_stmt pm_q_prop_ok_result_single pm_q_prop_ok_result_q_expr pm_q_prop_ok_result_q_stmt pm_q_prop_read_ok_result_single pm_q_prop_read_ok_result_q_expr pm_q_prop_read_ok_result_q_stmt pm_q_prop_e2s pm_q_prop_stmt_e2s pm_q_prop_read_e2s pm_q_prop_read_stmt_e2s pm_q_prop_read_ok_pm_q_prop_read_expr pm_q_prop_read_ok_pm_q_prop_read_stmt

Installing automatic rewrites from: if_else switch member break_catch_break break_break_stmt break_break_catch_continue break_break_catch_default break_break_lift_stmt break_break_case break_break_default break_break_stmt_break_break break_stmt_break break_stmt_catch_continue break_stmt_catch_default break_stmt_lift break_stmt_case break_stmt_default continue_catch_continue continue_break_stmt continue_break_catch_continue continue_break_catch_default continue_break_lift_stmt continue_break_case continue_break_default continue_continue stmt_continue_continue continue_stmt_continue continue_stmt_catch_continue continue_stmt_catch_default continue_stmt_lift continue_stmt_case continue_stmt_default switch_stmt stmt_switch_stmt switch_case_taken stmt_switch_case_taken switch_case_not_taken stmt_switch_case_not_taken switch_default stmt_switch_default switch_catch_break stmt_switch_catch_break switch_catch_continue stmt_switch_catch_continue switch_catch_default stmt_switch_catch_default default_stmt stmt_default_stmt default_case stmt_default_case default_default stmt_default_default default_catch_break stmt_default_catch_break default_catch_continue stmt_default_catch_continue default_catch_default stmt_default_catch_default return_void_result return_ex_result stmt_remove_catch_default stmt_remove_case stmt_remove_default stmt_remove_catch_continue stmt_remove_catch_break break_break_stmt_expr break_break_catch_continue_expr break_break_catch_default_expr break_break_lift_stmt_expr break_break_case_expr break_break_default_expr continue_break_stmt_expr continue_break_catch_continue_expr continue_break_catch_default_expr continue_break_lift_stmt_expr continue_break_case_expr continue_break_default_expr

Installing automatic rewrites from: skip_ok skip_state skip_elimination stmt_skip_elimination stmt_ok_lift stmt_ok_lift_expr e2s_ok stmt_e2s_ok e2s_state stmt_e2s_state ok_result_fexpr ok_result_fstmt e2s_merge stmt_e2s_merge e2s_expr stmt_e2s_expr

Installing automatic rewrites from: while_unroll stmt_while_unroll iterate_while_ok while_hang while_unroll_expr while_unroll_lexpr stmt_while_unroll_expr stmt_while_unroll_lexpr while_unroll_0 do_while_unroll do_while_inv_unroll for_unroll for_inv_unroll while_to_while_no_cb while_invariant?! while_variant?! while_inv_rewrite_termination_result! while_inv_rewrite_data_ok! while_inv_rewrite_data_break! while_inv_rewrite_data_return! stmt_while_inv_rewrite_termination_result! stmt_while_inv_rewrite_data_ok! stmt_while_inv_rewrite_data_break! stmt_while_inv_rewrite_data_return! pm_q_prop_while! pm_q_prop_stmt_while!

Installing automatic rewrites from: composition_assoc_rewrite_stmt_ok forward_s_c_s_of_s_expr comp_eval_if_ok_fstmt stmt_eval_if_ok_fstmt comp_eval_if_ok_fstmt_cstmt stmt_eval_if_ok_fstmt_cstmt composition_assoc_rewrite_1 composition_assoc_rewrite_2 composition_assoc_rewrite_3 composition_assoc_rewrite_4 composition_assoc_rewrite_5 composition_assoc_rewrite_6 composition_assoc_rewrite_7 composition_assoc_rewrite_8 composition_assoc_rewrite_expr_1 composition_assoc_rewrite_expr_2 composition_assoc_rewrite_expr_3 composition_assoc_rewrite_expr_4 composition_assoc_rewrite_expr_5 composition_assoc_rewrite_expr_6 composition_assoc_rewrite_stmt_ok composition_assoc_rewrite_stmt_ok_expr composition_assoc_rewrite_stmt_ok_expr_lexpr comp_eval_if_ok_fstmt comp_eval_if_ok_fstmt_expr comp_eval_if_ok_fstmt_cstmt_expr stmt_eval_if_ok_fstmt_expr stmt_eval_if_ok_fstmt_cstmt_expr comp_eval_if_ok_fexpr_expr comp_eval_if_ok_fexpr stmt_eval_if_ok_fexpr expr_eval_if_ok_fexpr composition_assoc_expression_rewrite composition_assoc_expression_rewrite_stmt stmt_eval_if_ok_fexpr_expr expr_eval_if_ok_fexpr_expr ok_result_fexpr ok_result_fstmt composition_assoc_rewrite_stmt_ok_cstmt

Installing automatic rewrites from: allocate_stack deallocate_stack with_new_stackvar

Installing automatic rewrites from: ok_result_ok ok_result_data ok_result_q_ok ok_result_q_state ok_result_q_data ok_result_elimination_1 ok_result_elimination_2 ok_result_state

Installing automatic rewrites from: l2r integral_to_bool pointer_to_bool bool2int

Installing automatic rewrites from: comp_simple_expr_simple_expr_ok comp_simple_expr_simple_expr_data comp_simple_expr_simple_expr_state comp_simple_stmt_simple_expr_ok comp_simple_stmt_simple_expr_data comp_simple_stmt_simple_expr_state

Installing automatic rewrites from: floating_point?

Installing automatic rewrites from: cv_base_result2 cv_base_result_c cv_base_result_v cv_base_result_cv cv c v

Installing automatic rewrites from: uchar_range_values

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `search_read_first`.
 Q.E.D.

C.198.31 Search_Example.search_read_current

Terse proof for `search_read_current`.

`search_read_current`:

{1} valid(Search_Pre, search, read_current)

Installing automatic rewrites from: valid Valid Search_Pre search read_first read_current
 search_read_first search_terminates

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 Repeatedly Skolemizing and flattening,

Installing automatic rewrites from: mod_range size_of uidt er_plus_ax er_minus_ax er_div_ax
 er_times_ax er_neg_ax address_of_spec address_of_spec2 add_spec add_spec2 same_array_spec
 same_array_spec2 check_bounds_spec extended_real_superset_of_number_fields

Installing automatic rewrites from: literal id member postinc postdec deref arrow unary_minus
 unary_minus_unsigned unary_not unary_bitnot preinc predec UnaryExpressions.sizeof ptm_member
 ptm_arrow arrow times div div_float mod plus plus_ptr postinc_ptr preinc_ptr subscript minus minus_ptr
 postdec_ptr predec_ptr minus_ptr_ptr cmp lt gt le ge lt_ptr gt_ptr le_ptr ge_ptr cmp_pointer eq_bool
 eq not_equal not_equal_bool eq_ptr not_equal_ptr bitand bitxor bitor and_exp or_exp conditional_exp
 assign assign_times assign_div assign_mod assign_div_float assign_plus assign_plus_ptr assign_minus
 assign_minus_ptr assign_bitand assign_bitxor assign_bitor comma

Installing automatic rewrites from: plain_memory_write_ok_single plain_memory_read_data_ok
 plain_memory_write_ok_q_expr plain_memory_write_ok_q_stmt in_blessed_memory_rw_ro pm_q_prop_ok
 pm_q_prop_single_read pm_q_prop_single_write pm_q_prop_read_write_single pm_q_prop_read_write_other_single
 pm_q_prop_read_read_single plain_memory_read_write_other_single_data plain_memory_read_read
 plain_memory_read_write_res pm_q_prop_read_ok_expr pm_q_prop_read_ok_stmt pm_q_prop_read_pm_q_prop_expr
 pm_q_prop_read_pm_q_prop_stmt pm_q_prop_read_write_q_expr pm_q_prop_read_write_q_stmt
 plain_memory_read_write_q_data_expr plain_memory_read_write_q_data_stmt pm_q_prop_read_write_other_q_expr
 pm_q_prop_read_write_other_q_stmt pm_q_prop_read_read_q_expr pm_q_prop_read_read_q_stmt
 plain_memory_read_write_other_q_data_expr plain_memory_read_write_other_q_data_stmt plain_memory_read_read
 plain_memory_read_read_q_data_stmt pm_q_prop_ok_result_single pm_q_prop_ok_result_q_expr
 pm_q_prop_ok_result_q_stmt pm_q_prop_read_ok_result_single pm_q_prop_read_ok_result_q_expr
 pm_q_prop_read_ok_result_q_stmt pm_q_prop_e2s pm_q_prop_stmt_e2s pm_q_prop_read_e2s
 pm_q_prop_read_stmt_e2s pm_q_prop_read_ok_pm_q_prop_read_expr pm_q_prop_read_ok_pm_q_prop_read_stmt

Installing automatic rewrites from: if_else switch member break_catch_break break_break_stmt
 break_break_catch_continue break_break_catch_default break_break_lift_stmt break_break_case
 break_break_default break_break_stmt_break_break break_stmt_break break_stmt_catch_continue
 break_stmt_catch_default break_stmt_lift break_stmt_case break_stmt_default continue_catch_continue
 continue_break_stmt continue_break_catch_continue continue_break_catch_default continue_break_lift_stmt
 continue_break_case continue_break_default continue_continue stmt_continue_continue con-
 tinue_stmt_continue continue_stmt_catch_continue continue_stmt_catch_default continue_stmt_lift
 continue_stmt_case continue_stmt_default switch_stmt stmt_switch_stmt switch_case_taken
 stmt_switch_case_taken switch_case_not_taken stmt_switch_case_not_taken switch_default stmt_switch_default
 switch_catch_break stmt_switch_catch_break switch_catch_continue stmt_switch_catch_continue
 switch_catch_default stmt_switch_catch_default default_stmt stmt_default_stmt default_case
 stmt_default_case default_default stmt_default_default default_catch_break stmt_default_catch_break
 default_catch_continue stmt_default_catch_continue default_catch_default stmt_default_catch_default
 return_void_result return_ex_result stmt_remove_catch_default stmt_remove_case stmt_remove_default
 stmt_remove_catch_continue stmt_remove_catch_break break_break_stmt_expr break_break_catch_continue_expr

break_break_catch_default_expr break_break_lift_stmt_expr break_break_case_expr break_break_default_expr
 continue_break_stmt_expr continue_break_catch_continue_expr continue_break_catch_default_expr con-
 tinue_break_lift_stmt_expr continue_break_case_expr continue_break_default_expr

Installing automatic rewrites from: skip_ok skip_state skip_elimination stmt_skip_elimination
 stmt_ok_lift stmt_ok_lift_expr e2s_ok stmt_e2s_ok e2s_state stmt_e2s_state ok_result_fexpr
 ok_result_fstmt e2s_merge stmt_e2s_merge e2s_expr stmt_e2s_expr

Installing automatic rewrites from: while_unroll stmt_while_unroll iterate_while_ok while_hang
 while_unroll_expr while_unroll_lexpr stmt_while_unroll_expr stmt_while_unroll_lexpr while_unroll_0
 do_while_unroll do_while_inv_unroll for_unroll for_inv_unroll while_to_while_no_cb while_invariant?!
 while_variant?! while_inv_rewrite_termination_result! while_inv_rewrite_data_ok! while_inv_rewrite_data_break!
 while_inv_rewrite_data_return! stmt_while_inv_rewrite_termination_result! stmt_while_inv_rewrite_data_ok!
 stmt_while_inv_rewrite_data_break! stmt_while_inv_rewrite_data_return! pm_q_prop_while!
 pm_q_prop_stmt_while!

Installing automatic rewrites from: composition_assoc_rewrite_stmt_ok forward_s_c_s_of_s_expr
 comp_eval_if_ok_fstmt stmt_eval_if_ok_fstmt comp_eval_if_ok_fstmt_cstmt stmt_eval_if_ok_fstmt_cstmt
 composition_assoc_rewrite_1 composition_assoc_rewrite_2 composition_assoc_rewrite_3 com-
 position_assoc_rewrite_4 composition_assoc_rewrite_5 composition_assoc_rewrite_6 composi-
 tion_assoc_rewrite_7 composition_assoc_rewrite_8 composition_assoc_rewrite_expr_1 composi-
 tion_assoc_rewrite_expr_2 composition_assoc_rewrite_expr_3 composition_assoc_rewrite_expr_4 composi-
 tion_assoc_rewrite_expr_5 composition_assoc_rewrite_expr_6 composition_assoc_rewrite_stmt_ok compo-
 sition_assoc_rewrite_stmt_ok_expr composition_assoc_rewrite_stmt_ok_expr_lexpr comp_eval_if_ok_fstmt
 comp_eval_if_ok_fstmt_expr comp_eval_if_ok_fstmt_cstmt_expr stmt_eval_if_ok_fstmt_expr stmt_eval_if_ok_fstmt_cstmt_expr
 comp_eval_if_ok_fexpr_expr comp_eval_if_ok_fexpr stmt_eval_if_ok_fexpr expr_eval_if_ok_fexpr compo-
 sition_assoc_expression_rewrite composition_assoc_expression_rewrite_stmt stmt_eval_if_ok_fexpr_expr
 expr_eval_if_ok_fexpr_expr ok_result_fexpr ok_result_fstmt composition_assoc_rewrite_stmt_ok_cstmt

Installing automatic rewrites from: allocate_stack deallocate_stack with_new_stackvar

Installing automatic rewrites from: ok_result_ok ok_result_data ok_result_q_ok ok_result_q_state
 ok_result_q_data ok_result_elimination_1 ok_result_elimination_2 ok_result_state

Installing automatic rewrites from: l2r integral_to_bool pointer_to_bool bool2int

Installing automatic rewrites from: comp_simple_expr_simple_expr_ok comp_simple_expr_simple_expr_data
 comp_simple_expr_simple_expr_state comp_simple_stmt_simple_expr_ok comp_simple_stmt_simple_expr_data
 comp_simple_stmt_simple_expr_state

Installing automatic rewrites from: floating_point?

Installing automatic rewrites from: cv_base_result2 cv_base_result_c cv_base_result_v
 cv_base_result_cv cv c v

Installing automatic rewrites from: uchar_range_values

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `search_read_current`.

Q.E.D.

C.198.32 Search_Example.search_deref_current

Terse proof for `search_deref_current`.

`search_deref_current`:

{1}	valid(Search_Pre, search, deref_current)
-----	--

Installing automatic rewrites from: valid Valid Search_Pre N search read_first deref_current
 search_read_current read_current search_terminates search_read_first

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Repeatedly Skolemizing and flattening,

C Proof scripts

Installing automatic rewrites from: mod_range size_of uidt er_plus_ax er_minus_ax er_div_ax er_times_ax er_neg_ax address_of_spec address_of_spec2 add_spec add_spec2 address_add_zero address_add_sum same_array_spec same_array_spec2 check_bounds_spec extended_real_superset_of_number_fields

Installing automatic rewrites from: literal id member postinc postdec deref arrow unary_minus unary_minus_unsigned unary_not unary_bitnot preinc predec UnaryExpressions.sizeof ptm_member ptm_arrow arrow times div div_float mod plus plus_ptr postinc_ptr preinc_ptr subscript minus minus_ptr postdec_ptr predec_ptr minus_ptr_ptr cmp lt gt le ge lt_ptr gt_ptr le_ptr ge_ptr cmp_pointer eq_bool eq not_equal not_equal_bool eq_ptr not_equal_ptr bitand bitxor bitor and_exp or_exp conditional_exp assign assign_times assign_div assign_mod assign_div_float assign_plus assign_plus_ptr assign_minus assign_minus_ptr assign_bitand assign_bitxor assign_bitor comma

Installing automatic rewrites from: plain_memory_write_ok_single plain_memory_read_data_ok plain_memory_write_ok_q_expr plain_memory_write_ok_q_stmt in_blessed_memory_rw_ro pm_q_prop_ok pm_q_prop_single_read pm_q_prop_single_write pm_q_prop_read_write_single pm_q_prop_read_write_other_single pm_q_prop_read_read_single plain_memory_read_write_other_single_data plain_memory_read_read plain_memory_read_write_res pm_q_prop_read_ok_expr pm_q_prop_read_ok_stmt pm_q_prop_read_pm_q_prop_expr pm_q_prop_read_pm_q_prop_stmt pm_q_prop_read_write_q_expr pm_q_prop_read_write_q_stmt plain_memory_read_read_q_data_expr plain_memory_read_write_q_data_stmt pm_q_prop_read_write_other_q_expr pm_q_prop_read_write_other_q_stmt pm_q_prop_read_read_q_expr pm_q_prop_read_read_q_stmt plain_memory_read_write_other_q_data_expr plain_memory_read_write_other_q_data_stmt plain_memory_read_read plain_memory_read_read_q_data_stmt pm_q_prop_ok_result_single pm_q_prop_ok_result_q_expr pm_q_prop_ok_result_q_stmt pm_q_prop_read_ok_result_single pm_q_prop_read_ok_result_q_expr pm_q_prop_read_ok_result_q_stmt pm_q_prop_e2s pm_q_prop_stmt_e2s pm_q_prop_read_e2s pm_q_prop_read_stmt_e2s pm_q_prop_read_ok_pm_q_prop_read_expr pm_q_prop_read_ok_pm_q_prop_read_stmt

Installing automatic rewrites from: if_else switch member break_catch_break break_break_stmt break_break_catch_continue break_break_catch_default break_break_lift_stmt break_break_case break_break_default break_break_stmt_break_break break_stmt_break break_stmt_catch_continue break_stmt_catch_default break_stmt_lift break_stmt_case break_stmt_default continue_catch_continue continue_break_stmt continue_break_catch_continue continue_break_catch_default continue_break_lift_stmt continue_break_case continue_break_default continue_continue stmt_continue_continue continue_stmt_continue continue_stmt_catch_continue continue_stmt_catch_default continue_stmt_lift continue_stmt_case continue_stmt_default switch_stmt stmt_switch_stmt switch_case_taken stmt_switch_case_taken switch_case_not_taken stmt_switch_case_not_taken switch_default stmt_switch_default switch_catch_break stmt_switch_catch_break switch_catch_continue stmt_switch_catch_continue switch_catch_default stmt_switch_catch_default default_stmt default_stmt default_case stmt_default_case default_default stmt_default_default default_catch_break stmt_default_catch_break default_catch_continue stmt_default_catch_continue default_catch_default stmt_default_catch_default return_void_result return_ex_result stmt_remove_catch_default stmt_remove_case stmt_remove_default stmt_remove_catch_continue stmt_remove_catch_break break_break_stmt_expr break_break_catch_continue_expr break_break_catch_default_expr break_break_lift_stmt_expr break_break_case_expr break_break_default_expr continue_break_stmt_expr continue_break_catch_continue_expr continue_break_catch_default_expr continue_break_lift_stmt_expr continue_break_case_expr continue_break_default_expr

Installing automatic rewrites from: skip_ok skip_state skip_elimination stmt_skip_elimination stmt_ok_lift stmt_ok_lift_expr e2s_ok stmt_e2s_ok e2s_state stmt_e2s_state ok_result_fexpr ok_result_fstmt e2s_merge stmt_e2s_merge e2s_expr stmt_e2s_expr

Installing automatic rewrites from: while_unroll stmt_while_unroll iterate_while_ok while_hang while_unroll_expr while_unroll_lexpr stmt_while_unroll_expr stmt_while_unroll_lexpr while_unroll_0 do_while_unroll do_while_inv_unroll for_unroll for_inv_unroll while_to_while_no_cb while_invariant?! while_variant?! while_inv_rewrite_termination_result! while_inv_rewrite_data_ok! while_inv_rewrite_data_break! while_inv_rewrite_data_return! stmt_while_inv_rewrite_termination_result! stmt_while_inv_rewrite_data_ok! stmt_while_inv_rewrite_data_break! stmt_while_inv_rewrite_data_return! pm_q_prop_while! pm_q_prop_stmt_while!

Installing automatic rewrites from: composition_assoc_rewrite_stmt_ok forward_s_c_s_of_s_expr

comp_eval_if_ok_fstmt stmt_eval_if_ok_fstmt comp_eval_if_ok_fstmt_cstmt stmt_eval_if_ok_fstmt_cstmt
 composition_assoc_rewrite_1 composition_assoc_rewrite_2 composition_assoc_rewrite_3 composition_assoc_rewrite_4
 composition_assoc_rewrite_5 composition_assoc_rewrite_6 composition_assoc_rewrite_7 composition_assoc_rewrite_8
 composition_assoc_rewrite_expr_1 composition_assoc_rewrite_expr_2 composition_assoc_rewrite_expr_3
 composition_assoc_rewrite_expr_4 composition_assoc_rewrite_expr_5 composition_assoc_rewrite_expr_6
 composition_assoc_rewrite_stmt_ok composition_assoc_rewrite_stmt_ok_expr
 composition_assoc_rewrite_stmt_ok_expr_lexpr comp_eval_if_ok_fstmt
 comp_eval_if_ok_fstmt_expr comp_eval_if_ok_fstmt_cstmt_expr stmt_eval_if_ok_fstmt_expr
 stmt_eval_if_ok_fstmt_cstmt_expr comp_eval_if_ok_fexpr_expr comp_eval_if_ok_fexpr
 stmt_eval_if_ok_fexpr expr_eval_if_ok_fexpr composition_assoc_expression_rewrite
 composition_assoc_expression_rewrite_stmt stmt_eval_if_ok_fexpr_expr
 expr_eval_if_ok_fexpr_expr ok_result_fexpr ok_result_fstmt composition_assoc_rewrite_stmt_ok_cstmt

Installing automatic rewrites from: allocate_stack deallocate_stack with_new_stackvar

Installing automatic rewrites from: ok_result_ok ok_result_data ok_result_q_ok ok_result_q_state
 ok_result_q_data ok_result_elimination_1 ok_result_elimination_2 ok_result_state

Installing automatic rewrites from: l2r integral_to_bool pointer_to_bool bool2int

Installing automatic rewrites from: comp_simple_expr_simple_expr_ok comp_simple_expr_simple_expr_data
 comp_simple_expr_simple_expr_state comp_simple_stmt_simple_expr_ok comp_simple_stmt_simple_expr_data
 comp_simple_stmt_simple_expr_state

Installing automatic rewrites from: floating_point?

Installing automatic rewrites from: cv_base_result2 cv_base_result_c cv_base_result_v
 cv_base_result_cv cv c v

Installing automatic rewrites from: uchar_range_values

Instantiating the top quantifier in -14 with the terms: (0),

Instantiating the top quantifier in -15 with the terms: (0),

Instantiating the top quantifier in -16 with the terms: (0),

Instantiating the top quantifier in -17 with the terms: (0),

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `search_deref_current`.

Q.E.D.

C.198.33 Search_Example.search_read_last

Terse proof for `search_read_last`.

`search_read_last`:

{1}	valid(Search_Pre, search, read_last)
-----	--------------------------------------

Installing automatic rewrites from: valid Valid Search_Pre search read_first read_current
 search_read_first deref_current read_last search_current search_deref_current search_terminates N

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Repeatedly Skolemizing and flattening,

Installing automatic rewrites from: mod_range size_of uidt er_plus_ax er_minus_ax er_div_ax
 er_times_ax er_neg_ax address_of_spec address_of_spec2 add_spec add_spec2 same_array_spec
 same_array_spec2 check_bounds_spec extended_real_superset_of_number_fields

Installing automatic rewrites from: literal id member postinc postdec deref arrow unary_minus
 unary_minus_unsigned unary_not unary_bitnot preinc predec UnaryExpressions.sizeof ptm_member
 ptm_arrow arrow times div div_float mod plus plus_ptr postinc_ptr preinc_ptr subscript minus minus_ptr
 postdec_ptr predec_ptr minus_ptr_ptr cmp lt gt le ge lt_ptr gt_ptr le_ptr ge_ptr cmp_pointer eq_bool
 eq not_equal not_equal_bool eq_ptr not_equal_ptr bitand bitxor bitor and_exp or_exp conditional_exp
 assign assign_times assign_div assign_mod assign_div_float assign_plus assign_plus_ptr assign_minus
 assign_minus_ptr assign_bitand assign_bitxor assign_bitor comma

C Proof scripts

Installing automatic rewrites from: plain_memory_write_ok_single plain_memory_read_data_ok plain_memory_write_ok_q_expr plain_memory_write_ok_q_stmt in_blessed_memory_rw_ro pm_q_prop_ok pm_q_prop_single_read pm_q_prop_single_write pm_q_prop_read_write_single pm_q_prop_read_write_other_single pm_q_prop_read_read_single plain_memory_read_write_other_single_data plain_memory_read_read plain_memory_read_write_res pm_q_prop_read_ok_expr pm_q_prop_read_ok_stmt pm_q_prop_read_pm_q_prop_expr pm_q_prop_read_pm_q_prop_stmt pm_q_prop_read_write_q_expr pm_q_prop_read_write_q_stmt plain_memory_read_write_q_data_expr plain_memory_read_write_q_data_stmt pm_q_prop_read_write_other_q_expr pm_q_prop_read_write_other_q_stmt pm_q_prop_read_read_q_expr pm_q_prop_read_read_q_stmt plain_memory_read_write_other_q_data_expr plain_memory_read_write_other_q_data_stmt plain_memory_read_read plain_memory_read_read_q_data_stmt pm_q_prop_ok_result_single pm_q_prop_ok_result_q_expr pm_q_prop_ok_result_q_stmt pm_q_prop_read_ok_result_single pm_q_prop_read_ok_result_q_expr pm_q_prop_read_ok_result_q_stmt pm_q_prop_e2s pm_q_prop_stmt_e2s pm_q_prop_read_e2s pm_q_prop_read_stmt_e2s pm_q_prop_read_ok_pm_q_prop_read_expr pm_q_prop_read_ok_pm_q_prop_read_stmt

Installing automatic rewrites from: if_else switch member break_catch_break break_break_stmt break_break_catch_continue break_break_catch_default break_break_lift_stmt break_break_case break_break_default break_break_stmt_break_break break_stmt_break break_stmt_catch_continue break_stmt_catch_default break_stmt_lift break_stmt_case break_stmt_default continue_catch_continue continue_break_stmt continue_break_catch_continue continue_break_catch_default continue_break_lift_stmt continue_break_case continue_break_default continue_continue stmt_continue_continue continue_stmt_continue continue_stmt_catch_continue continue_stmt_catch_default continue_stmt_lift continue_stmt_case continue_stmt_default switch_stmt stmt_switch_stmt switch_case_taken stmt_switch_case_taken switch_case_not_taken stmt_switch_case_not_taken switch_default stmt_switch_default switch_catch_break stmt_switch_catch_break switch_catch_continue stmt_switch_catch_continue switch_catch_default stmt_switch_catch_default default_stmt stmt_default_stmt default_case stmt_default_case default_default stmt_default_default default_catch_break stmt_default_catch_break default_catch_continue stmt_default_catch_continue default_catch_default stmt_default_catch_default return_void_result return_ex_result stmt_remove_catch_default stmt_remove_case stmt_remove_default stmt_remove_catch_continue stmt_remove_catch_break break_break_stmt_expr break_break_catch_continue_expr break_break_catch_default_expr break_break_lift_stmt_expr break_break_case_expr break_break_default_expr continue_break_stmt_expr continue_break_catch_continue_expr continue_break_catch_default_expr continue_break_lift_stmt_expr continue_break_case_expr continue_break_default_expr

Installing automatic rewrites from: skip_ok skip_state skip_elimination stmt_skip_elimination stmt_ok_lift stmt_ok_lift_expr e2s_ok stmt_e2s_ok e2s_state stmt_e2s_state ok_result_fexpr ok_result_fstmt e2s_merge stmt_e2s_merge e2s_expr stmt_e2s_expr

Installing automatic rewrites from: while_unroll stmt_while_unroll iterate_while_ok while_hang while_unroll_expr while_unroll_lexpr stmt_while_unroll_expr stmt_while_unroll_lexpr while_unroll_0 do_while_unroll do_while_inv_unroll for_unroll for_inv_unroll while_to_while_no_cb while_invariant?! while_variant?! while_inv_rewrite_termination_result! while_inv_rewrite_data_ok! while_inv_rewrite_data_break! while_inv_rewrite_data_return! stmt_while_inv_rewrite_termination_result! stmt_while_inv_rewrite_data_ok! stmt_while_inv_rewrite_data_break! stmt_while_inv_rewrite_data_return! pm_q_prop_while! pm_q_prop_stmt_while!

Installing automatic rewrites from: composition_assoc_rewrite_stmt_ok forward_s_c_s_of_s_expr comp_eval_if_ok_fstmt stmt_eval_if_ok_fstmt comp_eval_if_ok_fstmt_cstmt stmt_eval_if_ok_fstmt_cstmt composition_assoc_rewrite_1 composition_assoc_rewrite_2 composition_assoc_rewrite_3 composition_assoc_rewrite_4 composition_assoc_rewrite_5 composition_assoc_rewrite_6 composition_assoc_rewrite_7 composition_assoc_rewrite_8 composition_assoc_rewrite_expr_1 composition_assoc_rewrite_expr_2 composition_assoc_rewrite_expr_3 composition_assoc_rewrite_expr_4 composition_assoc_rewrite_expr_5 composition_assoc_rewrite_expr_6 composition_assoc_rewrite_stmt_ok composition_assoc_rewrite_stmt_ok_expr composition_assoc_rewrite_stmt_ok_expr_lexpr comp_eval_if_ok_fstmt comp_eval_if_ok_fstmt_expr comp_eval_if_ok_fstmt_cstmt_expr stmt_eval_if_ok_fstmt_expr stmt_eval_if_ok_fstmt_cstmt_expr comp_eval_if_ok_fexpr_expr comp_eval_if_ok_fexpr stmt_eval_if_ok_fexpr expr_eval_if_ok_fexpr composition_assoc_expression_rewrite composition_assoc_expression_rewrite_stmt stmt_eval_if_ok_fexpr_expr

expr_eval_if_ok_fexpr_expr ok_result_fexpr ok_result_fstmt composition_assoc_rewrite_stmt_ok_cstmt

Installing automatic rewrites from: allocate_stack deallocate_stack with_new_stackvar

Installing automatic rewrites from: ok_result_ok ok_result_data ok_result_q_ok ok_result_q_state
ok_result_q_data ok_result_elimination_1 ok_result_elimination_2 ok_result_state

Installing automatic rewrites from: l2r integral_to_bool pointer_to_bool bool2int

Installing automatic rewrites from: comp_simple_expr_simple_expr_ok comp_simple_expr_simple_expr_data
comp_simple_expr_simple_expr_state comp_simple_stmt_simple_expr_ok comp_simple_stmt_simple_expr_data
comp_simple_stmt_simple_expr_state

Installing automatic rewrites from: floating_point?

Installing automatic rewrites from: cv_base_result2 cv_base_result_c cv_base_result_v
cv_base_result_cv cv c v

Installing automatic rewrites from: uchar_range_values

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `search_read_last`.

Q.E.D.

C.198.34 Search_Example.search_read_array

Terse proof for `search_read_array`.

`search_read_array`:

	{1}	valid(Search_Pre, search, read_array)
--	-----	---------------------------------------

Installing automatic rewrites from: valid Valid Search_Pre search read_first read_current
search_read_first deref_current read_last search_current search_deref_current search_read_last read_array
search_terminates N

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Repeatedly Skolemizing and flattening,

Installing automatic rewrites from: mod_range size_of uidt er_plus_ax er_minus_ax er_div_ax
er_times_ax er_neg_ax address_of_spec address_of_spec2 add_spec add_spec2 same_array_spec
same_array_spec2 check_bounds_spec extended_real_superset_of_number_fields

Installing automatic rewrites from: literal id member postinc postdec deref arrow unary_minus
unary_minus_unsigned unary_not unary_bitnot preinc predec UnaryExpressions.sizeof ptm_member
ptm_arrow arrow times div div_float mod plus plus_ptr postinc_ptr preinc_ptr subscript minus minus_ptr
postdec_ptr predec_ptr minus_ptr_ptr cmp lt gt le ge lt_ptr gt_ptr le_ptr ge_ptr cmp_pointer eq_bool
eq not_equal not_equal_bool eq_ptr not_equal_ptr bitand bitxor bitor and_exp or_exp conditional_exp
assign assign_times assign_div assign_mod assign_div_float assign_plus assign_plus_ptr assign_minus
assign_minus_ptr assign_bitand assign_bitxor assign_bitor comma

Installing automatic rewrites from: plain_memory_write_ok_single plain_memory_read_data_ok
plain_memory_write_ok_q_expr plain_memory_write_ok_q_stmt in_blessed_memory_rw_ro pm_q_prop_ok
pm_q_prop_single_read pm_q_prop_single_write pm_q_prop_read_write_single pm_q_prop_read_write_other_single
pm_q_prop_read_read_single plain_memory_read_write_other_single_data plain_memory_read_read
plain_memory_read_write_res pm_q_prop_read_ok_expr pm_q_prop_read_ok_stmt pm_q_prop_read_pm_q_prop_expr
pm_q_prop_read_pm_q_prop_stmt pm_q_prop_read_write_q_expr pm_q_prop_read_write_q_stmt
plain_memory_read_write_q_data_expr plain_memory_read_write_q_data_stmt pm_q_prop_read_write_other_q_expr
pm_q_prop_read_write_other_q_stmt pm_q_prop_read_read_q_expr pm_q_prop_read_read_q_stmt
plain_memory_read_write_other_q_data_expr plain_memory_read_write_other_q_data_stmt plain_memory_read_read_q_data_exp
plain_memory_read_read_q_data_stmt pm_q_prop_ok_result_single pm_q_prop_ok_result_q_expr
pm_q_prop_ok_result_q_stmt pm_q_prop_read_ok_result_single pm_q_prop_read_ok_result_q_expr
pm_q_prop_read_ok_result_q_stmt pm_q_prop_e2s pm_q_prop_stmt_e2s pm_q_prop_read_e2s
pm_q_prop_read_stmt_e2s pm_q_prop_read_ok_pm_q_prop_read_expr pm_q_prop_read_ok_pm_q_prop_read_stmt

C Proof scripts

Installing automatic rewrites from: if_else switch member break_catch_break break_break_stmt break_break_catch_continue break_break_catch_default break_break_lift_stmt break_break_case break_break_default break_break_stmt_break_break break_stmt_break break_stmt_catch_continue break_stmt_catch_default break_stmt_lift break_stmt_case break_stmt_default continue_catch_continue continue_break_stmt continue_break_catch_continue continue_break_catch_default continue_break_lift_stmt continue_break_case continue_break_default continue_continue stmt_continue_continue continue_stmt_continue continue_stmt_catch_continue continue_stmt_catch_default continue_stmt_lift continue_stmt_case continue_stmt_default switch_stmt stmt_switch_stmt switch_case_taken stmt_switch_case_taken switch_case_not_taken stmt_switch_case_not_taken switch_default stmt_switch_default switch_catch_break stmt_switch_catch_break switch_catch_continue stmt_switch_catch_continue switch_catch_default stmt_switch_catch_default default_stmt stmt_default_stmt default_case stmt_default_case default_default stmt_default_default default_catch_break stmt_default_catch_break default_catch_continue stmt_default_catch_continue default_catch_default stmt_default_catch_default return_void_result return_ex_result stmt_remove_catch_default stmt_remove_case stmt_remove_default stmt_remove_catch_continue stmt_remove_catch_break break_break_stmt_expr break_break_catch_continue_expr break_break_catch_default_expr break_break_lift_stmt_expr break_break_case_expr break_break_default_expr continue_break_stmt_expr continue_break_catch_continue_expr continue_break_catch_default_expr continue_break_lift_stmt_expr continue_break_case_expr continue_break_default_expr

Installing automatic rewrites from: skip_ok skip_state skip_elimination stmt_skip_elimination stmt_ok_lift stmt_ok_lift_expr e2s_ok stmt_e2s_ok e2s_state stmt_e2s_state ok_result_fexpr ok_result_fstmt e2s_merge stmt_e2s_merge e2s_expr stmt_e2s_expr

Installing automatic rewrites from: while_unroll stmt_while_unroll iterate_while_ok while_hang while_unroll_expr while_unroll_lexpr stmt_while_unroll_expr stmt_while_unroll_lexpr while_unroll_0 do_while_unroll do_while_inv_unroll for_unroll for_inv_unroll while_to_while_no_cb while_invariant?! while_variant?! while_inv_rewrite_termination_result! while_inv_rewrite_data_ok! while_inv_rewrite_data_break! while_inv_rewrite_data_return! stmt_while_inv_rewrite_termination_result! stmt_while_inv_rewrite_data_ok! stmt_while_inv_rewrite_data_break! stmt_while_inv_rewrite_data_return! pm_q_prop_while! pm_q_prop_stmt_while!

Installing automatic rewrites from: composition_assoc_rewrite_stmt_ok forward_s_c_s_of_s_expr comp_eval_if_ok_fstmt stmt_eval_if_ok_fstmt comp_eval_if_ok_fstmt_cstmt stmt_eval_if_ok_fstmt_cstmt composition_assoc_rewrite_1 composition_assoc_rewrite_2 composition_assoc_rewrite_3 composition_assoc_rewrite_4 composition_assoc_rewrite_5 composition_assoc_rewrite_6 composition_assoc_rewrite_7 composition_assoc_rewrite_8 composition_assoc_rewrite_expr_1 composition_assoc_rewrite_expr_2 composition_assoc_rewrite_expr_3 composition_assoc_rewrite_expr_4 composition_assoc_rewrite_expr_5 composition_assoc_rewrite_expr_6 composition_assoc_rewrite_stmt_ok composition_assoc_rewrite_stmt_ok_expr composition_assoc_rewrite_stmt_ok_expr_lexpr comp_eval_if_ok_fstmt comp_eval_if_ok_fstmt_expr comp_eval_if_ok_fstmt_cstmt_expr stmt_eval_if_ok_fstmt_expr stmt_eval_if_ok_fstmt_cstmt_expr comp_eval_if_ok_fexpr_expr comp_eval_if_ok_fexpr stmt_eval_if_ok_fexpr expr_eval_if_ok_fexpr composition_assoc_expression_rewrite composition_assoc_expression_rewrite_stmt stmt_eval_if_ok_fexpr_expr expr_eval_if_ok_fexpr_expr ok_result_fexpr ok_result_fstmt composition_assoc_rewrite_stmt_ok_cstmt

Installing automatic rewrites from: allocate_stack deallocate_stack with_new_stackvar

Installing automatic rewrites from: ok_result_ok ok_result_data ok_result_q_ok ok_result_q_state ok_result_q_data ok_result_elimination_1 ok_result_elimination_2 ok_result_state

Installing automatic rewrites from: l2r integral_to_bool pointer_to_bool bool2int

Installing automatic rewrites from: comp_simple_expr_simple_expr_ok comp_simple_expr_simple_expr_data comp_simple_expr_simple_expr_state comp_simple_stmt_simple_expr_ok comp_simple_stmt_simple_expr_data comp_simple_stmt_simple_expr_state

Installing automatic rewrites from: floating_point?

Installing automatic rewrites from: cv_base_result2 cv_base_result_c cv_base_result_v cv_base_result_cv cv c v

Installing automatic rewrites from: uchar_range_values

Repeatedly simplifying with decision procedures, rewriting, propositional reasoning, quantifier in-

stantiation, skolemization, if-lifting and equality replacement,
 This completes the proof of search_read_array.
 Q.E.D.

C.198.35 Search_Example.search_not_found

Terse proof for search_not_found.

search_not_found:

	{1} valid(Search_Pre, search, not_found)
--	--

Installing automatic rewrites from: valid Valid Search_Pre search read_first read_current search_read_first deref_current read_last search_current search_deref_current search_read_last read_array search_read_array not_found search_terminates N

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 Repeatedly Skolemizing and flattening,

Installing automatic rewrites from: mod_range size_of uidt er_plus_ax er_minus_ax er_div_ax er_times_ax er_neg_ax address_of_spec address_of_spec2 add_spec add_spec2 same_array_spec same_array_spec2 check_bounds_spec extended_real_superset_of_number_fields

Installing automatic rewrites from: literal id member postinc postdec deref arrow unary_minus unary_minus_unsigned unary_not unary_bitnot preinc predec UnaryExpressions.sizeof ptm_member ptm_arrow arrow times div div_float mod plus plus_ptr postinc_ptr preinc_ptr subscript minus minus_ptr postdec_ptr predec_ptr minus_ptr_ptr cmp lt gt le ge lt_ptr gt_ptr le_ptr ge_ptr cmp_pointer eq_bool eq not_equal not_equal_bool eq_ptr not_equal_ptr bitand bitxor bitor and_exp or_exp conditional_expr assign assign_times assign_div assign_mod assign_div_float assign_plus assign_plus_ptr assign_minus assign_minus_ptr assign_bitand assign_bitxor assign_bitor comma

Installing automatic rewrites from: plain_memory_write_ok_single plain_memory_read_data_ok plain_memory_write_ok_q_expr plain_memory_write_ok_q_stmt in_blessed_memory_rw_ro pm_q_prop_ok pm_q_prop_single_read pm_q_prop_single_write pm_q_prop_read_write_single pm_q_prop_read_write_other_single pm_q_prop_read_read_single plain_memory_read_write_other_single_data plain_memory_read_read plain_memory_read_write_res pm_q_prop_read_ok_expr pm_q_prop_read_ok_stmt pm_q_prop_read_pm_q_prop_expr pm_q_prop_read_pm_q_prop_stmt pm_q_prop_read_write_q_expr pm_q_prop_read_write_q_stmt plain_memory_read_write_q_data_expr plain_memory_read_write_q_data_stmt pm_q_prop_read_write_other_q_expr pm_q_prop_read_write_other_q_stmt pm_q_prop_read_read_q_expr pm_q_prop_read_read_q_stmt plain_memory_read_write_other_q_data_expr plain_memory_read_write_other_q_data_stmt plain_memory_read_read_q_data_expr plain_memory_read_read_q_data_stmt pm_q_prop_ok_result_single pm_q_prop_ok_result_q_expr pm_q_prop_ok_result_q_stmt pm_q_prop_read_ok_result_single pm_q_prop_read_ok_result_q_expr pm_q_prop_read_ok_result_q_stmt pm_q_prop_e2s pm_q_prop_stmt_e2s pm_q_prop_read_e2s pm_q_prop_read_stmt_e2s pm_q_prop_read_ok_pm_q_prop_read_expr pm_q_prop_read_ok_pm_q_prop_read_stmt

Installing automatic rewrites from: if_else switch member break_catch_break break_break_stmt break_break_catch_continue break_break_catch_default break_break_lift_stmt break_break_case break_break_default break_break stmt_break_break break_stmt_break break_stmt_catch_continue break_stmt_catch_default break_stmt_lift break_stmt_case break_stmt_default continue_catch_continue continue_break_stmt continue_break_catch_continue continue_break_catch_default continue_break_lift_stmt continue_break_case continue_break_default continue_continue stmt_continue_continue continue_stmt_continue continue_stmt_catch_continue continue_stmt_catch_default continue_stmt_lift continue_stmt_case continue_stmt_default switch_stmt stmt_switch_stmt switch_case_taken stmt_switch_case_taken switch_case_not_taken stmt_switch_case_not_taken switch_default stmt_switch_default switch_catch_break stmt_switch_catch_break switch_catch_continue stmt_switch_catch_continue switch_catch_default stmt_switch_catch_default default_stmt stmt_default_stmt default_case stmt_default_case default_default stmt_default_default default_catch_break stmt_default_catch_break default_catch_continue stmt_default_catch_continue default_catch_default stmt_default_catch_default

return_void_result return_ex_result stmt_remove_catch_default stmt_remove_case stmt_remove_default
stmt_remove_catch_continue stmt_remove_catch_break break_break_stmt_expr break_break_catch_continue_expr
break_break_catch_default_expr break_break_lift_stmt_expr break_break_case_expr break_break_default_expr
continue_break_stmt_expr continue_break_catch_continue_expr continue_break_catch_default_expr con-
tinue_break_lift_stmt_expr continue_break_case_expr continue_break_default_expr

Installing automatic rewrites from: skip_ok skip_state skip_elimination stmt_skip_elimination
stmt_ok_lift stmt_ok_lift_expr e2s_ok stmt_e2s_ok e2s_state stmt_e2s_state ok_result_fexpr
ok_result_fstmt e2s_merge stmt_e2s_merge e2s_expr stmt_e2s_expr

Installing automatic rewrites from: while_unroll stmt_while_unroll iterate_while_ok while_hang
while_unroll_expr while_unroll_lexpr stmt_while_unroll_expr stmt_while_unroll_lexpr while_unroll_0
do_while_unroll do_while_inv_unroll for_unroll for_inv_unroll while_to_while_no_cb while_invariant?!
while_variant?! while_inv_rewrite_termination_result! while_inv_rewrite_data_ok! while_inv_rewrite_data_break!
while_inv_rewrite_data_return! stmt_while_inv_rewrite_termination_result! stmt_while_inv_rewrite_data_ok!
stmt_while_inv_rewrite_data_break! stmt_while_inv_rewrite_data_return! pm_q_prop_while!
pm_q_prop_stmt_while!

Installing automatic rewrites from: composition_assoc_rewrite_stmt_ok forward_s_c_s_of_s_expr
comp_eval_if_ok_fstmt stmt_eval_if_ok_fstmt comp_eval_if_ok_fstmt_cstmt stmt_eval_if_ok_fstmt_cstmt
composition_assoc_rewrite_1 composition_assoc_rewrite_2 composition_assoc_rewrite_3 com-
position_assoc_rewrite_4 composition_assoc_rewrite_5 composition_assoc_rewrite_6 composi-
tion_assoc_rewrite_7 composition_assoc_rewrite_8 composition_assoc_rewrite_expr_1 composi-
tion_assoc_rewrite_expr_2 composition_assoc_rewrite_expr_3 composition_assoc_rewrite_expr_4 composi-
tion_assoc_rewrite_expr_5 composition_assoc_rewrite_expr_6 composition_assoc_rewrite_stmt_ok compo-
sition_assoc_rewrite_stmt_ok_expr composition_assoc_rewrite_stmt_ok_expr_lexpr comp_eval_if_ok_fstmt
comp_eval_if_ok_fstmt_expr comp_eval_if_ok_fstmt_cstmt_expr stmt_eval_if_ok_fstmt_expr stmt_eval_if_ok_fstmt_cstm-
comp_eval_if_ok_fexpr_expr comp_eval_if_ok_fexpr stmt_eval_if_ok_fexpr expr_eval_if_ok_fexpr compo-
sition_assoc_expression_rewrite composition_assoc_expression_rewrite_stmt stmt_eval_if_ok_fexpr_expr
expr_eval_if_ok_fexpr_expr ok_result_fexpr ok_result_fstmt composition_assoc_rewrite_stmt_ok_cstmt

Installing automatic rewrites from: allocate_stack deallocate_stack with_new_stackvar

Installing automatic rewrites from: ok_result_ok ok_result_data ok_result_q_ok ok_result_q_state
ok_result_q_data ok_result_elimination_1 ok_result_elimination_2 ok_result_state

Installing automatic rewrites from: l2r integral_to_bool pointer_to_bool bool2int

Installing automatic rewrites from: comp_simple_expr_simple_expr_ok comp_simple_expr_simple_expr_data
comp_simple_expr_simple_expr_state comp_simple_stmt_simple_expr_ok comp_simple_stmt_simple_expr_data
comp_simple_stmt_simple_expr_state

Installing automatic rewrites from: floating_point?

Installing automatic rewrites from: cv_base_result2 cv_base_result_c cv_base_result_v
cv_base_result_cv cv c v

Installing automatic rewrites from: uchar_range_values

Repeatedly simplifying with decision procedures, rewriting, propositional reasoning, quantifier in-
stantiation, skolemization, if-lifting and equality replacement,

This completes the proof of `search_not_found`.

Q.E.D.

C.198.36 Search_Example.search_found

Terse proof for `search_found`.

`search_found`:

{1}	valid(Search_Pre, search, found)
-----	----------------------------------

Installing automatic rewrites from: valid Valid Search_Pre search read_first read_current
search_read_first deref_current read_last search_current search_deref_current search_read_last read_array

search_read_array not_found search_not_found found search_terminates N

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Repeatedly Skolemizing and flattening,

Installing automatic rewrites from: mod_range size_of uidt er_plus_ax er_minus_ax er_div_ax er_times_ax er_neg_ax address_of_spec address_of_spec2 add_spec add_spec2 same_array_spec same_array_spec2 check_bounds_spec extended_real_superset_of_number_fields

Installing automatic rewrites from: literal id member postinc postdec deref arrow unary_minus unary_minus_unsigned unary_not unary_bitnot preinc predec UnaryExpressions.sizeof ptm_member ptm_arrow arrow times div div_float mod plus plus_ptr postinc_ptr preinc_ptr subscript minus minus_ptr postdec_ptr predec_ptr minus_ptr_ptr cmp lt gt le ge lt_ptr gt_ptr le_ptr ge_ptr cmp_pointer eq_bool eq not_equal not_equal_bool eq_ptr not_equal_ptr bitand bitxor bitor and_exp or_exp conditional_exp assign assign_times assign_div assign_mod assign_div_float assign_plus assign_plus_ptr assign_minus assign_minus_ptr assign_bitand assign_bitxor assign_bitor comma

Installing automatic rewrites from: plain_memory_write_ok_single plain_memory_read_data_ok plain_memory_write_ok_q_expr plain_memory_write_ok_q_stmt in_blessed_memory_rw_ro pm_q_prop_ok pm_q_prop_single_read pm_q_prop_single_write pm_q_prop_read_write_single pm_q_prop_read_write_other_single pm_q_prop_read_read_single plain_memory_read_write_other_single_data plain_memory_read_read plain_memory_read_write_res pm_q_prop_read_ok_expr pm_q_prop_read_ok_stmt pm_q_prop_read_pm_q_prop_expr pm_q_prop_read_pm_q_prop_stmt pm_q_prop_read_write_q_expr pm_q_prop_read_write_q_stmt plain_memory_read_write_q_data_expr plain_memory_read_write_q_data_stmt pm_q_prop_read_write_other_q_expr pm_q_prop_read_write_other_q_stmt pm_q_prop_read_read_q_expr pm_q_prop_read_read_q_stmt plain_memory_read_write_other_q_data_expr plain_memory_read_write_other_q_data_stmt plain_memory_read_read_q_data_expr plain_memory_read_read_q_data_stmt pm_q_prop_ok_result_single pm_q_prop_ok_result_q_expr pm_q_prop_ok_result_q_stmt pm_q_prop_read_ok_result_single pm_q_prop_read_ok_result_q_expr pm_q_prop_read_ok_result_q_stmt pm_q_prop_e2s pm_q_prop_stmt_e2s pm_q_prop_read_e2s pm_q_prop_read_stmt_e2s pm_q_prop_read_ok_pm_q_prop_read_expr pm_q_prop_read_ok_pm_q_prop_read_stmt

Installing automatic rewrites from: if_else switch member break_catch_break break_break_stmt break_break_catch_continue break_break_catch_default break_break_lift_stmt break_break_case break_break_default break_break_stmt_break_break break_stmt_break break_stmt_catch_continue break_stmt_catch_default break_stmt_lift break_stmt_case break_stmt_default continue_catch_continue continue_break_stmt continue_break_catch_continue continue_break_catch_default continue_break_lift_stmt continue_break_case continue_break_default continue_continue stmt_continue_continue continue_stmt_continue continue_stmt_catch_continue continue_stmt_catch_default continue_stmt_lift continue_stmt_case continue_stmt_default switch_stmt stmt_switch_stmt switch_case_taken stmt_switch_case_taken switch_case_not_taken stmt_switch_case_not_taken switch_default stmt_switch_default switch_catch_break stmt_switch_catch_break switch_catch_continue stmt_switch_catch_continue switch_catch_default stmt_switch_catch_default default_stmt stmt_default_stmt default_case stmt_default_case default_default stmt_default_default default_catch_break stmt_default_catch_break default_catch_continue stmt_default_catch_continue default_catch_default stmt_default_catch_default return_void_result return_ex_result stmt_remove_catch_default stmt_remove_case stmt_remove_default stmt_remove_catch_continue stmt_remove_catch_break break_break_stmt_expr break_break_catch_continue_expr break_break_catch_default_expr break_break_lift_stmt_expr break_break_case_expr break_break_default_expr continue_break_stmt_expr continue_break_catch_continue_expr continue_break_catch_default_expr continue_break_lift_stmt_expr continue_break_case_expr continue_break_default_expr

Installing automatic rewrites from: skip_ok skip_state skip_elimination stmt_skip_elimination stmt_ok_lift stmt_ok_lift_expr e2s_ok stmt_e2s_ok e2s_state stmt_e2s_state ok_result_fexpr ok_result_fstmt e2s_merge stmt_e2s_merge e2s_expr stmt_e2s_expr

Installing automatic rewrites from: while_unroll stmt_while_unroll iterate_while_ok while_hang while_unroll_expr while_unroll_lexpr stmt_while_unroll_expr stmt_while_unroll_lexpr while_unroll_0 do_while_unroll do_while_inv_unroll for_unroll for_inv_unroll while_to_while_no_cb while_invariant?! while_variant?! while_inv_rewrite_termination_result! while_inv_rewrite_data_ok! while_inv_rewrite_data_break! while_inv_rewrite_data_return! stmt_while_inv_rewrite_termination_result! stmt_while_inv_rewrite_data_ok!

stmt_while_inv_rewrite_data_break! stmt_while_inv_rewrite_data_return! pm_q_prop_while!
 pm_q_prop_stmt_while!

Installing automatic rewrites from: composition_assoc_rewrite_stmt_ok forward_s_c_s_of_s_expr
 comp_eval_if_ok_fstmt stmt_eval_if_ok_fstmt comp_eval_if_ok_fstmt_cstmt stmt_eval_if_ok_fstmt_cstmt
 composition_assoc_rewrite_1 composition_assoc_rewrite_2 composition_assoc_rewrite_3 com-
 position_assoc_rewrite_4 composition_assoc_rewrite_5 composition_assoc_rewrite_6 composi-
 tion_assoc_rewrite_7 composition_assoc_rewrite_8 composition_assoc_rewrite_expr_1 composi-
 tion_assoc_rewrite_expr_2 composition_assoc_rewrite_expr_3 composition_assoc_rewrite_expr_4 composi-
 tion_assoc_rewrite_expr_5 composition_assoc_rewrite_expr_6 composition_assoc_rewrite_stmt_ok compo-
 sition_assoc_rewrite_stmt_ok_expr composition_assoc_rewrite_stmt_ok_expr_lexpr comp_eval_if_ok_fstmt
 comp_eval_if_ok_fstmt_expr comp_eval_if_ok_fstmt_cstmt_expr stmt_eval_if_ok_fstmt_expr stmt_eval_if_ok_fstmt_cst-
 comp_eval_if_ok_fexpr_expr comp_eval_if_ok_fexpr stmt_eval_if_ok_fexpr expr_eval_if_ok_fexpr compo-
 sition_assoc_expression_rewrite composition_assoc_expression_rewrite_stmt stmt_eval_if_ok_fexpr_expr
 expr_eval_if_ok_fexpr_expr ok_result_fexpr ok_result_fstmt composition_assoc_rewrite_stmt_ok_cstmt

Installing automatic rewrites from: allocate_stack deallocate_stack with_new_stackvar

Installing automatic rewrites from: ok_result_ok ok_result_data ok_result_q_ok ok_result_q_state
 ok_result_q_data ok_result_elimination_1 ok_result_elimination_2 ok_result_state

Installing automatic rewrites from: l2r integral_to_bool pointer_to_bool bool2int

Installing automatic rewrites from: comp_simple_expr_simple_expr_ok comp_simple_expr_simple_expr_data
 comp_simple_expr_simple_expr_state comp_simple_stmt_simple_expr_ok comp_simple_stmt_simple_expr_data
 comp_simple_stmt_simple_expr_state

Installing automatic rewrites from: floating_point?

Installing automatic rewrites from: cv_base_result2 cv_base_result_c cv_base_result_v
 cv_base_result_cv cv c v

Installing automatic rewrites from: uchar_range_values

Repeatedly simplifying with decision procedures, rewriting, propositional reasoning, quantifier in-
 stantiation, skolemization, if-lifting and equality replacement,

This completes the proof of `search_found`.

Q.E.D.

C.199 Proofs for Segment_Reg_Datatype (paging-data.pvs)

C.199.1 Segment_Reg_Datatype.segment_selector_data_type_TCC1

Terse proof for `segment_selector_data_type_TCC1`.

`segment_selector_data_type_TCC1`:

{1} $\exists (x: (\text{pod_data_type?}[\text{Segment_Selector_type}])): \text{TRUE}$

Rewriting using `segment_selector_type_exists`, matching in *,
 This completes the proof of `segment_selector_data_type_TCC1`.

Q.E.D.

C.199.2 Segment_Reg_Datatype.segment_reg_data_type_TCC1

Terse proof for `segment_reg_data_type_TCC1`.

`segment_reg_data_type_TCC1`:

{1} $\exists (x: (\text{pod_data_type?}[\text{Segment_Reg_type}])): \text{TRUE}$

Using lemma `segment_reg_data_type_exists`,
This completes the proof of `segment_reg_data_type_TCC1`.
Q.E.D.

C.200 Proofs for Segment_Types (paging-data.pvs)

This theory contains no provable formal statements.

C.201 Proofs for SelectionStatements (statements.pvs)

This theory contains no provable formal statements.

C.202 Proofs for Set_Lift (vfiasco-prelude.pvs)

This theory contains no provable formal statements.

C.203 Proofs for Single_Statement_Rewrites0 (statement-rewrites.pvs)

C.203.1 Single_Statement_Rewrites0.break_catch_break

Terse proof for `break_catch_break`.

`break_catch_break`:

$$\frac{\{1\} \quad \forall (s: \text{State}, \text{stmt}: [\text{State} \rightarrow \text{StmtResult}[\text{State}]])}{\text{OK}?(s) \supset (\text{stmt} \## \text{break} \## \text{catch_break})(s) = \text{stmt}(s)}$$

Repeatedly Skolemizing and flattening,
Trying repeated skolemization, instantiation, and if-lifting,
Applying extensionality,
This completes the proof of `break_catch_break`.
Q.E.D.

C.203.2 Single_Statement_Rewrites0.break_break_stmt

Terse proof for `break_break_stmt`.

`break_break_stmt`:

$$\frac{\{1\} \quad \forall (s: \text{State}, \text{stmt}, \text{stmt1}: [\text{State} \rightarrow \text{StmtResult}[\text{State}]])}{\text{Break}?(s) \supset (\text{stmt} \## \text{stmt1} \## \text{catch_break})(s) = (\text{stmt} \## \text{catch_break})(s)}$$

Repeatedly Skolemizing and flattening,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `break_break_stmt`.
Q.E.D.

C.203.3 Single_Statement_Rewrites0.break_break_catch_continue

Terse proof for `break_break_catch_continue`.

`break_break_catch_continue`:

$$\{1\} \quad \forall (s: \text{State}, \text{stmt}: [\text{State} \rightarrow \text{StmtResult}[\text{State}]]):$$

$$\text{Break?}(\text{stmt}(s)) \supset$$

$$(\text{stmt} \## \text{catch_continue} \## \text{catch_break})(s) = (\text{stmt} \## \text{catch_break})(s)$$

,
This completes the proof of `break_break_catch_continue`.

Q.E.D.

C.203.4 Single_Statement_Rewrites0.break_break_catch_default

Terse proof for `break_break_catch_default`.

`break_break_catch_default`:

$$\{1\} \quad \forall (s: \text{State}, \text{stmt}: [\text{State} \rightarrow \text{StmtResult}[\text{State}]]):$$

$$\text{Break?}(\text{stmt}(s)) \supset$$

$$(\text{stmt} \## \text{catch_default} \## \text{catch_break})(s) = (\text{stmt} \## \text{catch_break})(s)$$

,
This completes the proof of `break_break_catch_default`.

Q.E.D.

C.203.5 Single_Statement_Rewrites0.break_break_lift_stmt

Terse proof for `break_break_lift_stmt`.

`break_break_lift_stmt`:

$$\{1\} \quad \forall (s: \text{State}, \text{stmt}, \text{stmt1}: [\text{State} \rightarrow \text{StmtResult}[\text{State}]]):$$

$$\text{Break?}(\text{stmt}(s)) \supset$$

$$(\text{stmt} \## \text{lift}(\text{stmt1}) \## \text{catch_break})(s) = (\text{stmt} \## \text{catch_break})(s)$$

,
This completes the proof of `break_break_lift_stmt`.

Q.E.D.

C.203.6 Single_Statement_Rewrites0.break_break_case

Terse proof for `break_break_case`.

`break_break_case`:

$$\{1\} \quad \forall (n: \text{int}, s: \text{State}, \text{stmt}: [\text{State} \rightarrow \text{StmtResult}[\text{State}]]):$$

$$\text{Break?}(\text{stmt}(s)) \supset$$

$$(\text{stmt} \## \text{case}(n) \## \text{catch_break})(s) = (\text{stmt} \## \text{catch_break})(s)$$

,
This completes the proof of `break_break_case`.

Q.E.D.

C.203.7 Single_Statement_Rewrites0.break_break_default

Terse proof for break_break_default.

break_break_default:

$$\frac{\{1\} \quad \forall (s: \text{State}, \text{stmt}: [\text{State} \rightarrow \text{StmtResult}[\text{State}]]) : \quad \text{Break?}(\text{stmt}(s)) \supset (\text{stmt} \## \text{default} \## \text{catch_break})(s) = (\text{stmt} \## \text{catch_break})(s)}{}$$

This completes the proof of break_break_default.

Q.E.D.

C.203.8 Single_Statement_Rewrites0.break_break

Terse proof for break_break.

break_break:

$$\frac{\{1\} \quad \forall (s: \text{State}): \text{Break?}(\text{break}(s))}{}$$

This completes the proof of break_break.

Q.E.D.

C.203.9 Single_Statement_Rewrites0.stmt_break_break

Terse proof for stmt_break_break.

stmt_break_break:

$$\frac{\{1\} \quad \forall (s: \text{State}, \text{stmt}: [\text{State} \rightarrow \text{StmtResult}[\text{State}]]) : \quad \text{OK?}(\text{stmt}(s)) \supset \text{Break?}((\text{stmt} \## \text{break})(s))}{}$$

This completes the proof of stmt_break_break.

Q.E.D.

C.203.10 Single_Statement_Rewrites0.break_stmt_break

Terse proof for break_stmt_break.

break_stmt_break:

$$\frac{\{1\} \quad \forall (s: \text{State}, \text{stmt}, \text{stmt1}: [\text{State} \rightarrow \text{StmtResult}[\text{State}]]) : \quad \text{Break?}(\text{stmt}(s)) \supset \text{Break?}((\text{stmt} \## \text{stmt1})(s))}{}$$

This completes the proof of break_stmt_break.

Q.E.D.

C.203.11 Single_Statement_Rewrites0.break_stmt_catch_continue

Terse proof for break_stmt_catch_continue.

break_stmt_catch_continue:

$$\frac{\{1\} \quad \forall (s: \text{State}, \text{stmt}: [\text{State} \rightarrow \text{StmtResult}[\text{State}]]) : \quad \text{Break?}(\text{stmt}(s)) \supset \text{Break?}((\text{stmt} \## \text{catch_continue})(s))}{}$$

,
This completes the proof of `break_stmt_catch_continue`.
Q.E.D.

C.203.12 Single_Statement_Rewrites0.break_stmt_catch_default

Terse proof for `break_stmt_catch_default`.

`break_stmt_catch_default`:

$$\frac{\{1\} \quad \forall (s: \text{State}, \text{stmt}: [\text{State} \rightarrow \text{StmtResult}[\text{State}]]) : \quad \text{Break?}(\text{stmt}(s)) \supset \text{Break?}((\text{stmt} \ \#\# \ \text{catch_default})(s))}{}$$

,
This completes the proof of `break_stmt_catch_default`.
Q.E.D.

C.203.13 Single_Statement_Rewrites0.break_stmt_lift

Terse proof for `break_stmt_lift`.

`break_stmt_lift`:

$$\frac{\{1\} \quad \forall (s: \text{State}, \text{stmt}, \text{stmt1}: [\text{State} \rightarrow \text{StmtResult}[\text{State}]]) : \quad \text{Break?}(\text{stmt}(s)) \supset \text{Break?}((\text{stmt} \ \#\# \ \text{lift}(\text{stmt1}))(s))}{}$$

,
This completes the proof of `break_stmt_lift`.
Q.E.D.

C.203.14 Single_Statement_Rewrites0.break_stmt_case

Terse proof for `break_stmt_case`.

`break_stmt_case`:

$$\frac{\{1\} \quad \forall (n: \text{int}, s: \text{State}, \text{stmt}: [\text{State} \rightarrow \text{StmtResult}[\text{State}]]) : \quad \text{Break?}(\text{stmt}(s)) \supset \text{Break?}((\text{stmt} \ \#\# \ \text{case}(n))(s))}{}$$

,
This completes the proof of `break_stmt_case`.
Q.E.D.

C.203.15 Single_Statement_Rewrites0.break_stmt_default

Terse proof for `break_stmt_default`.

`break_stmt_default`:

$$\frac{\{1\} \quad \forall (s: \text{State}, \text{stmt}: [\text{State} \rightarrow \text{StmtResult}[\text{State}]]) : \quad \text{Break?}(\text{stmt}(s)) \supset \text{Break?}((\text{stmt} \ \#\# \ \text{default})(s))}{}$$

,
This completes the proof of `break_stmt_default`.
Q.E.D.

C.203.16 Single_Statement_Rewrites0.continue_catch_continue

Terse proof for continue_catch_continue.

continue_catch_continue:

$$\{1\} \quad \forall (s: \text{State}, \text{stmt}: [\text{State} \rightarrow \text{StmtResult}[\text{State}]]):$$

$$\text{OK?}(\text{stmt}(s)) \supset (\text{stmt} \## \text{continue} \## \text{catch_continue})(s) = \text{stmt}(s)$$

Repeatedly Skolemizing and flattening,
 Trying repeated skolemization, instantiation, and if-lifting,
 Applying extensionality,
 This completes the proof of continue_catch_continue.
 Q.E.D.

C.203.17 Single_Statement_Rewrites0.continue_break_stmt

Terse proof for continue_break_stmt.

continue_break_stmt:

$$\{1\} \quad \forall (s: \text{State}, \text{stmt}, \text{stmt1}: [\text{State} \rightarrow \text{StmtResult}[\text{State}]]):$$

$$\text{Continue?}(\text{stmt}(s)) \supset$$

$$(\text{stmt} \## \text{stmt1} \## \text{catch_continue})(s) = (\text{stmt} \## \text{catch_continue})(s)$$

,
 This completes the proof of continue_break_stmt.
 Q.E.D.

C.203.18**Single_Statement_Rewrites0.continue_break_catch_continue**

Terse proof for continue_break_catch_continue.

continue_break_catch_continue:

$$\{1\} \quad \forall (s: \text{State}, \text{stmt}: [\text{State} \rightarrow \text{StmtResult}[\text{State}]]):$$

$$\text{Continue?}(\text{stmt}(s)) \supset$$

$$(\text{stmt} \## \text{catch_break} \## \text{catch_continue})(s) = (\text{stmt} \## \text{catch_continue})(s)$$

,
 This completes the proof of continue_break_catch_continue.
 Q.E.D.

C.203.19 Single_Statement_Rewrites0.continue_break_catch_default

Terse proof for continue_break_catch_default.

continue_break_catch_default:

$$\{1\} \quad \forall (s: \text{State}, \text{stmt}: [\text{State} \rightarrow \text{StmtResult}[\text{State}]]):$$

$$\text{Continue?}(\text{stmt}(s)) \supset$$

$$(\text{stmt} \## \text{catch_default} \## \text{catch_continue})(s) = (\text{stmt} \## \text{catch_continue})(s)$$

,
 This completes the proof of continue_break_catch_default.
 Q.E.D.

C.203.20 Single_Statement_Rewrites0.continue_break_lift_stmt

Terse proof for `continue_break_lift_stmt`.

`continue_break_lift_stmt`:

$$\{1\} \quad \forall (s: \text{State}, \text{stmt}, \text{stmt1}: [\text{State} \rightarrow \text{StmtResult}[\text{State}]]):$$
$$\text{Continue?}(\text{stmt}(s)) \supset$$
$$(\text{stmt} \## \text{lift}(\text{stmt1}) \## \text{catch_continue})(s) = (\text{stmt} \## \text{catch_continue})(s)$$

,
This completes the proof of `continue_break_lift_stmt`.

Q.E.D.

C.203.21 Single_Statement_Rewrites0.continue_break_case

Terse proof for `continue_break_case`.

`continue_break_case`:

$$\{1\} \quad \forall (n: \text{int}, s: \text{State}, \text{stmt}: [\text{State} \rightarrow \text{StmtResult}[\text{State}]]):$$
$$\text{Continue?}(\text{stmt}(s)) \supset$$
$$(\text{stmt} \## \text{case}(n) \## \text{catch_continue})(s) = (\text{stmt} \## \text{catch_continue})(s)$$

,
This completes the proof of `continue_break_case`.

Q.E.D.

C.203.22 Single_Statement_Rewrites0.continue_break_default

Terse proof for `continue_break_default`.

`continue_break_default`:

$$\{1\} \quad \forall (s: \text{State}, \text{stmt}: [\text{State} \rightarrow \text{StmtResult}[\text{State}]]):$$
$$\text{Continue?}(\text{stmt}(s)) \supset$$
$$(\text{stmt} \## \text{default} \## \text{catch_continue})(s) = (\text{stmt} \## \text{catch_continue})(s)$$

,
This completes the proof of `continue_break_default`.

Q.E.D.

C.203.23 Single_Statement_Rewrites0.continue_continue

Terse proof for `continue_continue`.

`continue_continue`:

$$\{1\} \quad \forall (s: \text{State}): \text{Continue?}(\text{continue}(s))$$

,
This completes the proof of `continue_continue`.

Q.E.D.

C.203.24 Single_Statement_Rewrites0.stmt_continue_continue

Terse proof for `stmt_continue_continue`.

stmt_continue_continue:

$$\frac{\{1\} \quad \forall (s: \text{State}, \text{stmt}: [\text{State} \rightarrow \text{StmtResult}[\text{State}]]) : \text{OK}?(\text{stmt}(s)) \supset \text{Continue}?((\text{stmt} \## \text{continue})(s))}{}$$

,
This completes the proof of stmt_continue_continue.
Q.E.D.

C.203.25 Single_Statement_Rewrites0.continue_stmt_continue

Terse proof for continue_stmt_continue.

continue_stmt_continue:

$$\frac{\{1\} \quad \forall (s: \text{State}, \text{stmt}, \text{stmt1}: [\text{State} \rightarrow \text{StmtResult}[\text{State}]]) : \text{Continue}?(\text{stmt}(s)) \supset \text{Continue}?((\text{stmt} \## \text{stmt1})(s))}{}$$

,
This completes the proof of continue_stmt_continue.
Q.E.D.

C.203.26 Single_Statement_Rewrites0.continue_stmt_catch_break

Terse proof for continue_stmt_catch_break.

continue_stmt_catch_break:

$$\frac{\{1\} \quad \forall (s: \text{State}, \text{stmt}: [\text{State} \rightarrow \text{StmtResult}[\text{State}]]) : \text{Continue}?(\text{stmt}(s)) \supset \text{Continue}?((\text{stmt} \## \text{catch_break})(s))}{}$$

,
This completes the proof of continue_stmt_catch_break.
Q.E.D.

C.203.27 Single_Statement_Rewrites0.continue_stmt_catch_default

Terse proof for continue_stmt_catch_default.

continue_stmt_catch_default:

$$\frac{\{1\} \quad \forall (s: \text{State}, \text{stmt}: [\text{State} \rightarrow \text{StmtResult}[\text{State}]]) : \text{Continue}?(\text{stmt}(s)) \supset \text{Continue}?((\text{stmt} \## \text{catch_default})(s))}{}$$

,
This completes the proof of continue_stmt_catch_default.
Q.E.D.

C.203.28 Single_Statement_Rewrites0.continue_stmt_lift

Terse proof for continue_stmt_lift.

continue_stmt_lift:

$$\frac{\{1\} \quad \forall (s: \text{State}, \text{stmt}, \text{stmt1}: [\text{State} \rightarrow \text{StmtResult}[\text{State}]]) : \text{Continue}?(\text{stmt}(s)) \supset \text{Continue}?((\text{stmt} \## \text{lift}(\text{stmt1}))(s))}{}$$

,

This completes the proof of `continue_stmt_lift`.
Q.E.D.

C.203.29 Single_Statement_Rewrites0.continue_stmt_case

Terse proof for `continue_stmt_case`.

`continue_stmt_case`:

$$\frac{\{1\} \quad \forall (n: \text{int}, s: \text{State}, \text{stmt}: [\text{State} \rightarrow \text{StmtResult}[\text{State}]])}{\text{Continue?}(\text{stmt}(s)) \supset \text{Continue?}((\text{stmt} \ \#\# \ \text{case}(n))(s))}$$

This completes the proof of `continue_stmt_case`.
Q.E.D.

C.203.30 Single_Statement_Rewrites0.continue_stmt_default

Terse proof for `continue_stmt_default`.

`continue_stmt_default`:

$$\frac{\{1\} \quad \forall (s: \text{State}, \text{stmt}: [\text{State} \rightarrow \text{StmtResult}[\text{State}]])}{\text{Continue?}(\text{stmt}(s)) \supset \text{Continue?}((\text{stmt} \ \#\# \ \text{default})(s))}$$

This completes the proof of `continue_stmt_default`.
Q.E.D.

C.203.31 Single_Statement_Rewrites0.switch_stmt

Terse proof for `switch_stmt`.

`switch_stmt`:

$$\frac{\{1\} \quad \forall (n: \text{int}, \text{stmt}: [\text{State} \rightarrow \text{StmtResult}[\text{State}]])}{\text{switch_result}(n) \ \#\# \ \text{stmt} = \text{switch_result}(n)}$$

Repeatedly Skolemizing and flattening,
Trying repeated skolemization, instantiation, and if-lifting,
Applying extensionality,
This completes the proof of `switch_stmt`.
Q.E.D.

C.203.32 Single_Statement_Rewrites0.stmt_switch_stmt

Terse proof for `stmt_switch_stmt`.

`stmt_switch_stmt`:

$$\frac{\{1\} \quad \forall (n: \text{int}, \text{stmt}, \text{stmt1}: [\text{State} \rightarrow \text{StmtResult}[\text{State}]])}{\text{stmt} \ \#\# \ \text{switch_result}(n) \ \#\# \ \text{stmt1} = \text{stmt} \ \#\# \ \text{switch_result}(n)}$$

Repeatedly Skolemizing and flattening,
Trying repeated skolemization, instantiation, and if-lifting,
Applying extensionality,
Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `stmt_switch_stmt`.
Q.E.D.

C.203.33 Single_Statement_Rewrites0.switch_case_taken

Terse proof for `switch_case_taken`.

`switch_case_taken`:

$$\{1\} \quad \forall (n: \text{int}): (\text{switch_result}[\text{State}](n) \#\# \text{case}(n)) = \text{skip}$$

Repeatedly Skolemizing and flattening,
Trying repeated skolemization, instantiation, and if-lifting,
Applying extensionality,
This completes the proof of `switch_case_taken`.
Q.E.D.

C.203.34 Single_Statement_Rewrites0.stmt_switch_case_taken

Terse proof for `stmt_switch_case_taken`.

`stmt_switch_case_taken`:

$$\{1\} \quad \forall (n: \text{int}, s: \text{State}, \text{stmt}: [\text{State} \rightarrow \text{StmtResult}[\text{State}]]): \\ \text{OK}?(\text{stmt}(s)) \supset (\text{stmt} \#\# \text{switch_result}(n) \#\# \text{case}(n))(s) = \text{stmt}(s)$$

Repeatedly Skolemizing and flattening,
Trying repeated skolemization, instantiation, and if-lifting,
Applying extensionality,
This completes the proof of `stmt_switch_case_taken`.
Q.E.D.

C.203.35 Single_Statement_Rewrites0.switch_case_not_taken

Terse proof for `switch_case_not_taken`.

`switch_case_not_taken`:

$$\{1\} \quad \forall (n_1, n_2: \text{int}): n_1 \neq n_2 \supset \text{switch_result}[\text{State}](n_1) \#\# \text{case}(n_2) = \text{switch_result}(n_1)$$

Expanding the definition of `##`,
Expanding the definition of `case`,
Expanding the definition of `switch_result`,
Repeatedly Skolemizing and flattening,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `switch_case_not_taken`.
Q.E.D.

C.203.36 Single_Statement_Rewrites0.stmt_switch_case_not_taken

Terse proof for `stmt_switch_case_not_taken`.

C Proof scripts

stmt_switch_case_not_taken:

$$\{1\} \quad \forall (s: \text{State}, \text{stmt}: [\text{State} \rightarrow \text{StmtResult}[\text{State}]], n_1, n_2: \text{int}):$$
$$\text{OK?}(\text{stmt}(s)) \wedge n_1 \neq n_2 \supset$$
$$(\text{stmt} \#\# \text{switch_result}(n_1) \#\# \text{case}(n_2))(s) = (\text{stmt} \#\# \text{switch_result}(n_1))(s)$$

,
This completes the proof of `stmt_switch_case_not_taken`.

Q.E.D.

C.203.37 Single_Statement_Rewrites0.switch_default

Terse proof for `switch_default`.

switch_default:

$$\{1\} \quad \forall (n: \text{int}): \text{switch_result}[\text{State}](n) \#\# \text{default} = \text{switch_result}(n)$$

Repeatedly Skolemizing and flattening,

Applying decompose-equality,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `switch_default`.

Q.E.D.

C.203.38 Single_Statement_Rewrites0.stmt_switch_default

Terse proof for `stmt_switch_default`.

stmt_switch_default:

$$\{1\} \quad \forall (n: \text{int}, s: \text{State}, \text{stmt}: [\text{State} \rightarrow \text{StmtResult}[\text{State}]]):$$
$$\text{OK?}(\text{stmt}(s)) \supset$$
$$(\text{stmt} \#\# \text{switch_result}(n) \#\# \text{default})(s) = (\text{stmt} \#\# \text{switch_result}(n))(s)$$

,
This completes the proof of `stmt_switch_default`.

Q.E.D.

C.203.39 Single_Statement_Rewrites0.switch_catch_break

Terse proof for `switch_catch_break`.

switch_catch_break:

$$\{1\} \quad \forall (n: \text{int}): \text{switch_result}[\text{State}](n) \#\# \text{catch_break} = \text{switch_result}(n)$$

Repeatedly Skolemizing and flattening,

Applying decompose-equality,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `switch_catch_break`.

Q.E.D.

C.203.40 Single_Statement_Rewrites0.stmt_switch_catch_break

Terse proof for stmt_switch_catch_break.

stmt_switch_catch_break:

$$\begin{array}{|l} \hline \{1\} \quad \forall (n: \text{int}, s: \text{State}, \text{stmt}: [\text{State} \rightarrow \text{StmtResult}[\text{State}]]): \\ \quad \text{OK?}(\text{stmt}(s)) \supset \\ \quad (\text{stmt} \#\#\ \text{switch_result}(n) \#\#\ \text{catch_break})(s) = (\text{stmt} \#\#\ \text{switch_result}(n))(s) \\ \hline \end{array}$$

,
This completes the proof of stmt_switch_catch_break.

Q.E.D.

C.203.41 Single_Statement_Rewrites0.switch_catch_continue

Terse proof for switch_catch_continue.

switch_catch_continue:

$$\begin{array}{|l} \hline \{1\} \quad \forall (n: \text{int}): \text{switch_result}[\text{State}](n) \#\#\ \text{catch_continue} = \text{switch_result}(n) \\ \hline \end{array}$$

Repeatedly Skolemizing and flattening,

Applying decompose-equality,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of switch_catch_continue.

Q.E.D.

C.203.42 Single_Statement_Rewrites0.stmt_switch_catch_continue

Terse proof for stmt_switch_catch_continue.

stmt_switch_catch_continue:

$$\begin{array}{|l} \hline \{1\} \quad \forall (n: \text{int}, s: \text{State}, \text{stmt}: [\text{State} \rightarrow \text{StmtResult}[\text{State}]]): \\ \quad \text{OK?}(\text{stmt}(s)) \supset \\ \quad (\text{stmt} \#\#\ \text{switch_result}(n) \#\#\ \text{catch_continue})(s) = (\text{stmt} \#\#\ \text{switch_result}(n))(s) \\ \hline \end{array}$$

,
This completes the proof of stmt_switch_catch_continue.

Q.E.D.

C.203.43 Single_Statement_Rewrites0.switch_catch_default

Terse proof for switch_catch_default.

switch_catch_default:

$$\begin{array}{|l} \hline \{1\} \quad \forall (n: \text{int}): \text{switch_result}[\text{State}](n) \#\#\ \text{catch_default} = \text{switch_result}(n) \\ \hline \end{array}$$

Repeatedly Skolemizing and flattening,

Applying decompose-equality,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of switch_catch_default.

Q.E.D.

C.203.44 Single_Statement_Rewrites0.stmt_switch_catch_default

Terse proof for `stmt_switch_catch_default`.

`stmt_switch_catch_default`:

$$\{1\} \quad \forall (n: \text{int}, s: \text{State}, \text{stmt}: [\text{State} \rightarrow \text{StmtResult}[\text{State}]]):$$

$$\text{OK?}(\text{stmt}(s)) \supset$$

$$(\text{stmt} \ \#\# \ \text{switch_result}(n) \ \#\# \ \text{catch_default})(s) = (\text{stmt} \ \#\# \ \text{switch_result}(n))(s)$$

This completes the proof of `stmt_switch_catch_default`.

Q.E.D.

C.203.45 Single_Statement_Rewrites0.default_stmt

Terse proof for `default_stmt`.

`default_stmt`:

$$\{1\} \quad \forall (\text{stmt}: [\text{State} \rightarrow \text{StmtResult}[\text{State}]]):$$

$$\text{default_result} \ \#\# \ \text{stmt} = \text{default_result}$$

Repeatedly Skolemizing and flattening,

Applying decompose-equality,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `default_stmt`.

Q.E.D.

C.203.46 Single_Statement_Rewrites0.stmt_default_stmt

Terse proof for `stmt_default_stmt`.

`stmt_default_stmt`:

$$\{1\} \quad \forall (\text{stmt}, \text{stmt1}: [\text{State} \rightarrow \text{StmtResult}[\text{State}]]):$$

$$\text{stmt} \ \#\# \ \text{default_result} \ \#\# \ \text{stmt1} = \text{stmt} \ \#\# \ \text{default_result}$$

Repeatedly Skolemizing and flattening,

Applying decompose-equality,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `stmt_default_stmt`.

Q.E.D.

C.203.47 Single_Statement_Rewrites0.default_case

Terse proof for `default_case`.

`default_case`:

$$\{1\} \quad \forall (n: \text{int}): \text{default_result}[\text{State}] \ \#\# \ \text{case}(n) = \text{default_result}$$

Repeatedly Skolemizing and flattening,

Applying decompose-equality,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `default_case`.

Q.E.D.

C.203.48 Single_Statement_Rewrites0.stmt_default_case

Terse proof for `stmt_default_case`.

`stmt_default_case`:

$$\frac{\{1\} \quad \forall (n: \text{int}, s: \text{State}, \text{stmt}: [\text{State} \rightarrow \text{StmtResult}[\text{State}]]) : \text{OK?}(\text{stmt}(s)) \supset (\text{stmt} \#\# \text{default_result} \#\# \text{case}(n))(s) = (\text{stmt} \#\# \text{default_result})(s)}{}$$

This completes the proof of `stmt_default_case`.

Q.E.D.

C.203.49 Single_Statement_Rewrites0.default_default

Terse proof for `default_default`.

`default_default`:

$$\frac{\{1\} \quad \text{default_result}[\text{State}] \#\# \text{default} = \text{skip}}{}$$

Applying `decompose-equality`,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `default_default`.

Q.E.D.

C.203.50 Single_Statement_Rewrites0.stmt_default_default

Terse proof for `stmt_default_default`.

`stmt_default_default`:

$$\frac{\{1\} \quad \forall (s: \text{State}, \text{stmt}: [\text{State} \rightarrow \text{StmtResult}[\text{State}]]) : \text{OK?}(\text{stmt}(s)) \supset (\text{stmt} \#\# \text{default_result} \#\# \text{default})(s) = \text{stmt}(s)}{}$$

Repeatedly Skolemizing and flattening,
Trying repeated skolemization, instantiation, and if-lifting,
Applying extensionality,
This completes the proof of `stmt_default_default`.

Q.E.D.

C.203.51 Single_Statement_Rewrites0.default_catch_break

Terse proof for `default_catch_break`.

`default_catch_break`:

$$\frac{\{1\} \quad \text{default_result}[\text{State}] \#\# \text{catch_break} = \text{default_result}}{}$$

Applying `decompose-equality`,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `default_catch_break`.

Q.E.D.

C.203.52 Single_Statement_Rewrites0.stmt_default_catch_break

Terse proof for `stmt_default_catch_break`.

`stmt_default_catch_break`:

$$\{1\} \quad \forall (s: \text{State}, \text{stmt}: [\text{State} \rightarrow \text{StmtResult}[\text{State}]]):$$
$$\text{OK?}(\text{stmt}(s)) \supset$$
$$(\text{stmt} \#\# \text{default_result} \#\# \text{catch_break})(s) = (\text{stmt} \#\# \text{default_result})(s)$$

,
This completes the proof of `stmt_default_catch_break`.
Q.E.D.

C.203.53 Single_Statement_Rewrites0.default_catch_continue

Terse proof for `default_catch_continue`.

`default_catch_continue`:

$$\{1\} \quad \text{default_result}[\text{State}] \#\# \text{catch_continue} = \text{default_result}$$

Applying `decompose-equality`,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `default_catch_continue`.
Q.E.D.

C.203.54 Single_Statement_Rewrites0.stmt_default_catch_continue

Terse proof for `stmt_default_catch_continue`.

`stmt_default_catch_continue`:

$$\{1\} \quad \forall (s: \text{State}, \text{stmt}: [\text{State} \rightarrow \text{StmtResult}[\text{State}]]):$$
$$\text{OK?}(\text{stmt}(s)) \supset$$
$$(\text{stmt} \#\# \text{default_result} \#\# \text{catch_continue})(s) = (\text{stmt} \#\# \text{default_result})(s)$$

,
This completes the proof of `stmt_default_catch_continue`.
Q.E.D.

C.203.55 Single_Statement_Rewrites0.default_catch_default

Terse proof for `default_catch_default`.

`default_catch_default`:

$$\{1\} \quad \text{default_result}[\text{State}] \#\# \text{catch_default} = \text{skip}$$

Applying `decompose-equality`,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `default_catch_default`.
Q.E.D.

C.203.56 Single_Statement_Rewrites0.stmt_default_catch_default

Terse proof for stmt_default_catch_default.

stmt_default_catch_default:

$$\frac{}{\{1\} \quad \forall (s: \text{State}, \text{stmt}: [\text{State} \rightarrow \text{StmtResult}[\text{State}]]) : \text{OK}?(\text{stmt}(s)) \supset (\text{stmt} \text{ ## } \text{default_result} \text{ ## } \text{catch_default})(s) = \text{stmt}(s)}$$

Repeatedly Skolemizing and flattening,
 Trying repeated skolemization, instantiation, and if-lifting,
 Applying extensionality,
 This completes the proof of stmt_default_catch_default.
 Q.E.D.

C.203.57 Single_Statement_Rewrites0.return_void_result

Terse proof for return_void_result.

return_void_result:

$$\frac{}{\{1\} \quad \text{return_void}[\text{State}] = \text{return_result}}$$

This completes the proof of return_void_result.
 Q.E.D.

C.203.58 Single_Statement_Rewrites0.skip_ok

Terse proof for skip_ok.

skip_ok:

$$\frac{}{\{1\} \quad \forall (s: \text{State}): \text{OK}?(\text{skip}(s))}$$

This completes the proof of skip_ok.
 Q.E.D.

C.203.59 Single_Statement_Rewrites0.skip_state_TCC1

Terse proof for skip_state_TCC1.

skip_state_TCC1:

$$\frac{}{\{1\} \quad \forall (s: \text{State}): \text{OK}?[\text{State}](\text{skip}[\text{State}](s)) \vee \text{Break}?[\text{State}](\text{skip}[\text{State}](s)) \vee \text{Continue}?[\text{State}](\text{skip}[\text{State}](s)) \vee \text{Return}?[\text{State}](\text{skip}[\text{State}](s)) \vee \text{Switch}?[\text{State}](\text{skip}[\text{State}](s)) \vee \text{Default}?[\text{State}](\text{skip}[\text{State}](s)) \vee \text{Exception}?[\text{State}](\text{skip}[\text{State}](s))}$$

Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of skip_state_TCC1.
 Q.E.D.

C.203.60 Single_Statement_Rewrites0.skip_state

Terse proof for skip_state.

skip_state:

$$\frac{}{\{1\} \quad \forall (s: \text{State}): \text{state}(\text{skip}(s)) = s}$$

This completes the proof of skip_state.

Q.E.D.

C.203.61 Single_Statement_Rewrites0.skip_elimination

Terse proof for skip_elimination.

skip_elimination:

$$\frac{}{\{1\} \quad \forall (\text{stmt}: [\text{State} \rightarrow \text{StmtResult}[\text{State}]]): (\text{stmt} \ \#\# \ \text{skip}) = \text{stmt}}$$

Repeatedly Skolemizing and flattening,

Applying decompose-equality,

Trying repeated skolemization, instantiation, and if-lifting,

Applying extensionality,

This completes the proof of skip_elimination.

Q.E.D.

C.203.62 Single_Statement_Rewrites0.stmt_skip_elimination

Terse proof for stmt_skip_elimination.

stmt_skip_elimination:

$$\frac{}{\{1\} \quad \forall (\text{stmt}: [\text{State} \rightarrow \text{StmtResult}[\text{State}]]): (\text{skip} \ \#\# \ \text{stmt}) = \text{stmt}}$$

Repeatedly Skolemizing and flattening,

Applying decompose-equality,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of stmt_skip_elimination.

Q.E.D.

C.203.63 Single_Statement_Rewrites0.stmt_remove_catch_default

Terse proof for stmt_remove_catch_default.

stmt_remove_catch_default:

$$\frac{}{\{1\} \quad \forall (s: \text{State}, \text{stmt}: [\text{State} \rightarrow \text{StmtResult}[\text{State}]]): \text{OK}?(s) \supset (\text{stmt} \ \#\# \ \text{catch_default})(s) = \text{stmt}(s)}$$

This completes the proof of stmt_remove_catch_default.

Q.E.D.

C.203.64 Single_Statement_Rewrites0.stmt_remove_case

Terse proof for stmt_remove_case.

stmt_remove_case:

$$\frac{\{1\} \quad \forall (n: \text{int}, s: \text{State}, \text{stmt1}: [\text{State} \rightarrow \text{StmtResult}[\text{State}]]) : \quad \text{OK?}(\text{stmt1}(s)) \supset (\text{stmt1} \#\# \text{case}(n))(s) = \text{stmt1}(s)}{}$$

This completes the proof of stmt_remove_case.

Q.E.D.

C.203.65 Single_Statement_Rewrites0.stmt_remove_default

Terse proof for stmt_remove_default.

stmt_remove_default:

$$\frac{\{1\} \quad \forall (s: \text{State}, \text{stmt1}: [\text{State} \rightarrow \text{StmtResult}[\text{State}]]) : \quad \text{OK?}(\text{stmt1}(s)) \supset (\text{stmt1} \#\# \text{default})(s) = \text{stmt1}(s)}{}$$

This completes the proof of stmt_remove_default.

Q.E.D.

C.203.66 Single_Statement_Rewrites0.stmt_remove_catch_continue

Terse proof for stmt_remove_catch_continue.

stmt_remove_catch_continue:

$$\frac{\{1\} \quad \forall (s: \text{State}, \text{stmt}: [\text{State} \rightarrow \text{StmtResult}[\text{State}]]) : \quad \text{OK?}(\text{stmt}(s)) \supset (\text{stmt} \#\# \text{catch_continue})(s) = \text{stmt}(s)}{}$$

This completes the proof of stmt_remove_catch_continue.

Q.E.D.

C.203.67 Single_Statement_Rewrites0.stmt_remove_catch_break

Terse proof for stmt_remove_catch_break.

stmt_remove_catch_break:

$$\frac{\{1\} \quad \forall (s: \text{State}, \text{stmt}: [\text{State} \rightarrow \text{StmtResult}[\text{State}]]) : \quad \text{OK?}(\text{stmt}(s)) \supset (\text{stmt} \#\# \text{catch_break})(s) = \text{stmt}(s)}{}$$

This completes the proof of stmt_remove_catch_break.

Q.E.D.

C.203.68 Single_Statement_Rewrites0.stmt_ok_lift

Terse proof for stmt_ok_lift.

C Proof scripts

stmt_ok_lift:

$$\frac{\{1\} \quad \forall (s: \text{State}, \text{stmt1}, \text{stmt2}: [\text{State} \rightarrow \text{StmtResult}[\text{State}]]) : \text{OK}?(\text{stmt1}(s)) \supset (\text{stmt1} \text{ ## lift}(\text{stmt2}))(s) = (\text{stmt1} \text{ ## stmt2})(s)}$$

This completes the proof of `stmt_ok_lift`.

Q.E.D.

C.203.69 Single_Statement_Rewrites0.while_unroll_TCC1

Terse proof for `while_unroll_TCC1`.

while_unroll_TCC1:

$$\frac{\{1\} \quad \forall (\text{max}: \text{nat}): \text{max} > 0 \supset \text{max} - 1 \geq 0}$$

Repeatedly Skolemizing and flattening,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `while_unroll_TCC1`.

Q.E.D.

C.203.70 Single_Statement_Rewrites0.while_unroll

Terse proof for `while_unroll`.

while_unroll:

$$\frac{\{1\} \quad \forall (\text{b_ex}: [\text{State} \rightarrow \text{ExprResult}[\text{State}, \text{bool}]], \text{max}: \text{nat}, s: \text{State}, \text{stmt}: [\text{State} \rightarrow \text{StmtResult}[\text{State}]]) : \text{max} > 0 \supset \text{while}(\text{max}, \text{b_ex}, \text{stmt})(s) = \text{if_else}(\text{b_ex}, \text{stmt} \text{ ## catch_continue} \text{ ## while}(\text{max} - 1, \text{b_ex}, \text{stmt}) \text{ ## catch_break}, \text{skip})(s)}$$

Repeatedly Skolemizing and flattening,

Expanding the definition of `while`,

Using lemma `while_as_if_while`,

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `while_unroll`.

Q.E.D.

C.203.71 Single_Statement_Rewrites0.stmt_while_unroll

Terse proof for `stmt_while_unroll`.

stmt_while_unroll:

$\{1\} \quad \forall (b_ex: [State \rightarrow ExprResult[State, bool]], \max: nat, s: State, \\ \text{stmt1}, \text{stmt2}: [State \rightarrow StmtResult[State]]): \\ \max > 0 \supset \\ (\text{stmt1} \#\# \text{while}(\max, b_ex, \text{stmt2}))(s) = \\ (\text{stmt1} \#\# \\ \text{if_else}(b_ex, \\ \text{stmt2} \#\# \text{catch_continue} \#\# \text{while}(\max - 1, b_ex, \text{stmt2}) \#\# \text{catch_break}, \\ \text{skip})) \\ (s)$

Repeatedly Skolemizing and flattening,

Expanding the definition of $\#\#$,

Expanding the definition of lift,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Rewriting using while_unroll, matching in $*$,

This completes the proof of `stmt_while_unroll`.

Q.E.D.

C.203.72 Single_Statement_Rewrites0.iterate_while_ok

Terse proof for `iterate_while_ok`.

iterate_while_ok:

$\{1\} \quad \forall (b_ex: [State \rightarrow ExprResult[State, bool]], s_1: State, \\ \text{stmt}: [State \rightarrow StmtResult[State]], n: nat): \\ (\forall (s: State): OK?((e2s(b_ex) \#\# \text{stmt})(s))) \supset \\ OK?(iterate_while(n, b_ex, \text{stmt})(s_1))$
--

Inducting on n on formula 1,

we get 2 subgoals:

iterate_while_ok.1:

$\{1\} \quad \forall (b_ex: [State \rightarrow ExprResult[State, bool]], s_1: State, \\ \text{stmt}: [State \rightarrow StmtResult[State]]): \\ (\forall (s: State): OK?((e2s(b_ex) \#\# \text{stmt})(s))) \supset \\ OK?(iterate_while(0, b_ex, \text{stmt})(s_1))$
--

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `iterate_while_ok.1`.

iterate_while_ok.2:

<pre> {1} ∀ j: (∀ (b_ex: [State → ExprResult[State, bool]], s1: State, stmt: [State → StmtResult[State]]): (∀ (s: State): OK?((e2s(b_ex) ## stmt)(s)) ⊃ OK?(iterate_while(j, b_ex, stmt)(s1))) ⊃ (∀ (b_ex: [State → ExprResult[State, bool]], s1: State, stmt: [State → StmtResult[State]]): (∀ (s: State): OK?((e2s(b_ex) ## stmt)(s)) ⊃ OK?(iterate_while(j + 1, b_ex, stmt)(s1))) </pre>
--

Repeatedly Skolemizing and flattening,
Expanding the definition of iterate_while,
Instantiating quantified variables,
Expanding the definition of ##,
Expanding the definition of lift,
Case splitting on OK?(catch_continue((e2s(b_ex!1) ## stmt!1)(s1!1))),
we get 2 subgoals:

iterate_while_ok.2.1:

<pre> {-1} OK?(catch_continue((e2s(b_ex') ## stmt')(s1'))) {-2} j' ≥ 0 {-3} ∀ (s1: State): (∀ (s: State): OK?((e2s(b_ex') ## stmt')(s)) ⊃ OK?(iterate_while(j', b_ex', stmt')(s1))) {-4} ∀ (s: State): OK?((e2s(b_ex') ## stmt')(s)) </pre>
<pre> {1} OK?(CASES catch_continue((e2s(b_ex') ## stmt')(s1')) OF OK(state): iterate_while(j', b_ex', stmt')(state) ELSE catch_continue((e2s(b_ex') ## stmt')(s1')) ENDCASES) </pre>

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Instantiating the top quantifier in -3 with the terms: (state(catch_continue((e2s(b_ex!1) ## stmt!1)(s1!1))))),

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of iterate_while_ok.2.1.

iterate_while_ok.2.2:

<pre> {-1} j' ≥ 0 {-2} ∀ (s1: State): (∀ (s: State): OK?((e2s(b_ex') ## stmt')(s)) ⊃ OK?(iterate_while(j', b_ex', stmt')(s1))) {-3} ∀ (s: State): OK?((e2s(b_ex') ## stmt')(s)) </pre>
<pre> {1} OK?(catch_continue((e2s(b_ex') ## stmt')(s1'))) {2} OK?(CASES catch_continue((e2s(b_ex') ## stmt')(s1')) OF OK(state): iterate_while(j', b_ex', stmt')(state) ELSE catch_continue((e2s(b_ex') ## stmt')(s1')) ENDCASES) </pre>

Hiding formulas: 2,
Expanding the definition of catch_continue,
Instantiating quantified variables,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `iterate_while_ok.2.2`.

Q.E.D.

C.203.73 Single_Statement_Rewrites0.while_hang

Terse proof for `while_hang`.

`while_hang`:

$$\frac{\{1\} \quad \forall (b_ex: [State \rightarrow ExprResult[State, bool]], stmt: [State \rightarrow StmtResult[State]]): \quad (\forall (s: State): \quad OK?(b_ex(s) \wedge data(b_ex(s)) \wedge OK?((e2s(b_ex) \#\# stmt)(s))) \quad \supset \quad while(b_ex, stmt) = hang_result}{}$$

Repeatedly Skolemizing and flattening,

Applying decompose-equality,

Expanding the definition of `while`,

Expanding the definition of `hang_result`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Repeatedly Skolemizing and flattening,

Expanding the definition of `while_termination_point?`,

Hiding formulas: 1,

Case splitting on `OK?(iterate_while(n!1, b_ex!1, stmt!1)(x!1))`,

we get 2 subgoals:

`while_hang.1`:

$$\frac{\begin{array}{l} \{-1\} \quad OK?(iterate_while(n', b_ex', stmt')(x')) \\ \{-2\} \quad n' \geq 0 \\ \{-3\} \quad \neg OK?(iterate_while(n', b_ex', stmt')(x')) \vee \\ \quad \neg OK?(b_ex'(state(iterate_while(n', b_ex', stmt')(x')))) \vee \\ \quad \neg data(b_ex'(state(iterate_while(n', b_ex', stmt')(x')))) \\ \{-4\} \quad \forall (s: State): \\ \quad OK?(b_ex'(s) \wedge data(b_ex'(s)) \wedge OK?((e2s(b_ex') \#\# stmt')(s))) \end{array}}{}$$

Instantiating the top quantifier in -4 with the terms: `(state(iterate_while(n!1, b_ex!1, stmt!1)(x!1)))`,

we get 2 subgoals:

`while_hang.1.1`:

$$\frac{\begin{array}{l} \{-1\} \quad OK?(iterate_while(n', b_ex', stmt')(x')) \\ \{-2\} \quad n' \geq 0 \\ \{-3\} \quad \neg OK?(iterate_while(n', b_ex', stmt')(x')) \vee \\ \quad \neg OK?(b_ex'(state(iterate_while(n', b_ex', stmt')(x')))) \vee \\ \quad \neg data(b_ex'(state(iterate_while(n', b_ex', stmt')(x')))) \\ \{-4\} \quad OK?(b_ex'(state(iterate_while(n', b_ex', stmt')(x')))) \wedge \\ \quad data(b_ex'(state(iterate_while(n', b_ex', stmt')(x')))) \wedge \\ \quad OK?((e2s(b_ex') \#\# stmt')(state(iterate_while(n', b_ex', stmt')(x')))) \end{array}}{}$$

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `while_hang.1.1`.

`while_hang.1.2:`

<pre>{-1} OK?(iterate_while(n', b_ex', stmt')(x')) {-2} n' ≥ 0 {-3} ¬ OK?(iterate_while(n', b_ex', stmt')(x')) ∨ ¬ OK?(b_ex'(state(iterate_while(n', b_ex', stmt')(x')))) ∨ ¬ data(b_ex'(state(iterate_while(n', b_ex', stmt')(x'))))</pre>	<hr style="border: 0.5px solid black;"/> <pre>{1} OK?[State](iterate_while[State](n', b_ex', stmt')(x')) ∨ Break?[State](iterate_while[State](n', b_ex', stmt')(x')) ∨ Continue?[State](iterate_while[State](n', b_ex', stmt')(x')) ∨ Return?[State](iterate_while[State](n', b_ex', stmt')(x')) ∨ Switch?[State](iterate_while[State](n', b_ex', stmt')(x')) ∨ Default?[State](iterate_while[State](n', b_ex', stmt')(x')) ∨ Exception?[State](iterate_while[State](n', b_ex', stmt')(x'))</pre>
---	---

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `while_hang.1.2`.

`while_hang.2:`

<pre>{-1} n' ≥ 0 {-2} ¬ OK?(iterate_while(n', b_ex', stmt')(x')) ∨ ¬ OK?(b_ex'(state(iterate_while(n', b_ex', stmt')(x')))) ∨ ¬ data(b_ex'(state(iterate_while(n', b_ex', stmt')(x')))) {-3} ∃ (s: State): OK?(b_ex'(s)) ∧ data(b_ex'(s)) ∧ OK?((e2s(b_ex') ## stmt')(s))</pre>	<hr style="border: 0.5px solid black;"/> <pre>{1} OK?(iterate_while(n', b_ex', stmt')(x'))</pre>
---	--

Rewriting using `iterate_while_ok`, matching in *,
Keeping (-3 1) and hiding *,
Repeatedly Skolemizing and flattening,
Instantiating quantified variables,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `while_hang.2`.
Q.E.D.

C.203.74

Single_Statement_Rewrites0.composition_assoc_rewrite_stmt_ok

Terse proof for `composition_assoc_rewrite_stmt_ok`.

`composition_assoc_rewrite_stmt_ok:`

<pre>{1} ∃ (cstmt: [StmtResult[State] → StmtResult[State]], s: State, stmt, stmt1: [State → StmtResult[State]]): OK?(stmt(s)) ⊃ (stmt ## (stmt1 ## cstmt))(s) = ((stmt ## stmt1) ## cstmt)(s)</pre>	<hr style="border: 0.5px solid black;"/>
---	--

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `composition_assoc_rewrite_stmt_ok`.
Q.E.D.

C.203.75 Single_Statement_Rewrites0.composition_assoc_rewrite_stmt_ok_cstmt

Terse proof for composition_assoc_rewrite_stmt_ok_cstmt.

composition_assoc_rewrite_stmt_ok_cstmt:

$$\{1\} \quad \forall (cstmt, cstmt1: [\text{StmtResult}[\text{State}] \rightarrow \text{StmtResult}[\text{State}]], s: \text{State}, \\ \text{stmt}, \text{stmt1}: [\text{State} \rightarrow \text{StmtResult}[\text{State}]]): \\ \text{OK?}(\text{stmt}(s)) \supset \\ ((\text{stmt} \## (\text{stmt1} \## \text{cstmt})) \## \text{cstmt1})(s) = \\ (((\text{stmt} \## \text{stmt1}) \## \text{cstmt}) \## \text{cstmt1})(s)$$

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of composition_assoc_rewrite_stmt_ok_cstmt.
Q.E.D.

C.204 Proofs for Single_Statement_Rewrites1 (statement-rewrites.pvs)

C.204.1 Single_Statement_Rewrites1.forward_s_c_s_of_s_expr

Terse proof for forward_s_c_s_of_s_expr.

forward_s_c_s_of_s_expr:

$$\{1\} \quad \forall (cstmt: [\text{StmtResult}[\text{State}] \rightarrow \text{StmtResult}[\text{State}]], \\ \text{expr}: [\text{State} \rightarrow \text{ExprResult}[\text{State}, \text{Data}]], s: \text{State}, \\ \text{stmt}: [\text{State} \rightarrow \text{StmtResult}[\text{State}]]): \\ (\text{stmt} \## \text{cstmt})(s) = \text{stmt}(s) \supset (\text{stmt} \## \text{cstmt} \## \text{expr})(s) = (\text{stmt} \## \text{expr})(s)$$

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of forward_s_c_s_of_s_expr.
Q.E.D.

C.204.2 Single_Statement_Rewrites1.stmt_ok_lift_expr

Terse proof for stmt_ok_lift_expr.

stmt_ok_lift_expr:

$$\{1\} \quad \forall (\text{expr}: [\text{State} \rightarrow \text{ExprResult}[\text{State}, \text{Data}]], s: \text{State}, \\ \text{stmt1}, \text{stmt2}: [\text{State} \rightarrow \text{StmtResult}[\text{State}]]): \\ \text{OK?}(\text{stmt1}(s)) \supset (\text{stmt1} \## \text{lift}(\text{stmt2}) \## \text{expr})(s) = (\text{stmt1} \## \text{stmt2} \## \text{expr})(s)$$

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of stmt_ok_lift_expr.
Q.E.D.

C.204.3 Single_Statement_Rewrites1.break_break_stmt_expr

Terse proof for break_break_stmt_expr.

break_break_stmt_expr:

$\{1\} \quad \forall (\text{expr}: [\text{State} \rightarrow \text{ExprResult}[\text{State}, \text{Data}]], s: \text{State}, \\ \text{stmt}, \text{stmt1}: [\text{State} \rightarrow \text{StmtResult}[\text{State}]]): \\ \text{Break?}(\text{stmt}(s)) \supset \\ (\text{stmt} \## \text{stmt1} \## \text{catch_break} \## \text{expr})(s) = (\text{stmt} \## \text{catch_break} \## \text{expr})(s)$

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `break_break_stmt_expr`.
Q.E.D.

C.204.4

Single_Statement_Rewrites1.break_break_catch_continue_expr

Terse proof for `break_break_catch_continue_expr`.

break_break_catch_continue_expr:

$\{1\} \quad \forall (\text{expr}: [\text{State} \rightarrow \text{ExprResult}[\text{State}, \text{Data}]], s: \text{State}, \\ \text{stmt}: [\text{State} \rightarrow \text{StmtResult}[\text{State}]]): \\ \text{Break?}(\text{stmt}(s)) \supset \\ (\text{stmt} \## \text{catch_continue} \## \text{catch_break} \## \text{expr})(s) = (\text{stmt} \## \text{catch_break} \## \text{expr})(s)$

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `break_break_catch_continue_expr`.
Q.E.D.

C.204.5

Single_Statement_Rewrites1.break_break_catch_default_expr

Terse proof for `break_break_catch_default_expr`.

break_break_catch_default_expr:

$\{1\} \quad \forall (\text{expr}: [\text{State} \rightarrow \text{ExprResult}[\text{State}, \text{Data}]], s: \text{State}, \\ \text{stmt}: [\text{State} \rightarrow \text{StmtResult}[\text{State}]]): \\ \text{Break?}(\text{stmt}(s)) \supset \\ (\text{stmt} \## \text{catch_default} \## \text{catch_break} \## \text{expr})(s) = (\text{stmt} \## \text{catch_break} \## \text{expr})(s)$
--

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `break_break_catch_default_expr`.
Q.E.D.

C.204.6 Single_Statement_Rewrites1.break_break_lift_stmt_expr

Terse proof for `break_break_lift_stmt_expr`.

break_break_lift_stmt_expr:

$\{1\} \quad \forall (\text{expr}: [\text{State} \rightarrow \text{ExprResult}[\text{State}, \text{Data}]], s: \text{State}, \\ \text{stmt}, \text{stmt1}: [\text{State} \rightarrow \text{StmtResult}[\text{State}]]): \\ \text{Break?}(\text{stmt}(s)) \supset \\ (\text{stmt} \## \text{lift}(\text{stmt1}) \## \text{catch_break} \## \text{expr})(s) = (\text{stmt} \## \text{catch_break} \## \text{expr})(s)$
--

Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of `break_break_lift_stmt_expr`.
 Q.E.D.

C.204.7 Single_Statement_Rewrites1.break_break_case_expr

Terse proof for `break_break_case_expr`.

`break_break_case_expr`:

$$\begin{array}{|l} \hline \{1\} \quad \forall (\text{expr}: [\text{State} \rightarrow \text{ExprResult}[\text{State}, \text{Data}]], n: \text{int}, s: \text{State}, \\ \quad \text{stmt}: [\text{State} \rightarrow \text{StmtResult}[\text{State}]]): \\ \quad \text{Break?}(\text{stmt}(s)) \supset \\ \quad (\text{stmt} \## \text{case}(n) \## \text{catch_break} \## \text{expr})(s) = (\text{stmt} \## \text{catch_break} \## \text{expr})(s) \\ \hline \end{array}$$

Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of `break_break_case_expr`.
 Q.E.D.

C.204.8 Single_Statement_Rewrites1.break_break_default_expr

Terse proof for `break_break_default_expr`.

`break_break_default_expr`:

$$\begin{array}{|l} \hline \{1\} \quad \forall (\text{expr}: [\text{State} \rightarrow \text{ExprResult}[\text{State}, \text{Data}]], s: \text{State}, \\ \quad \text{stmt}: [\text{State} \rightarrow \text{StmtResult}[\text{State}]]): \\ \quad \text{Break?}(\text{stmt}(s)) \supset \\ \quad (\text{stmt} \## \text{default} \## \text{catch_break} \## \text{expr})(s) = (\text{stmt} \## \text{catch_break} \## \text{expr})(s) \\ \hline \end{array}$$

Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of `break_break_default_expr`.
 Q.E.D.

C.204.9 Single_Statement_Rewrites1.continue_break_stmt_expr

Terse proof for `continue_break_stmt_expr`.

`continue_break_stmt_expr`:

$$\begin{array}{|l} \hline \{1\} \quad \forall (\text{expr}: [\text{State} \rightarrow \text{ExprResult}[\text{State}, \text{Data}]], s: \text{State}, \\ \quad \text{stmt}, \text{stmt1}: [\text{State} \rightarrow \text{StmtResult}[\text{State}]]): \\ \quad \text{Continue?}(\text{stmt}(s)) \supset \\ \quad (\text{stmt} \## \text{stmt1} \## \text{catch_break} \## \text{expr})(s) = (\text{stmt} \## \text{catch_break} \## \text{expr})(s) \\ \hline \end{array}$$

Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of `continue_break_stmt_expr`.
 Q.E.D.

C.204.10

Single_Statement_Rewrites1.continue_break_catch_continue_expr

Terse proof for `continue_break_catch_continue_expr`.

continue_break_catch_continue_expr:

$$\frac{\{1\} \quad \forall (\text{expr}: [\text{State} \rightarrow \text{ExprResult}[\text{State}, \text{Data}]], s: \text{State}, \text{stmt}: [\text{State} \rightarrow \text{StmtResult}[\text{State}]]) : \text{Continue?}(\text{stmt}(s)) \supset (\text{stmt} \## \text{catch_break} \## \text{catch_continue} \## \text{expr})(s) = (\text{stmt} \## \text{catch_continue} \## \text{expr})(s)}{}{}$$

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `continue_break_catch_continue_expr`.
Q.E.D.

C.204.11

Single_Statement_Rewrites1.continue_break_catch_default_expr

Terse proof for `continue_break_catch_default_expr`.

continue_break_catch_default_expr:

$$\frac{\{1\} \quad \forall (\text{expr}: [\text{State} \rightarrow \text{ExprResult}[\text{State}, \text{Data}]], s: \text{State}, \text{stmt}: [\text{State} \rightarrow \text{StmtResult}[\text{State}]]) : \text{Continue?}(\text{stmt}(s)) \supset (\text{stmt} \## \text{catch_default} \## \text{catch_continue} \## \text{expr})(s) = (\text{stmt} \## \text{catch_continue} \## \text{expr})(s)}{}{}$$

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `continue_break_catch_default_expr`.
Q.E.D.

C.204.12 Single_Statement_Rewrites1.continue_break_lift_stmt_expr

Terse proof for `continue_break_lift_stmt_expr`.

continue_break_lift_stmt_expr:

$$\frac{\{1\} \quad \forall (\text{expr}: [\text{State} \rightarrow \text{ExprResult}[\text{State}, \text{Data}]], s: \text{State}, \text{stmt}, \text{stmt1}: [\text{State} \rightarrow \text{StmtResult}[\text{State}]]) : \text{Continue?}(\text{stmt}(s)) \supset (\text{stmt} \## \text{lift}(\text{stmt1}) \## \text{catch_continue} \## \text{expr})(s) = (\text{stmt} \## \text{catch_continue} \## \text{expr})(s)}{}{}$$

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `continue_break_lift_stmt_expr`.
Q.E.D.

C.204.13 Single_Statement_Rewrites1.continue_break_case_expr

Terse proof for `continue_break_case_expr`.

continue_break_case_expr:

$$\frac{\{1\} \quad \forall (\text{expr}: [\text{State} \rightarrow \text{ExprResult}[\text{State}, \text{Data}]], n: \text{int}, s: \text{State}, \\ \text{stmt}: [\text{State} \rightarrow \text{StmtResult}[\text{State}]]) : \\ \text{Continue?}(\text{stmt}(s)) \supset \\ (\text{stmt} \## \text{case}(n) \## \text{catch_continue} \## \text{expr})(s) = \\ (\text{stmt} \## \text{catch_continue} \## \text{expr})(s)}{}$$

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of continue_break_case_expr.
Q.E.D.

C.204.14 Single_Statement_Rewrites1.continue_break_default_expr

Terse proof for continue_break_default_expr.

continue_break_default_expr:

$$\frac{\{1\} \quad \forall (\text{expr}: [\text{State} \rightarrow \text{ExprResult}[\text{State}, \text{Data}]], s: \text{State}, \\ \text{stmt}: [\text{State} \rightarrow \text{StmtResult}[\text{State}]]) : \\ \text{Continue?}(\text{stmt}(s)) \supset \\ (\text{stmt} \## \text{default} \## \text{catch_continue} \## \text{expr})(s) = (\text{stmt} \## \text{catch_continue} \## \text{expr})(s)}{}$$

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of continue_break_default_expr.
Q.E.D.

C.204.15 Single_Statement_Rewrites1.while_unroll_expr_TCC1

Terse proof for while_unroll_expr_TCC1.

while_unroll_expr_TCC1:

$$\frac{\{1\} \quad \forall (\text{max}: \text{nat}): \text{max} > 0 \supset \text{max} - 1 \geq 0}{}$$

Repeatedly Skolemizing and flattening,
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of while_unroll_expr_TCC1.
Q.E.D.

C.204.16 Single_Statement_Rewrites1.while_unroll_expr

Terse proof for while_unroll_expr.

`while_unroll_expr`:

<pre> {1} ∃ (b_ex: [State → ExprResult[State, bool]], expr: [State → ExprResult[State, Data]]), max: nat, s: State, stmt: [State → StmtResult[State]]): max > 0 ⊃ (while(max, b_ex, stmt) ## expr)(s) = (if_else(b_ex, stmt ## catch_continue ## while(max - 1, b_ex, stmt) ## catch_break, skip) ## expr) (s) </pre>
--

Repeatedly Skolemizing and flattening,
 Expanding the definition of ##,
 Rewriting using `while_unroll`, matching in *,
 This completes the proof of `while_unroll_expr`.
 Q.E.D.

C.204.17 Single_Statement_Rewrites1.while_unroll_lexpr

Terse proof for `while_unroll_lexpr`.

`while_unroll_lexpr`:

<pre> {1} ∃ (b_ex: [State → ExprResult[State, bool]], lexpr: [StmtResult[State] → ExprResult[State, Data]], max: nat, s: State, stmt: [State → StmtResult[State]]): max > 0 ⊃ (while(max, b_ex, stmt) ## lexpr)(s) = (if_else(b_ex, stmt ## catch_continue ## while(max - 1, b_ex, stmt) ## catch_break, skip) ## lexpr) (s) </pre>
--

Repeatedly Skolemizing and flattening,
 Expanding the definition of ##,
 Rewriting using `while_unroll`, matching in *,
 This completes the proof of `while_unroll_lexpr`.
 Q.E.D.

C.204.18 Single_Statement_Rewrites1.stmt_while_unroll_expr

Terse proof for `stmt_while_unroll_expr`.

stmt_while_unroll_expr:

<pre> {1} ∃ (b_ex: [State → ExprResult[State, bool]], expr: [State → ExprResult[State, Data]], max: nat, s: State, stmt1, stmt2: [State → StmtResult[State]]): max > 0 ⊃ (stmt1 ## while(max, b_ex, stmt2) ## expr)(s) = (stmt1 ## if_else(b_ex, stmt2 ## catch_continue ## while(max - 1, b_ex, stmt2) ## catch_break, skip) ## expr) (s) </pre>
--

Repeatedly Skolemizing and flattening,
 Expanding the definition of ##,
 Expanding the definition of while,
 Using lemma while_as_if_while,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 This completes the proof of stmt_while_unroll_expr.
 Q.E.D.

C.204.19 Single_Statement_Rewrites1.stmt_while_unroll_lexpr

Terse proof for stmt_while_unroll_lexpr.

stmt_while_unroll_lexpr:

<pre> {1} ∃ (b_ex: [State → ExprResult[State, bool]], lexpr: [StmtResult[State] → ExprResult[State, Data]], max: nat, s: State, stmt1, stmt2: [State → StmtResult[State]]): max > 0 ⊃ (stmt1 ## while(max, b_ex, stmt2) ## lexpr)(s) = (stmt1 ## if_else(b_ex, stmt2 ## catch_continue ## while(max - 1, b_ex, stmt2) ## catch_break, skip) ## lexpr) (s) </pre>

Repeatedly Skolemizing and flattening,
 Expanding the definition of ##,
 Expanding the definition of while,
 Using lemma while_as_if_while,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 This completes the proof of stmt_while_unroll_lexpr.
 Q.E.D.

C.204.20 Single_Statement_Rewrites1.e2s_ok

Terse proof for e2s_ok.

e2s_ok:

$$\{1\} \quad \forall (\text{expr}: [\text{State} \rightarrow \text{ExprResult}[\text{State}, \text{Data}]], s: \text{State}):$$

$$\text{OK}?(\text{expr}(s)) \supset \text{OK}?(\text{e2s}(\text{expr})(s))$$

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of e2s_ok.

Q.E.D.

C.204.21 Single_Statement_Rewrites1.stmt_e2s_ok

Terse proof for stmt_e2s_ok.

stmt_e2s_ok:

$$\{1\} \quad \forall (\text{expr}: [\text{State} \rightarrow \text{ExprResult}[\text{State}, \text{Data}]], s: \text{State},$$

$$\text{stmt}: [\text{State} \rightarrow \text{StmtResult}[\text{State}]]):$$

$$\text{OK}?((\text{stmt} \ \#\# \ \text{expr})(s)) \supset \text{OK}?((\text{stmt} \ \#\# \ \text{e2s}(\text{expr}))(s))$$

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of stmt_e2s_ok.

Q.E.D.

C.204.22 Single_Statement_Rewrites1.e2s_state_TCC1

Terse proof for e2s_state_TCC1.

e2s_state_TCC1:

$$\{1\} \quad \forall (\text{expr}: [\text{State} \rightarrow \text{ExprResult}[\text{State}, \text{Data}]], s: \text{State}):$$

$$\text{OK}?(\text{expr}(s)) \supset$$

$$\text{OK}?[\text{State}](\text{e2s}[\text{State}, \text{Data}](\text{expr}(s))) \vee$$

$$\text{Break}?[\text{State}](\text{e2s}[\text{State}, \text{Data}](\text{expr}(s))) \vee$$

$$\text{Continue}?[\text{State}](\text{e2s}[\text{State}, \text{Data}](\text{expr}(s))) \vee$$

$$\text{Return}?[\text{State}](\text{e2s}[\text{State}, \text{Data}](\text{expr}(s))) \vee$$

$$\text{Switch}?[\text{State}](\text{e2s}[\text{State}, \text{Data}](\text{expr}(s))) \vee$$

$$\text{Default}?[\text{State}](\text{e2s}[\text{State}, \text{Data}](\text{expr}(s))) \vee$$

$$\text{Exception}?[\text{State}](\text{e2s}[\text{State}, \text{Data}](\text{expr}(s)))$$

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of e2s_state_TCC1.

Q.E.D.

C.204.23 Single_Statement_Rewrites1.e2s_state_TCC2

Terse proof for e2s_state_TCC2.

e2s_state_TCC2:

$$\{1\} \quad \forall (\text{expr}: [\text{State} \rightarrow \text{ExprResult}[\text{State}, \text{Data}]], s: \text{State}):$$

$$\text{OK}?(\text{expr}(s)) \supset \text{OK}?[\text{State}, \text{Data}](\text{expr}(s)) \vee \text{Exception}?[\text{State}, \text{Data}](\text{expr}(s))$$

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of e2s_state_TCC2.

Q.E.D.

C.204.24 Single_Statement_Rewrites1.e2s_state

Terse proof for e2s_state.

e2s_state:

$\{1\} \quad \forall (\text{expr}: [\text{State} \rightarrow \text{ExprResult}[\text{State}, \text{Data}]], s: \text{State}):$ $\text{OK}?(\text{expr}(s)) \supset \text{state}(\text{e2s}(\text{expr})(s)) = \text{state}(\text{expr}(s))$

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of e2s_state.

Q.E.D.

C.204.25 Single_Statement_Rewrites1.stmt_e2s_state_TCC1

Terse proof for stmt_e2s_state_TCC1.

stmt_e2s_state_TCC1:

$\{1\} \quad \forall (\text{expr}: [\text{State} \rightarrow \text{ExprResult}[\text{State}, \text{Data}]], s: \text{State},$ $\text{stmt}: [\text{State} \rightarrow \text{StmtResult}[\text{State}]]):$ $\text{OK}?((\text{stmt} \ \#\# \ \text{expr})(s)) \supset$ $\text{OK}?[\text{State}]((\#\# [\text{State}] (\text{stmt}, \text{e2s}[\text{State}, \text{Data}] (\text{expr})))(s)) \vee$ $\text{Break}?[\text{State}]((\#\# [\text{State}] (\text{stmt}, \text{e2s}[\text{State}, \text{Data}] (\text{expr})))(s)) \vee$ $\text{Continue}?[\text{State}]((\#\# [\text{State}] (\text{stmt}, \text{e2s}[\text{State}, \text{Data}] (\text{expr})))(s)) \vee$ $\text{Return}?[\text{State}]((\#\# [\text{State}] (\text{stmt}, \text{e2s}[\text{State}, \text{Data}] (\text{expr})))(s)) \vee$ $\text{Switch}?[\text{State}]((\#\# [\text{State}] (\text{stmt}, \text{e2s}[\text{State}, \text{Data}] (\text{expr})))(s)) \vee$ $\text{Default}?[\text{State}]((\#\# [\text{State}] (\text{stmt}, \text{e2s}[\text{State}, \text{Data}] (\text{expr})))(s)) \vee$ $\text{Exception}?[\text{State}]((\#\# [\text{State}] (\text{stmt}, \text{e2s}[\text{State}, \text{Data}] (\text{expr})))(s))$

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of stmt_e2s_state_TCC1.

Q.E.D.

C.204.26 Single_Statement_Rewrites1.stmt_e2s_state_TCC2

Terse proof for stmt_e2s_state_TCC2.

stmt_e2s_state_TCC2:

$\{1\} \quad \forall (\text{expr}: [\text{State} \rightarrow \text{ExprResult}[\text{State}, \text{Data}]], s: \text{State},$ $\text{stmt}: [\text{State} \rightarrow \text{StmtResult}[\text{State}]]):$ $\text{OK}?((\text{stmt} \ \#\# \ \text{expr})(s)) \supset$ $\text{OK}?[\text{State}, \text{Data}]((\#\# [\text{State}, \text{Data}] (\text{stmt}, \text{expr})(s)) \vee$ $\text{Exception}?[\text{State}, \text{Data}]((\#\# [\text{State}, \text{Data}] (\text{stmt}, \text{expr})(s))$

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of stmt_e2s_state_TCC2.

Q.E.D.

C.204.27 Single_Statement_Rewrites1.stmt_e2s_state

Terse proof for `stmt_e2s_state`.

`stmt_e2s_state`:

$$\{1\} \quad \forall (\text{expr}: [\text{State} \rightarrow \text{ExprResult}[\text{State}, \text{Data}]], s: \text{State}, \\ \text{stmt}: [\text{State} \rightarrow \text{StmtResult}[\text{State}]]) : \\ \text{OK?}((\text{stmt} \## \text{expr})(s)) \supset \\ \text{state}((\text{stmt} \## \text{e2s}(\text{expr}))(s)) = \text{state}((\text{stmt} \## \text{expr})(s))$$

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `stmt_e2s_state`.

Q.E.D.

C.204.28 Single_Statement_Rewrites1.comp_eval_if_ok_fstmt

Terse proof for `comp_eval_if_ok_fstmt`.

`comp_eval_if_ok_fstmt`:

$$\{1\} \quad \forall (\text{expr}: [\text{State} \rightarrow \text{ExprResult}[\text{State}, \text{Data}]], \\ \text{fstmt}: [\text{Data} \rightarrow [\text{State} \rightarrow \text{StmtResult}[\text{State}]]], s: \text{State}) : \\ \text{OK?}(\text{expr}(s)) \supset \\ (\text{expr} \## \text{fstmt})(s) = (\text{e2s}(\text{expr}) \## \text{fstmt}(\text{data}(\text{expr}(s))))(s)$$

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `comp_eval_if_ok_fstmt`.

Q.E.D.

C.204.29 Single_Statement_Rewrites1.stmt_eval_if_ok_fstmt

Terse proof for `stmt_eval_if_ok_fstmt`.

`stmt_eval_if_ok_fstmt`:

$$\{1\} \quad \forall (\text{expr}: [\text{State} \rightarrow \text{ExprResult}[\text{State}, \text{Data}]], \\ \text{fstmt}: [\text{Data} \rightarrow [\text{State} \rightarrow \text{StmtResult}[\text{State}]]], s: \text{State}, \\ \text{stmt}: [\text{State} \rightarrow \text{StmtResult}[\text{State}]]) : \\ \text{OK?}((\text{stmt} \## \text{expr})(s)) \supset \\ (\text{stmt} \## (\text{expr} \## \text{fstmt}))(s) = \\ (\text{stmt} \## (\text{e2s}(\text{expr}) \## \text{fstmt}(\text{data}((\text{stmt} \## \text{expr})(s))))) (s)$$

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `stmt_eval_if_ok_fstmt`.

Q.E.D.

C.204.30 Single_Statement_Rewrites1.comp_eval_if_ok_fstmt_cstmt

Terse proof for `comp_eval_if_ok_fstmt_cstmt`.

comp_eval_if_ok_fstmt_cstmt:

$\{1\} \quad \forall (\text{cstmt}: [\text{StmtResult}[\text{State}] \rightarrow \text{StmtResult}[\text{State}]],$ $\quad \text{expr}: [\text{State} \rightarrow \text{ExprResult}[\text{State}, \text{Data}]],$ $\quad \text{fstmt}: [\text{Data} \rightarrow [\text{State} \rightarrow \text{StmtResult}[\text{State}]]], s: \text{State}):$ $\text{OK?}(\text{expr}(s)) \supset$ $((\text{expr} \## \text{fstmt}) \## \text{cstmt})(s) =$ $(\text{e2s}(\text{expr}) \## \text{fstmt}(\text{data}(\text{expr}(s))) \## \text{cstmt})(s)$

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of comp_eval_if_ok_fstmt_cstmt.
Q.E.D.

C.204.31 Single_Statement_Rewrites1.stmt_eval_if_ok_fstmt_cstmt

Terse proof for stmt_eval_if_ok_fstmt_cstmt.

stmt_eval_if_ok_fstmt_cstmt:

$\{1\} \quad \forall (\text{cstmt}: [\text{StmtResult}[\text{State}] \rightarrow \text{StmtResult}[\text{State}]],$ $\quad \text{expr}: [\text{State} \rightarrow \text{ExprResult}[\text{State}, \text{Data}]],$ $\quad \text{fstmt}: [\text{Data} \rightarrow [\text{State} \rightarrow \text{StmtResult}[\text{State}]]], s: \text{State},$ $\quad \text{stmt}: [\text{State} \rightarrow \text{StmtResult}[\text{State}]]):$ $\text{OK?}((\text{stmt} \## \text{expr})(s)) \supset$ $(\text{stmt} \## (\text{expr} \## \text{fstmt}) \## \text{cstmt})(s) =$ $(\text{stmt} \## \text{e2s}(\text{expr}) \## \text{fstmt}(\text{data}((\text{stmt} \## \text{expr})(s))) \## \text{cstmt})(s)$
--

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of stmt_eval_if_ok_fstmt_cstmt.
Q.E.D.

C.204.32

Single_Statement_Rewrites1.composition_assoc_rewrite_expr_1

Terse proof for composition_assoc_rewrite_expr_1.

composition_assoc_rewrite_expr_1:

$\{1\} \quad \forall (\text{expr}: [\text{State} \rightarrow \text{ExprResult}[\text{State}, \text{Data}]], \text{stmt}, \text{stmt1}: [\text{State} \rightarrow \text{StmtResult}[\text{State}]]):$ $(\text{stmt} \## (\text{stmt1} \## \text{expr})) = ((\text{stmt} \## \text{stmt1}) \## \text{expr})$
--

Repeatedly Skolemizing and flattening,
Applying decompose-equality,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of composition_assoc_rewrite_expr_1.
Q.E.D.

C.204.33

Single_Statement_Rewrites1.composition_assoc_rewrite_expr_2

Terse proof for composition_assoc_rewrite_expr_2.

composition_assoc_rewrite_expr_2:

$$\{1\} \quad \forall (cstmt: [\text{StmtResult}[\text{State}] \rightarrow \text{StmtResult}[\text{State}]], \\ \text{expr}: [\text{State} \rightarrow \text{ExprResult}[\text{State}, \text{Data}]], \text{stmt}: [\text{State} \rightarrow \text{StmtResult}[\text{State}]]):$$

$$(\text{stmt} \## (\text{cstmt} \## \text{expr})) = ((\text{stmt} \## \text{cstmt}) \## \text{expr})$$

Repeatedly Skolemizing and flattening,
 Applying decompose-equality,
 Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of composition_assoc_rewrite_expr_2.
 Q.E.D.

C.204.34

Single_Statement_Rewrites1.composition_assoc_rewrite_expr_3

Terse proof for composition_assoc_rewrite_expr_3.

composition_assoc_rewrite_expr_3:

$$\{1\} \quad \forall (cstmt, cstmt1: [\text{StmtResult}[\text{State}] \rightarrow \text{StmtResult}[\text{State}]], \\ \text{expr}: [\text{State} \rightarrow \text{ExprResult}[\text{State}, \text{Data}]]):$$

$$(\text{cstmt} \## (\text{cstmt1} \## \text{expr})) = ((\text{cstmt} \## \text{cstmt1}) \## \text{expr})$$

Repeatedly Skolemizing and flattening,
 Applying decompose-equality,
 Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of composition_assoc_rewrite_expr_3.
 Q.E.D.

C.204.35

Single_Statement_Rewrites1.composition_assoc_rewrite_expr_4

Terse proof for composition_assoc_rewrite_expr_4.

composition_assoc_rewrite_expr_4:

$$\{1\} \quad \forall (\text{lexpr}: [\text{StmtResult}[\text{State}] \rightarrow \text{ExprResult}[\text{State}, \text{Data}]], \\ \text{stmt}, \text{stmt1}: [\text{State} \rightarrow \text{StmtResult}[\text{State}]]):$$

$$(\text{stmt} \## (\text{lift}(\text{stmt1}) \## \text{lexpr})) = ((\text{stmt} \## \text{stmt1}) \## \text{lexpr})$$

,
 Applying decompose-equality,
 This completes the proof of composition_assoc_rewrite_expr_4.
 Q.E.D.

C.204.36

Single_Statement_Rewrites1.composition_assoc_rewrite_expr_5

Terse proof for composition_assoc_rewrite_expr_5.

composition_assoc_rewrite_expr_5:

$$\{1\} \quad \forall (\text{cstmt}: [\text{StmtResult}[\text{State}] \rightarrow \text{StmtResult}[\text{State}]], \\ \text{lexpr}: [\text{StmtResult}[\text{State}] \rightarrow \text{ExprResult}[\text{State}, \text{Data}]], \\ \text{stmt}: [\text{State} \rightarrow \text{StmtResult}[\text{State}]]): \\ (\text{stmt} \## (\text{cstmt} \## \text{lexpr})) = ((\text{stmt} \## \text{cstmt}) \## \text{lexpr})$$

Repeatedly Skolemizing and flattening,
 Applying decompose-equality,
 Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of `composition_assoc_rewrite_expr_5`.
 Q.E.D.

C.204.37

Single_Statement_Rewrites1.composition_assoc_rewrite_expr_6

Terse proof for `composition_assoc_rewrite_expr_6`.

composition_assoc_rewrite_expr_6:

$$\{1\} \quad \forall (\text{cstmt}, \text{cstmt1}: [\text{StmtResult}[\text{State}] \rightarrow \text{StmtResult}[\text{State}]], \\ \text{lexpr}: [\text{StmtResult}[\text{State}] \rightarrow \text{ExprResult}[\text{State}, \text{Data}]]): \\ (\text{cstmt} \## (\text{cstmt1} \## \text{lexpr})) = ((\text{cstmt} \## \text{cstmt1}) \## \text{lexpr})$$

Repeatedly Skolemizing and flattening,
 Applying decompose-equality,
 Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of `composition_assoc_rewrite_expr_6`.
 Q.E.D.

C.204.38

Single_Statement_Rewrites1.composition_assoc_rewrite_stmt_ok

Terse proof for `composition_assoc_rewrite_stmt_ok`.

composition_assoc_rewrite_stmt_ok:

$$\{1\} \quad \forall (\text{cstmt}: [\text{StmtResult}[\text{State}] \rightarrow \text{StmtResult}[\text{State}]], s: \text{State}, \\ \text{stmt}, \text{stmt1}: [\text{State} \rightarrow \text{StmtResult}[\text{State}]]): \\ \text{OK?}(\text{stmt}(s)) \supset (\text{stmt} \## (\text{stmt1} \## \text{cstmt}))(s) = ((\text{stmt} \## \text{stmt1}) \## \text{cstmt})(s)$$

Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of `composition_assoc_rewrite_stmt_ok`.
 Q.E.D.

C.204.39 Sin-

gle_Statement_Rewrites1.composition_assoc_rewrite_stmt_ok_expr

Terse proof for `composition_assoc_rewrite_stmt_ok_expr`.

composition_assoc_rewrite_stmt_ok_expr:

$$\{1\} \quad \forall (cstmt: [\text{StmtResult}[\text{State}] \rightarrow \text{StmtResult}[\text{State}]], \\ \text{expr}: [\text{State} \rightarrow \text{ExprResult}[\text{State}, \text{Data}]], s: \text{State}, \\ \text{stmt}, \text{stmt1}: [\text{State} \rightarrow \text{StmtResult}[\text{State}]]): \\ \text{OK}?(stmt(s)) \supset \\ (stmt \## (stmt1 \## cstmt) \## expr)(s) = ((stmt \## stmt1) \## cstmt \## expr)(s)$$

,
This completes the proof of composition_assoc_rewrite_stmt_ok_expr.
Q.E.D.

C.204.40 Single_Statement_Rewrites1.composition_assoc_rewrite_stmt_ok_expr_lexpr

Terse proof for composition_assoc_rewrite_stmt_ok_expr_lexpr.

composition_assoc_rewrite_stmt_ok_expr_lexpr:

$$\{1\} \quad \forall (cstmt: [\text{StmtResult}[\text{State}] \rightarrow \text{StmtResult}[\text{State}]], \\ \text{lexpr}: [\text{StmtResult}[\text{State}] \rightarrow \text{ExprResult}[\text{State}, \text{Data}]], s: \text{State}, \\ \text{stmt}, \text{stmt1}: [\text{State} \rightarrow \text{StmtResult}[\text{State}]]): \\ \text{OK}?(stmt(s)) \supset \\ (stmt \## (stmt1 \## cstmt) \## lexpr)(s) = ((stmt \## stmt1) \## cstmt \## lexpr)(s)$$

,
This completes the proof of composition_assoc_rewrite_stmt_ok_expr_lexpr.
Q.E.D.

C.204.41 Single_Statement_Rewrites1.ok_result_fstmt

Terse proof for ok_result_fstmt.

ok_result_fstmt:

$$\{1\} \quad \forall (fstmt: [\text{Data} \rightarrow [\text{State} \rightarrow \text{StmtResult}[\text{State}]]], \text{data}: \text{Data}): \\ \text{ok_result}(\text{data}) \## \text{fstmt} = \text{fstmt}(\text{data})$$

Repeatedly Skolemizing and flattening,
Applying decompose-equality,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of ok_result_fstmt.
Q.E.D.

C.204.42 Single_Statement_Rewrites1.pm_q_prop_read_ok_s_read_ok

Terse proof for pm_q_prop_read_ok_s_read_ok.

pm_q_prop_read_ok_s_read_ok:

$$\{1\} \quad \forall (\text{addr}: \text{Address}, \text{dt}: (\text{interpreted_data_type}?\text{[Data]}), \text{pm}: \text{Plain_Memory}[\text{State}], s: \text{State}): \\ \text{pm_q_prop_read}(\text{pm}, \text{dt}, \text{addr})(\text{skip}(s)) \supset \text{OK}?(read_data(\text{pm}, \text{dt})(\text{addr})(s))$$

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of pm_q_prop_read_ok_s_read_ok.

Q.E.D.

C.205 Proofs for Single_Statement_Rewrites2 (statement-rewrites.pvs)

C.205.1 Single_Statement_Rewrites2.ok_result_fexpr

Terse proof for `ok_result_fexpr`.

`ok_result_fexpr`:

$$\frac{\{1\} \quad \forall (\text{fexpr}: [\text{Data2} \rightarrow [\text{State} \rightarrow \text{ExprResult}[\text{State}, \text{Data1}]]], \text{data}: \text{Data2}):}{\text{ok_result}(\text{data}) \ \#\# \ \text{fexpr} = \text{fexpr}(\text{data})}$$

Repeatedly Skolemizing and flattening,
 Applying decompose-equality,
 Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of `ok_result_fexpr`.
 Q.E.D.

C.205.2 Single_Statement_Rewrites2.e2s_merge

Terse proof for `e2s_merge`.

`e2s_merge`:

$$\frac{\{1\} \quad \forall (\text{expr}: [\text{State} \rightarrow \text{ExprResult}[\text{State}, \text{Data1}]], \text{expr1}: [\text{State} \rightarrow \text{ExprResult}[\text{State}, \text{Data2}]]):}{(\text{e2s}(\text{expr}) \ \#\# \ \text{e2s}(\text{expr1})) = \text{e2s}(\text{expr} \ \#\# \ \text{expr1})}$$

Repeatedly Skolemizing and flattening,
 Applying decompose-equality,
 Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of `e2s_merge`.
 Q.E.D.

C.205.3 Single_Statement_Rewrites2.stmt_e2s_merge

Terse proof for `stmt_e2s_merge`.

`stmt_e2s_merge`:

$$\frac{\{1\} \quad \forall (\text{expr}: [\text{State} \rightarrow \text{ExprResult}[\text{State}, \text{Data1}]], \text{expr1}: [\text{State} \rightarrow \text{ExprResult}[\text{State}, \text{Data2}]], \text{stmt}: [\text{State} \rightarrow \text{StmtResult}[\text{State}]]):}{(\text{stmt} \ \#\# \ \text{e2s}(\text{expr}) \ \#\# \ \text{e2s}(\text{expr1})) = (\text{stmt} \ \#\# \ \text{e2s}(\text{expr} \ \#\# \ \text{expr1}))}$$

Repeatedly Skolemizing and flattening,
 Applying decompose-equality,
 Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of `stmt_e2s_merge`.
 Q.E.D.

C.205.4 Single_Statement_Rewrites2.e2s_expr

Terse proof for e2s_expr.

e2s_expr:

$\{1\} \quad \forall (\text{expr}: [\text{State} \rightarrow \text{ExprResult}[\text{State}, \text{Data1}]], \text{expr1}: [\text{State} \rightarrow \text{ExprResult}[\text{State}, \text{Data2}]], \\ s: \text{State}): \\ (\text{e2s}(\text{expr1}) \## \text{expr})(s) = (\text{expr1} \## \text{expr})(s)$

,
This completes the proof of e2s_expr.
Q.E.D.

C.205.5 Single_Statement_Rewrites2.stmt_e2s_expr

Terse proof for stmt_e2s_expr.

stmt_e2s_expr:

$\{1\} \quad \forall (\text{expr}: [\text{State} \rightarrow \text{ExprResult}[\text{State}, \text{Data1}]], \text{expr1}: [\text{State} \rightarrow \text{ExprResult}[\text{State}, \text{Data2}]], \\ s: \text{State}, \text{stmt}: [\text{State} \rightarrow \text{StmtResult}[\text{State}]]): \\ (\text{stmt} \## \text{e2s}(\text{expr1}) \## \text{expr})(s) = (\text{stmt} \## \text{expr1} \## \text{expr})(s)$
--

,
This completes the proof of stmt_e2s_expr.
Q.E.D.

C.205.6 Single_Statement_Rewrites2.composition_assoc_expression_rewrite_stmt

Terse proof for composition_assoc_expression_rewrite_stmt.

composition_assoc_expression_rewrite_stmt:

$\{1\} \quad \forall (\text{expr}: [\text{State} \rightarrow \text{ExprResult}[\text{State}, \text{Data1}]], \text{expr1}: [\text{State} \rightarrow \text{ExprResult}[\text{State}, \text{Data2}]], \\ \text{stmt}: [\text{State} \rightarrow \text{StmtResult}[\text{State}]]): \\ (\text{stmt} \## (\text{expr} \## \text{expr1})) = ((\text{stmt} \## \text{expr}) \## \text{expr1})$

Repeatedly Skolemizing and flattening,
Applying decompose-equality,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of composition_assoc_expression_rewrite_stmt.
Q.E.D.

C.205.7 Single_Statement_Rewrites2.comp_eval_if_ok_fstmt_expr

Terse proof for comp_eval_if_ok_fstmt_expr.

comp_eval_if_ok_fstmt_expr:

$$\{1\} \quad \forall (\text{expr}: [\text{State} \rightarrow \text{ExprResult}[\text{State}, \text{Data1}]], \text{expr1}: [\text{State} \rightarrow \text{ExprResult}[\text{State}, \text{Data2}]], \\ \text{fstmt}: [\text{Data1} \rightarrow [\text{State} \rightarrow \text{StmtResult}[\text{State}]]], s: \text{State}): \\ \text{OK?}(\text{expr}(s)) \supset \\ ((\text{expr} \## \text{fstmt}) \## \text{expr1})(s) = \\ (\text{e2s}(\text{expr}) \## \text{fstmt}(\text{data}(\text{expr}(s))) \## \text{expr1})(s)$$

,
This completes the proof of comp_eval_if_ok_fstmt_expr.
Q.E.D.

C.205.8 Single_Statement_Rewrites2.stmt_eval_if_ok_fstmt_expr

Terse proof for stmt_eval_if_ok_fstmt_expr.

stmt_eval_if_ok_fstmt_expr:

$$\{1\} \quad \forall (\text{expr}: [\text{State} \rightarrow \text{ExprResult}[\text{State}, \text{Data1}]], \text{expr1}: [\text{State} \rightarrow \text{ExprResult}[\text{State}, \text{Data2}]], \\ \text{fstmt}: [\text{Data1} \rightarrow [\text{State} \rightarrow \text{StmtResult}[\text{State}]]], s: \text{State}, \\ \text{stmt}: [\text{State} \rightarrow \text{StmtResult}[\text{State}]]): \\ \text{OK?}((\text{stmt} \## \text{expr})(s)) \supset \\ (\text{stmt} \## (\text{expr} \## \text{fstmt}) \## \text{expr1})(s) = \\ (\text{stmt} \## \text{e2s}(\text{expr}) \## \text{fstmt}(\text{data}((\text{stmt} \## \text{expr})(s))) \## \text{expr1})(s)$$

,
This completes the proof of stmt_eval_if_ok_fstmt_expr.
Q.E.D.

C.205.9 Single_Statement_Rewrites2.stmt_eval_if_ok_fexpr

Terse proof for stmt_eval_if_ok_fexpr.

stmt_eval_if_ok_fexpr:

$$\{1\} \quad \forall (\text{expr1}: [\text{State} \rightarrow \text{ExprResult}[\text{State}, \text{Data2}]], \\ \text{fexpr}: [\text{Data2} \rightarrow [\text{State} \rightarrow \text{ExprResult}[\text{State}, \text{Data1}]]], s: \text{State}, \\ \text{stmt}: [\text{State} \rightarrow \text{StmtResult}[\text{State}]]): \\ \text{OK?}((\text{stmt} \## \text{expr1})(s)) \supset \\ (\text{stmt} \## (\text{expr1} \## \text{fexpr}))(s) = \\ (\text{stmt} \## \text{expr1} \## \text{fexpr}(\text{data}((\text{stmt} \## \text{expr1})(s))))(s)$$

,
This completes the proof of stmt_eval_if_ok_fexpr.
Q.E.D.

C.205.10

Single_Statement_Rewrites2.comp_eval_if_ok_fstmt_cstmt_expr

Terse proof for comp_eval_if_ok_fstmt_cstmt_expr.

comp_eval_if_ok_fstmt_cstmt_expr:

{1} \forall (cstmt: [StmtResult[State] \rightarrow StmtResult[State]],
 expr: [State \rightarrow ExprResult[State, Data1]], expr1: [State \rightarrow ExprResult[State, Data2]],
 fstmt: [Data1 \rightarrow [State \rightarrow StmtResult[State]]], s: State):
 OK?(expr(s)) \supset
 ((expr ## fstmt) ## cstmt) ## expr1(s) =
 ((e2s(expr) ## fstmt(data(expr(s)))) ## cstmt) ## expr1(s)

,
 This completes the proof of comp_eval_if_ok_fstmt_cstmt_expr.
 Q.E.D.

C.205.11

Single_Statement_Rewrites2.stmt_eval_if_ok_fstmt_cstmt_expr

Terse proof for stmt_eval_if_ok_fstmt_cstmt_expr.

stmt_eval_if_ok_fstmt_cstmt_expr:

{1} \forall (cstmt: [StmtResult[State] \rightarrow StmtResult[State]],
 expr: [State \rightarrow ExprResult[State, Data1]], expr1: [State \rightarrow ExprResult[State, Data2]],
 fstmt: [Data1 \rightarrow [State \rightarrow StmtResult[State]]], s: State,
 stmt: [State \rightarrow StmtResult[State]]):
 OK?((stmt ## expr)(s)) \supset
 (stmt ## (expr ## fstmt) ## cstmt ## expr1)(s) =
 (stmt ## e2s(expr) ## fstmt(data((stmt ## expr)(s))) ## cstmt ## expr1)(s)

,
 This completes the proof of stmt_eval_if_ok_fstmt_cstmt_expr.
 Q.E.D.

C.206 Proofs for Single_Statement_Rewrites3 (statement-rewrites.pvs)

C.206.1 Single_Statement_Rewrites3.ok_result_elimination_2

Terse proof for ok_result_elimination_2.

ok_result_elimination_2:

{1} \forall (expr: [State \rightarrow ExprResult[State, Data1]], expr1: [State \rightarrow ExprResult[State, Data2]],
 data: Data3):
 ((expr ## ok_result[State, Data3](data)) ## expr1) = (expr ## expr1)

Repeatedly Skolemizing and flattening,
 Applying decompose-equality,
 Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of ok_result_elimination_2.

Q.E.D.

C.206.2 Single_Statement_Rewrites3.comp_eval_if_ok_fexpr_expr

Terse proof for comp_eval_if_ok_fexpr_expr.

comp_eval_if_ok_fexpr_expr:

$\{1\} \quad \forall (\text{expr}: [\text{State} \rightarrow \text{ExprResult}[\text{State}, \text{Data1}]], \text{expr1}: [\text{State} \rightarrow \text{ExprResult}[\text{State}, \text{Data2}]], \\ \text{fexpr}: [\text{Data1} \rightarrow [\text{State} \rightarrow \text{ExprResult}[\text{State}, \text{Data3}]]], s: \text{State}): \\ \text{OK?}(\text{expr}(s)) \supset \\ ((\text{expr} \## \text{fexpr}) \## \text{expr1})(s) = (\text{expr} \## \text{fexpr}(\text{data}(\text{expr}(s))) \## \text{expr1})(s)$

Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of comp_eval_if_ok_fexpr_expr.
 Q.E.D.

C.206.3 Single_Statement_Rewrites3.expr_eval_if_ok_fexpr

Terse proof for expr_eval_if_ok_fexpr.

expr_eval_if_ok_fexpr:

$\{1\} \quad \forall (\text{expr}: [\text{State} \rightarrow \text{ExprResult}[\text{State}, \text{Data1}]], \text{expr1}: [\text{State} \rightarrow \text{ExprResult}[\text{State}, \text{Data2}]], \\ \text{fexpr}: [\text{Data1} \rightarrow [\text{State} \rightarrow \text{ExprResult}[\text{State}, \text{Data3}]]], s: \text{State}): \\ \text{OK?}((\text{expr1} \## \text{expr})(s)) \supset \\ (\text{expr1} \## (\text{expr} \## \text{fexpr}))(s) = \\ (\text{expr1} \## \text{expr} \## \text{fexpr}(\text{data}((\text{expr1} \## \text{expr})(s))))(s)$
--

Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of expr_eval_if_ok_fexpr.
 Q.E.D.

C.206.4

Single_Statement_Rewrites3.composition_assoc_expression_rewrite

Terse proof for composition_assoc_expression_rewrite.

composition_assoc_expression_rewrite:

$\{1\} \quad \forall (\text{expr}: [\text{State} \rightarrow \text{ExprResult}[\text{State}, \text{Data1}]], \text{expr1}: [\text{State} \rightarrow \text{ExprResult}[\text{State}, \text{Data2}]], \\ \text{expr2}: [\text{State} \rightarrow \text{ExprResult}[\text{State}, \text{Data3}]]): \\ (\text{expr} \## (\text{expr1} \## \text{expr2})) = ((\text{expr} \## \text{expr1}) \## \text{expr2})$
--

Trying repeated skolemization, instantiation, and if-lifting,
 Applying decompose-equality,
 Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of composition_assoc_expression_rewrite.
 Q.E.D.

C.206.5 Single_Statement_Rewrites3.stmt_eval_if_ok_fexpr_expr

Terse proof for `stmt_eval_if_ok_fexpr_expr`.

`stmt_eval_if_ok_fexpr_expr`:

$\{1\} \quad \forall (\text{expr}: [\text{State} \rightarrow \text{ExprResult}[\text{State}, \text{Data1}]], \text{expr1}: [\text{State} \rightarrow \text{ExprResult}[\text{State}, \text{Data2}]],$ $\quad \text{fexpr}: [\text{Data1} \rightarrow [\text{State} \rightarrow \text{ExprResult}[\text{State}, \text{Data3}]]], s: \text{State},$ $\quad \text{stmt}: [\text{State} \rightarrow \text{StmtResult}[\text{State}]]):$ $\text{OK?}((\text{stmt} \ \#\# \ \text{expr})(s)) \supset$ $(\text{stmt} \ \#\# \ (\text{expr} \ \#\# \ \text{fexpr}) \ \#\# \ \text{expr1})(s) =$ $(\text{stmt} \ \#\# \ \text{expr} \ \#\# \ \text{fexpr}(\text{data}((\text{stmt} \ \#\# \ \text{expr})(s)))) \ \#\# \ \text{expr1})(s)$
--

This completes the proof of `stmt_eval_if_ok_fexpr_expr`.
Q.E.D.

C.207 Proofs for Single_Statement_Rewrites4 (statement-rewrites.pvs)

C.207.1 Single_Statement_Rewrites4.expr_eval_if_ok_fexpr_expr

Terse proof for `expr_eval_if_ok_fexpr_expr`.

`expr_eval_if_ok_fexpr_expr`:

$\{1\} \quad \forall (\text{expr}: [\text{State} \rightarrow \text{ExprResult}[\text{State}, \text{Data1}]], \text{expr1}: [\text{State} \rightarrow \text{ExprResult}[\text{State}, \text{Data2}]],$ $\quad \text{expr2}: [\text{State} \rightarrow \text{ExprResult}[\text{State}, \text{Data4}]],$ $\quad \text{fexpr}: [\text{Data2} \rightarrow [\text{State} \rightarrow \text{ExprResult}[\text{State}, \text{Data3}]]], s: \text{State}):$ $\text{OK?}((\text{expr} \ \#\# \ \text{expr1})(s)) \supset$ $(\text{expr} \ \#\# \ (\text{expr1} \ \#\# \ \text{fexpr}) \ \#\# \ \text{expr2})(s) =$ $(\text{expr} \ \#\# \ \text{expr1} \ \#\# \ \text{fexpr}(\text{data}((\text{expr} \ \#\# \ \text{expr1})(s)))) \ \#\# \ \text{expr2})(s)$

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `expr_eval_if_ok_fexpr_expr`.
Q.E.D.

C.208 Proofs for SkipStatements (statements.pvs)

This theory contains no provable formal statements.

C.209 Proofs for State_Transformer (state-transformer.pvs)

This theory contains no provable formal statements.

C.210 Proofs for State_Transformer_Lift (state-transformer.pvs)

This theory contains no provable formal statements.

C.211 Proofs for State_Transformer_Lift_Expr (state-transformer.pvs)

C.211.1 State_Transformer_Lift_Expr.ok_result_ok

Terse proof for ok_result_ok.

ok_result_ok:

$$\frac{}{\{1\} \quad \forall (s: \text{State}, \text{data}: \text{Data}): \text{OK?}(\text{ok_result}(\text{data})(s))}$$

,
This completes the proof of ok_result_ok.
Q.E.D.

C.211.2 State_Transformer_Lift_Expr.ok_result_data_TCC1

Terse proof for ok_result_data_TCC1.

ok_result_data_TCC1:

$$\frac{}{\{1\} \quad \forall (s: \text{State}, \text{data}: \text{Data}): \text{OK?}[\text{State}, \text{Data}](\text{ok_result}(\text{data})(s))}$$

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of ok_result_data_TCC1.
Q.E.D.

C.211.3 State_Transformer_Lift_Expr.ok_result_data

Terse proof for ok_result_data.

ok_result_data:

$$\frac{}{\{1\} \quad \forall (s: \text{State}, \text{data}: \text{Data}): \text{data}(\text{ok_result}(\text{data})(s)) = \text{data}}$$

,
This completes the proof of ok_result_data.
Q.E.D.

C.211.4 State_Transformer_Lift_Expr.ok_result_state

Terse proof for ok_result_state.

ok_result_state:

$$\frac{}{\{1\} \quad \forall (s: \text{State}, \text{data}: \text{Data}): \text{state}(\text{ok_result}(\text{data})(s)) = s}$$

,
This completes the proof of ok_result_state.
Q.E.D.

C.211.5 State_Transformer_Lift_Expr.expr_2_super_ok_result

Terse proof for `expr_2_super_ok_result`.

`expr_2_super_ok_result`:

$$\frac{}{\{1\} \quad \forall (s: \text{State}, \text{data}: \text{Data}): \text{OK}?(\text{expr_2_super}(\text{ok_result}(\text{data}))(s))}$$

This completes the proof of `expr_2_super_ok_result`.

Q.E.D.

C.211.6 State_Transformer_Lift_Expr.ok_lift_ok

Terse proof for `ok_lift_ok`.

`ok_lift_ok`:

$$\frac{}{\{1\} \quad \forall (s: \text{State}, d: \text{lift}[\text{Data}]): \text{up}?(d) \supset \text{ok_lift}(d)(s) = \text{OK}(s, \text{down}(d))}$$

This completes the proof of `ok_lift_ok`.

Q.E.D.

C.211.7 State_Transformer_Lift_Expr.has_next_state_ok_result

Terse proof for `has_next_state_ok_result`.

`has_next_state_ok_result`:

$$\frac{}{\{1\} \quad \forall (s: \text{State}, \text{data}: \text{Data}): \text{has_next_state}(\text{ok_result}(\text{data}))(s)}$$

This completes the proof of `has_next_state_ok_result`.

Q.E.D.

C.211.8 State_Transformer_Lift_Expr.has_next_state_fatal_result

Terse proof for `has_next_state_fatal_result`.

`has_next_state_fatal_result`:

$$\frac{}{\{1\} \quad \forall (s: \text{State}): \neg \text{has_next_state}(\text{fatal_result})(s)}$$

This completes the proof of `has_next_state_fatal_result`.

Q.E.D.

C.211.9 State_Transformer_Lift_Expr.expr_2_super_fatal_result

Terse proof for `expr_2_super_fatal_result`.

`expr_2_super_fatal_result`:

$$\frac{}{\{1\} \quad \forall (s: \text{State}): \text{expr_2_super}(\text{fatal_result})(s) = \text{Bottom}}$$

This completes the proof of `expr_2_super_fatal_result`.

Q.E.D.

C.212 Proofs for State_Transformer_Lift_Stmt (state-transformer.pvs)

This theory contains no provable formal statements.

C.213 Proofs for Statement_Composition_Rewrites (state-transformer.pvs)

C.213.1

Statement_Composition_Rewrites.composition_assoc_rewrite_1

Terse proof for composition_assoc_rewrite_1.

composition_assoc_rewrite_1:

$$\frac{\{1\} \quad \forall (\text{stmt}_1, \text{stmt}_2, \text{stmt}_3: [\text{State} \rightarrow \text{StmtResult}[\text{State}]])}{(\text{stmt}_1 \## (\text{stmt}_2 \## \text{stmt}_3)) = ((\text{stmt}_1 \## \text{stmt}_2) \## \text{stmt}_3)}$$

Repeatedly Skolemizing and flattening,
Applying decompose-equality,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of composition_assoc_rewrite_1.
Q.E.D.

C.213.2

Statement_Composition_Rewrites.composition_assoc_rewrite_2

Terse proof for composition_assoc_rewrite_2.

composition_assoc_rewrite_2:

$$\frac{\{1\} \quad \forall (\text{lift_stmt}_3: [\text{StmtResult}[\text{State}] \rightarrow \text{StmtResult}[\text{State}]], \text{stmt}_1, \text{stmt}_2: [\text{State} \rightarrow \text{StmtResult}[\text{State}]])}{(\text{stmt}_1 \## (\text{lift}(\text{stmt}_2) \## \text{lift_stmt}_3)) = ((\text{stmt}_1 \## \text{stmt}_2) \## \text{lift_stmt}_3)}$$

Repeatedly Skolemizing and flattening,
Applying decompose-equality,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of composition_assoc_rewrite_2.
Q.E.D.

C.213.3

Statement_Composition_Rewrites.composition_assoc_rewrite_3

Terse proof for composition_assoc_rewrite_3.

composition_assoc_rewrite_3:

$$\frac{\{1\} \quad \forall (\text{lift_stmt}_2: [\text{StmtResult}[\text{State}] \rightarrow \text{StmtResult}[\text{State}]], \text{stmt}_1, \text{stmt}_3: [\text{State} \rightarrow \text{StmtResult}[\text{State}]])}{(\text{stmt}_1 \## (\text{lift_stmt}_2 \## \text{stmt}_3)) = ((\text{stmt}_1 \## \text{lift_stmt}_2) \## \text{stmt}_3)}$$

Repeatedly Skolemizing and flattening,
 Applying decompose-equality,
 Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of `composition_assoc_rewrite_3`.
 Q.E.D.

C.213.4

Statement_Composition_Rewrites.composition_assoc_rewrite_4

Terse proof for `composition_assoc_rewrite_4`.

`composition_assoc_rewrite_4`:

$$\frac{\{1\} \quad \forall (\text{lift_stmt_2}, \text{lift_stmt_3}: [\text{StmtResult}[\text{State}] \rightarrow \text{StmtResult}[\text{State}]], \text{stmt_1}: [\text{State} \rightarrow \text{StmtResult}[\text{State}]])}{(\text{stmt_1} \## (\text{lift_stmt_2} \## \text{lift_stmt_3})) = ((\text{stmt_1} \## \text{lift_stmt_2}) \## \text{lift_stmt_3})}$$

Repeatedly Skolemizing and flattening,
 Applying decompose-equality,
 Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of `composition_assoc_rewrite_4`.
 Q.E.D.

C.213.5

Statement_Composition_Rewrites.composition_assoc_rewrite_5

Terse proof for `composition_assoc_rewrite_5`.

`composition_assoc_rewrite_5`:

$$\frac{\{1\} \quad \forall (\text{lift_stmt_1}: [\text{StmtResult}[\text{State}] \rightarrow \text{StmtResult}[\text{State}]], \text{stmt_2}, \text{stmt_3}: [\text{State} \rightarrow \text{StmtResult}[\text{State}]])}{(\text{lift_stmt_1} \## (\text{stmt_2} \## \text{stmt_3})) = ((\text{lift_stmt_1} \## \text{stmt_2}) \## \text{stmt_3})}$$

Repeatedly Skolemizing and flattening,
 Applying decompose-equality,
 Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of `composition_assoc_rewrite_5`.
 Q.E.D.

C.213.6

Statement_Composition_Rewrites.composition_assoc_rewrite_6

Terse proof for `composition_assoc_rewrite_6`.

`composition_assoc_rewrite_6`:

$$\frac{\{1\} \quad \forall (\text{lift_stmt_1}, \text{lift_stmt_3}: [\text{StmtResult}[\text{State}] \rightarrow \text{StmtResult}[\text{State}]], \text{stmt_2}: [\text{State} \rightarrow \text{StmtResult}[\text{State}]])}{(\text{equalities}[[\text{StmtResult}[\text{State}] \rightarrow \text{StmtResult}[\text{State}]]]. = ((\text{lift_stmt_1} \## (\text{lift}(\text{stmt_2}) \## \text{lift_stmt_3})), ((\text{lift_stmt_1} \## \text{stmt_2}) \## \text{lift_stmt_3}))}$$

Repeatedly Skolemizing and flattening,
 Applying decompose-equality,
 Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of `composition_assoc_rewrite_6`.
 Q.E.D.

C.213.7

Statement_Composition_Rewrites.composition_assoc_rewrite_7

Terse proof for `composition_assoc_rewrite_7`.

`composition_assoc_rewrite_7`:

$$\frac{\{1\} \quad \forall (\text{lift_stmt}_1, \text{lift_stmt}_2: [\text{StmtResult}[\text{State}] \rightarrow \text{StmtResult}[\text{State}]], \text{stmt}_3: [\text{State} \rightarrow \text{StmtResult}[\text{State}]])}{(\text{lift_stmt}_1 \## (\text{lift_stmt}_2 \## \text{stmt}_3)) = ((\text{lift_stmt}_1 \## \text{lift_stmt}_2) \## \text{stmt}_3)}$$

Repeatedly Skolemizing and flattening,
 Applying decompose-equality,
 Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of `composition_assoc_rewrite_7`.
 Q.E.D.

C.213.8

Statement_Composition_Rewrites.composition_assoc_rewrite_8

Terse proof for `composition_assoc_rewrite_8`.

`composition_assoc_rewrite_8`:

$$\frac{\{1\} \quad \forall (\text{lift_stmt}_1, \text{lift_stmt}_2, \text{lift_stmt}_3: [\text{StmtResult}[\text{State}] \rightarrow \text{StmtResult}[\text{State}]])}{(\text{equalities}[[\text{StmtResult}[\text{State}] \rightarrow \text{StmtResult}[\text{State}]]]. = ((\text{lift_stmt}_1 \## (\text{lift_stmt}_2 \## \text{lift_stmt}_3)), ((\text{lift_stmt}_1 \## \text{lift_stmt}_2) \## \text{lift_stmt}_3)))}$$

Repeatedly Skolemizing and flattening,
 Applying decompose-equality,
 Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of `composition_assoc_rewrite_8`.
 Q.E.D.

C.214 Proofs for Statement_Rewrites (statement-rewrites.pvs)

This theory contains no provable formal statements.

C.215 Proofs for Statements (statements.pvs)

This theory contains no provable formal statements.

C.216 Proofs for StmtResult (result.pvs)

This theory contains no provable formal statements.

C.217 Proofs for StmtResult_adt (StmtResult_adt.pvs)

This theory contains no provable formal statements.

C.218 Proofs for StmtResult_adt_map (StmtResult_adt.pvs)

This theory contains no provable formal statements.

C.219 Proofs for StmtResult_adt_reduce (StmtResult_adt.pvs)

This theory contains no provable formal statements.

C.220 Proofs for Subtype_stable (graph.pvs)

C.220.1 Subtype_stable.length_stable

Terse proof for length_stable.

length_stable:

$$\{1\} \quad \forall (P: \text{PRED}[T], l: \text{list}[T]):$$
$$\text{every}(P)(l) \supset \text{length}[(P)](l) = \text{length}[T](l)$$

By induction on l , and by repeatedly rewriting and simplifying,
This completes the proof of length_stable.
Q.E.D.

C.220.2 Subtype_stable.nth_stable_TCC1

Terse proof for nth_stable_TCC1.

nth_stable_TCC1:

$$\{1\} \quad \forall (P: \text{PRED}[T], l: \text{list}[T], n: \text{below}(\text{length}[T](l))):$$
$$\text{every}(P)(l) \supset n < \text{length}[(P)](l)$$

Repeatedly Skolemizing and flattening,
Rewriting using length_stable, matching in *,
This completes the proof of nth_stable_TCC1.
Q.E.D.

C.220.3 Subtype_stable.nth_stable

Terse proof for nth_stable.

nth_stable:

$$\frac{}{\{1\} \quad \forall (P: \text{PRED}[T], l: \text{list}[T], n: \text{below}(\text{length}(l))): \\ \text{every}(P)(l) \supset \text{nth}[(P)](l, n) = \text{nth}[T](l, n)}$$

Installing automatic rewrites from: length_stable

Inducting on l on formula 1,

we get 3 subgoals:

nth_stable.1:

$$\frac{}{\{1\} \quad \forall (P: \text{PRED}[T], n: \text{below}(\text{length}(\text{null}))) : \\ \text{every}(P)(\text{null}) \supset \text{nth}[(P)](\text{null}, n) = \text{nth}[T](\text{null}, n)}$$

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of nth_stable.1.

nth_stable.2:

$$\frac{}{\{1\} \quad \forall (\text{cons1_var}: T, \text{cons2_var}: \text{list}[T]): \\ (\forall (P: \text{PRED}[T], n: \text{below}(\text{length}(\text{cons2_var}))) : \\ \text{every}(P)(\text{cons2_var}) \supset \text{nth}[(P)](\text{cons2_var}, n) = \text{nth}[T](\text{cons2_var}, n)) \\ \supset \\ (\forall (P: \text{PRED}[T], n: \text{below}(\text{length}(\text{cons}(\text{cons1_var}, \text{cons2_var})))) : \\ \text{every}(P)(\text{cons}(\text{cons1_var}, \text{cons2_var})) \supset \\ \text{nth}[(P)](\text{cons}(\text{cons1_var}, \text{cons2_var}), n) = \\ \text{nth}[T](\text{cons}(\text{cons1_var}, \text{cons2_var}), n))}$$

Repeatedly Skolemizing and flattening,

Expanding the definition of nth,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Expanding the definition of every,

Applying disjunctive simplification to flatten sequent,

Instantiating quantified variables,

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of nth_stable.2.

nth_stable.3:

$$\frac{}{\{1\} \quad \forall (l: \text{list}[T], P: \text{PRED}[T], n: \text{below}(\text{length}[T](l))): \\ \text{every}(P)(l) \supset n < \text{length}[(P)](l) \\ \{2\} \quad \forall (P: \text{PRED}[T], n: \text{below}(\text{length}(l'))): \\ \text{every}(P)(l') \supset \text{nth}[(P)](l', n) = \text{nth}[T](l', n)}$$

Hiding formulas: 2,

Repeatedly Skolemizing and flattening,

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of nth_stable.3.

Q.E.D.

C.221 Proofs for SuperResult (result.pvs)

This theory contains no provable formal statements.

C.222 Proofs for SuperResult_adt (SuperResult_adt.pvs)

This theory contains no provable formal statements.

C.223 Proofs for SuperResult_adt_map (SuperResult_adt.pvs)

This theory contains no provable formal statements.

C.224 Proofs for SuperResult_adt_reduce (SuperResult_adt.pvs)

This theory contains no provable formal statements.

C.225 Proofs for Super_Embedding (result.pvs)

This theory contains no provable formal statements.

C.226 Proofs for Super_Embedding_Expr (result.pvs)

C.226.1 Super_Embedding_Expr.expr_2_super_res_TCC1

Terse proof for `expr_2_super_res_TCC1`.

`expr_2_super_res_TCC1`:

$$\{1\} \quad \forall (\text{expr}: \text{ExprResult}[\text{State}, \text{Data}]):$$
$$\text{OK}?(\text{expr}) \supset \text{OK}?[\text{State}, \text{Data}](\text{expr}) \vee \text{Exception}?[\text{State}, \text{Data}](\text{expr})$$

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `expr_2_super_res_TCC1`.
Q.E.D.

C.226.2 Super_Embedding_Expr.expr_2_super_res_TCC2

Terse proof for `expr_2_super_res_TCC2`.

`expr_2_super_res_TCC2`:

$$\{1\} \quad \forall (\text{expr}: \text{ExprResult}[\text{State}, \text{Data}]):$$
$$\text{has_next_state}(\text{expr}) \wedge \neg \text{OK}?(\text{expr}) \supset$$
$$\text{OK}?[\text{State}, \text{Data}](\text{expr}) \vee \text{Exception}?[\text{State}, \text{Data}](\text{expr})$$

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `expr_2_super_res_TCC2`.

Q.E.D.

C.227 Proofs for Super_Embedding_Stmt (result.pvs)

C.227.1 Super_Embedding_Stmt.stmt_2_super_res_TCC1

Terse proof for `stmt_2_super_res_TCC1`.

`stmt_2_super_res_TCC1`:

$\{1\} \quad \forall (\text{stmt} : \text{StmtResult}[\text{State}]):$ $\text{OK?}(\text{stmt}) \supset$ $\text{OK?}[\text{State}](\text{stmt}) \vee$ $\text{Break?}[\text{State}](\text{stmt}) \vee$ $\text{Continue?}[\text{State}](\text{stmt}) \vee$ $\text{Return?}[\text{State}](\text{stmt}) \vee$ $\text{Switch?}[\text{State}](\text{stmt}) \vee \text{Default?}[\text{State}](\text{stmt}) \vee \text{Exception?}[\text{State}](\text{stmt})$

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `stmt_2_super_res_TCC1`.

Q.E.D.

C.227.2 Super_Embedding_Stmt.stmt_2_super_res_TCC2

Terse proof for `stmt_2_super_res_TCC2`.

`stmt_2_super_res_TCC2`:

$\{1\} \quad \forall (\text{stmt} : \text{StmtResult}[\text{State}]):$ $\text{has_next_state}(\text{stmt}) \wedge \neg \text{OK?}(\text{stmt}) \supset$ $\text{OK?}[\text{State}](\text{stmt}) \vee$ $\text{Break?}[\text{State}](\text{stmt}) \vee$ $\text{Continue?}[\text{State}](\text{stmt}) \vee$ $\text{Return?}[\text{State}](\text{stmt}) \vee$ $\text{Switch?}[\text{State}](\text{stmt}) \vee \text{Default?}[\text{State}](\text{stmt}) \vee \text{Exception?}[\text{State}](\text{stmt})$
--

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `stmt_2_super_res_TCC2`.

Q.E.D.

C.228 Proofs for Super_Result_Util (result.pvs)

This theory contains no provable formal statements.

C.229 Proofs for Transformer_Invariant (state-transformer.pvs)

C.229.1 Transformer_Invariant.result_pred_TCC1

Terse proof for result_pred_TCC1.

result_pred_TCC1:

$$\{1\} \quad \forall (\text{res}: \text{SuperResult}[\text{State}]):$$

$$\text{has_next_state}(\text{res}) \supset \text{OK?}[\text{State}](\text{res}) \vee \text{abnormal?}[\text{State}](\text{res})$$

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of result_pred_TCC1.

Q.E.D.

C.229.2

Transformer_Invariant.super_transformer_invariant_next_ok_TCC1

Terse proof for super_transformer_invariant_next_ok_TCC1.

super_transformer_invariant_next_ok_TCC1:

$$\{1\} \quad \forall (\text{states}: \text{PRED}[\text{State}], \text{transformers}: \text{PRED}[[\text{State} \rightarrow \text{SuperResult}[\text{State}]]], s: \text{State},$$

$$q: [\text{State} \rightarrow \text{SuperResult}[\text{State}]]):$$
$$\text{transformer_invariant?}(\text{states}, \text{transformers}) \wedge$$
$$\text{states}(s) \wedge \text{transformers}(q) \wedge \text{has_next_state}(q(s))$$
$$\supset \text{OK?}[\text{State}](q(s)) \vee \text{abnormal?}[\text{State}](q(s))$$

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of super_transformer_invariant_next_ok_TCC1.

Q.E.D.

C.229.3 Transformer_Invariant.super_transformer_invariant_next_ok

Terse proof for super_transformer_invariant_next_ok.

super_transformer_invariant_next_ok:

$$\{1\} \quad \forall (\text{states}: \text{PRED}[\text{State}], \text{transformers}: \text{PRED}[[\text{State} \rightarrow \text{SuperResult}[\text{State}]]], s: \text{State},$$

$$q: [\text{State} \rightarrow \text{SuperResult}[\text{State}]]):$$
$$\text{transformer_invariant?}(\text{states}, \text{transformers}) \wedge$$
$$\text{states}(s) \wedge \text{transformers}(q) \wedge \text{has_next_state}(q(s))$$
$$\supset \text{states}(\text{state}(q(s)))$$

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of super_transformer_invariant_next_ok.

Q.E.D.

C.229.4**Transformer_Invariant.transformer_invariant_mono_transformers**

Terse proof for transformer_invariant_mono_transformers.

transformer_invariant_mono_transformers:

$\{1\} \quad \forall (\text{states}: \text{PRED}[\text{State}],$ $\quad \text{transformers}_1, \text{transformers}_2: \text{PRED}[[\text{State} \rightarrow \text{SuperResult}[\text{State}]]]):$ $(\text{transformers}_1 \subseteq \text{transformers}_2) \wedge \text{transformer_invariant?}(\text{states}, \text{transformers}_2) \supset$ $\text{transformer_invariant?}(\text{states}, \text{transformers}_1)$
--

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of transformer_invariant_mono_transformers.

Q.E.D.

C.229.5**Transformer_Invariant.transformer_invariant_union_transformers**

Terse proof for transformer_invariant_union_transformers.

transformer_invariant_union_transformers:

$\{1\} \quad \forall (\text{states}: \text{PRED}[\text{State}],$ $\quad \text{transformers}_1, \text{transformers}_2: \text{PRED}[[\text{State} \rightarrow \text{SuperResult}[\text{State}]]]):$ $\text{transformer_invariant?}(\text{states}, \text{transformers}_1) \wedge$ $\text{transformer_invariant?}(\text{states}, \text{transformers}_2)$ $\supset \text{transformer_invariant?}(\text{states}, (\text{transformers}_1 \cup \text{transformers}_2))$
--

Installing automatic rewrites from: member

Repeatedly Skolemizing and flattening,

Expanding the definition of transformer_invariant?,

Repeatedly Skolemizing and flattening,

Expanding the definition of union,

Splitting conjunctions,

we get 2 subgoals:

transformer_invariant_union_transformers.1:

$\{-1\} \quad (q' \in \text{transformers}'_1)$ $\{-2\} \quad \forall (s: \text{State}, q: [\text{State} \rightarrow \text{SuperResult}[\text{State}]]):$ $\quad \text{states}'(s) \wedge \text{transformers}'(q) \supset \text{result_pred}(\text{states}'(q(s)))$ $\{-3\} \quad \forall (s: \text{State}, q: [\text{State} \rightarrow \text{SuperResult}[\text{State}]]):$ $\quad \text{states}'(s) \wedge \text{transformers}''(q) \supset \text{result_pred}(\text{states}'(q(s)))$ $\{-4\} \quad \text{states}'(s')$ <hr/> $\{1\} \quad \text{result_pred}(\text{states}'(q'(s')))$

Instantiating quantified variables,

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of transformer_invariant_union_transformers.1.

`transformer_invariant_union_transformers.2:`

<div style="display: flex; flex-direction: column; gap: 5px;"> <div style="border-bottom: 1px solid black; padding: 2px 5px;">{-1} $(q' \in \text{transformers}'')$</div> <div style="padding: 2px 5px;">{-2} $\forall (s: \text{State}, q: [\text{State} \rightarrow \text{SuperResult}[\text{State}]]):$ $\text{states}'(s) \wedge \text{transformers}'(q) \supset \text{result_pred}(\text{states}')(q(s))$</div> <div style="padding: 2px 5px;">{-3} $\forall (s: \text{State}, q: [\text{State} \rightarrow \text{SuperResult}[\text{State}]]):$ $\text{states}'(s) \wedge \text{transformers}''(q) \supset \text{result_pred}(\text{states}')(q(s))$</div> <div style="padding: 2px 5px;">{-4} $\text{states}'(s')$</div> </div>	<div style="border-top: 1px solid black; border-left: 1px solid black; border-right: 1px solid black; padding: 5px;"> <div style="border-bottom: 1px solid black; padding: 2px 5px;">{1} $\text{result_pred}(\text{states}')(q'(s'))$</div> </div>
--	--

Instantiating quantified variables,
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `transformer_invariant_union_transformers.2`.
Q.E.D.

C.229.6 Transformer_Invariant.transformer_invariant_truth

Terse proof for `transformer_invariant_truth`.

`transformer_invariant_truth:`

<div style="border-bottom: 1px solid black; padding: 2px 5px;">{1} $\forall (\text{transformers}: \text{PRED}[[\text{State} \rightarrow \text{SuperResult}[\text{State}]]):$ $\text{transformer_invariant}?(fullset[\text{State}], \text{transformers})$</div>	<div style="border-top: 1px solid black; border-left: 1px solid black; border-right: 1px solid black; padding: 5px;"> <div style="border-bottom: 1px solid black; padding: 2px 5px;">{1} $\forall (\text{transformers}: \text{PRED}[[\text{State} \rightarrow \text{SuperResult}[\text{State}]]):$ $\text{transformer_invariant}?(fullset[\text{State}], \text{transformers})$</div> </div>
--	---

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `transformer_invariant_truth`.
Q.E.D.

C.229.7

Transformer_Invariant.transformer_invariant_all_transformers

Terse proof for `transformer_invariant_all_transformers`.

`transformer_invariant_all_transformers:`

<div style="border-bottom: 1px solid black; padding: 2px 5px;">{1} $\forall (\text{states}: \text{PRED}[\text{State}], \text{transformers}: \text{PRED}[[\text{State} \rightarrow \text{SuperResult}[\text{State}]]):$ $(\forall (q: [\text{State} \rightarrow \text{SuperResult}[\text{State}]]):$ $\text{transformers}(q) \supset \text{transformer_invariant}?(states, singleton(q))$ $\supset \text{transformer_invariant}?(states, \text{transformers})$</div>	<div style="border-top: 1px solid black; border-left: 1px solid black; border-right: 1px solid black; padding: 5px;"> <div style="border-bottom: 1px solid black; padding: 2px 5px;">{1} $\forall (\text{states}: \text{PRED}[\text{State}], \text{transformers}: \text{PRED}[[\text{State} \rightarrow \text{SuperResult}[\text{State}]]):$ $(\forall (q: [\text{State} \rightarrow \text{SuperResult}[\text{State}]]):$ $\text{transformers}(q) \supset \text{transformer_invariant}?(states, singleton(q))$ $\supset \text{transformer_invariant}?(states, \text{transformers})$</div> </div>
--	---

Expanding the definition of `transformer_invariant?`,
Repeatedly Skolemizing and flattening,
Instantiating quantified variables,
Simplifying, rewriting, and recording with decision procedures,
Instantiating quantified variables,
Expanding the definition of `singleton`,
which is trivially true.
This completes the proof of `transformer_invariant_all_transformers`.
Q.E.D.

C.229.8 Transformer_Invariant.super_transformers_ok_ok

Terse proof for `super_transformers_ok_ok`.

super_transformers_ok_ok:

$$\{1\} \quad \forall (\text{states: PRED}[\text{State}], \text{transformers: PRED}[[\text{State} \rightarrow \text{SuperResult}[\text{State}]]], s: \text{State}, \\ q: [\text{State} \rightarrow \text{SuperResult}[\text{State}]]): \\ \text{transformers_ok?}(\text{states}, \text{transformers}) \wedge \text{states}(s) \wedge \text{transformers}(q) \supset \text{OK?}(q(s))$$

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of super_transformers_ok_ok.
Q.E.D.

C.229.9 Transformer_Invariant.transformers_ok_mono_transformers

Terse proof for transformers_ok_mono_transformers.

transformers_ok_mono_transformers:

$$\{1\} \quad \forall (\text{states: PRED}[\text{State}], \\ \text{transformers}_1, \text{transformers}_2: \text{PRED}[[\text{State} \rightarrow \text{SuperResult}[\text{State}]]]): \\ (\text{transformers}_1 \subseteq \text{transformers}_2) \wedge \text{transformers_ok?}(\text{states}, \text{transformers}_2) \supset \\ \text{transformers_ok?}(\text{states}, \text{transformers}_1)$$

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of transformers_ok_mono_transformers.
Q.E.D.

C.229.10 Transformer_Invariant.transformers_ok_mono_states

Terse proof for transformers_ok_mono_states.

transformers_ok_mono_states:

$$\{1\} \quad \forall (\text{transformers: PRED}[[\text{State} \rightarrow \text{SuperResult}[\text{State}]]], \text{states}_1, \text{states}_2: \text{PRED}[\text{State}]): \\ (\text{states}_1 \subseteq \text{states}_2) \wedge \text{transformers_ok?}(\text{states}_2, \text{transformers}) \supset \\ \text{transformers_ok?}(\text{states}_1, \text{transformers})$$

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of transformers_ok_mono_states.
Q.E.D.

C.229.11 Transformer_Invariant.transformers_ok_union_transformers

Terse proof for transformers_ok_union_transformers.

transformers_ok_union_transformers:

$$\{1\} \quad \forall (\text{states: PRED}[\text{State}], \\ \text{transformers}_1, \text{transformers}_2: \text{PRED}[[\text{State} \rightarrow \text{SuperResult}[\text{State}]]]): \\ \text{transformers_ok?}(\text{states}, \text{transformers}_1) \wedge \text{transformers_ok?}(\text{states}, \text{transformers}_2) \supset \\ \text{transformers_ok?}(\text{states}, (\text{transformers}_1 \cup \text{transformers}_2))$$

Installing automatic rewrites from: member
Repeatedly Skolemizing and flattening,
Expanding the definition of transformers_ok?,

C Proof scripts

Repeatedly Skolemizing and flattening,
 Expanding the definition of union,
 Splitting conjunctions,
 we get 2 subgoals:

`transformers_ok_union_transformers.1:`

{-1}	$(q' \in \text{transformers}')$
{-2}	$\forall (s: \text{State}, q: [\text{State} \rightarrow \text{SuperResult}[\text{State}]]):$ $\text{states}'(s) \wedge \text{transformers}'(q) \supset \text{OK?}(q(s))$
{-3}	$\forall (s: \text{State}, q: [\text{State} \rightarrow \text{SuperResult}[\text{State}]]):$ $\text{states}'(s) \wedge \text{transformers}''(q) \supset \text{OK?}(q(s))$
{-4}	$\text{states}'(s')$
{1} $\text{OK?}(q'(s'))$	

Instantiating quantified variables,
 Simplifying, rewriting, and recording with decision procedures,
 This completes the proof of `transformers_ok_union_transformers.1`.
`transformers_ok_union_transformers.2:`

{-1}	$(q' \in \text{transformers}'')$
{-2}	$\forall (s: \text{State}, q: [\text{State} \rightarrow \text{SuperResult}[\text{State}]]):$ $\text{states}'(s) \wedge \text{transformers}'(q) \supset \text{OK?}(q(s))$
{-3}	$\forall (s: \text{State}, q: [\text{State} \rightarrow \text{SuperResult}[\text{State}]]):$ $\text{states}'(s) \wedge \text{transformers}''(q) \supset \text{OK?}(q(s))$
{-4}	$\text{states}'(s')$
{1} $\text{OK?}(q'(s'))$	

Instantiating quantified variables,
 Simplifying, rewriting, and recording with decision procedures,
 This completes the proof of `transformers_ok_union_transformers.2`.
 Q.E.D.

C.229.12 Transformer_Invariant.transformers_ok_all_transformers

Terse proof for `transformers_ok_all_transformers`.

`transformers_ok_all_transformers:`

{1}	$\forall (\text{states}: \text{PRED}[\text{State}], \text{transformers}: \text{PRED}[[\text{State} \rightarrow \text{SuperResult}[\text{State}]]]):$ $(\forall (q: [\text{State} \rightarrow \text{SuperResult}[\text{State}]]):$ $\text{transformers}(q) \supset \text{transformers_ok?}(\text{states}, \text{singleton}(q))$ $\supset \text{transformers_ok?}(\text{states}, \text{transformers}))$
-----	--

Expanding the definition of `transformers_ok?`,
 Repeatedly Skolemizing and flattening,
 Instantiating quantified variables,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 Instantiating quantified variables,
 Expanding the definition of `singleton`,
 which is trivially true.
 This completes the proof of `transformers_ok_all_transformers`.
 Q.E.D.

C.230 Proofs for Transformer_Invariant_2 (state-transformer.pvs)

C.230.1

Transformer_Invariant_2.expr_transformer_invariant_next_ok_TCC1

Terse proof for `expr_transformer_invariant_next_ok_TCC1`.

`expr_transformer_invariant_next_ok_TCC1`:

$$\{1\} \quad \forall (\text{states: PRED}[\text{State}], \text{transformers: PRED}[[\text{State} \rightarrow \text{SuperResult}[\text{State}]]], s: \text{State}, \\ q: [\text{State} \rightarrow \text{ExprResult}[\text{State}, \text{Data}]]): \\ \text{transformer_invariant?}(\text{states}, \text{transformers}) \wedge \\ \text{states}(s) \wedge \text{transformers}(\text{expr_2_super}(q)) \wedge \text{has_next_state}(q(s)) \\ \supset \text{OK?}[\text{State}, \text{Data}](q(s)) \vee \text{Exception?}[\text{State}, \text{Data}](q(s))$$

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `expr_transformer_invariant_next_ok_TCC1`.
Q.E.D.

C.230.2 Transformer_Invariant_2.expr_transformer_invariant_next_ok

Terse proof for `expr_transformer_invariant_next_ok`.

`expr_transformer_invariant_next_ok`:

$$\{1\} \quad \forall (\text{states: PRED}[\text{State}], \text{transformers: PRED}[[\text{State} \rightarrow \text{SuperResult}[\text{State}]]], s: \text{State}, \\ q: [\text{State} \rightarrow \text{ExprResult}[\text{State}, \text{Data}]]): \\ \text{transformer_invariant?}(\text{states}, \text{transformers}) \wedge \\ \text{states}(s) \wedge \text{transformers}(\text{expr_2_super}(q)) \wedge \text{has_next_state}(q(s)) \\ \supset \text{states}(\text{state}(q(s)))$$

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `expr_transformer_invariant_next_ok`.
Q.E.D.

C.230.3 Transformer_Invariant_2.expr_transformers_ok_ok

Terse proof for `expr_transformers_ok_ok`.

`expr_transformers_ok_ok`:

$$\{1\} \quad \forall (\text{states: PRED}[\text{State}], \text{transformers: PRED}[[\text{State} \rightarrow \text{SuperResult}[\text{State}]]], s: \text{State}, \\ q: [\text{State} \rightarrow \text{ExprResult}[\text{State}, \text{Data}]]): \\ \text{transformers_ok?}(\text{states}, \text{transformers}) \wedge \text{states}(s) \wedge \text{transformers}(\text{expr_2_super}(q)) \supset \\ \text{OK?}(q(s))$$

Repeatedly Skolemizing and flattening,
Using lemma `super_transformers_ok_ok[State]`,

C Proof scripts

Simplifying, rewriting, and recording with decision procedures,
Rewriting using `ok_expr_2_super`, matching in *,
This completes the proof of `expr_transformers_ok_ok`.
Q.E.D.

C.230.4 Transformer_Invariant_2.expr_result_pred_next_state_TCC1

Terse proof for `expr_result_pred_next_state_TCC1`.

`expr_result_pred_next_state_TCC1`:

$$\{1\} \quad \forall (states: \text{PRED}[\text{State}], s: \text{State}, q: [\text{State} \rightarrow \text{ExprResult}[\text{State}, \text{Data}]]):$$

$$\text{has_next_state}(q(s)) \wedge \text{result_pred}(states)(\text{expr_2_super}(q)(s)) \supset$$
$$\text{OK}?[\text{State}, \text{Data}](q(s)) \vee \text{Exception}?[\text{State}, \text{Data}](q(s))$$

Repeatedly Skolemizing and flattening,
Expanding the definition of `has_next_state`,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `expr_result_pred_next_state_TCC1`.
Q.E.D.

C.230.5 Transformer_Invariant_2.expr_result_pred_next_state

Terse proof for `expr_result_pred_next_state`.

`expr_result_pred_next_state`:

$$\{1\} \quad \forall (states: \text{PRED}[\text{State}], s: \text{State}, q: [\text{State} \rightarrow \text{ExprResult}[\text{State}, \text{Data}]]):$$

$$\text{has_next_state}(q(s)) \wedge \text{result_pred}(states)(\text{expr_2_super}(q)(s)) \supset$$
$$\text{states}(\text{state}(q(s)))$$

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `expr_result_pred_next_state`.
Q.E.D.

C.231 Proofs for Transformer_Invariant_3 (state-transformer.pvs)

C.231.1 Transformer_Invariant_3.fexpr_composition_transformers_ok

Terse proof for `fexpr_composition_transformers_ok`.

`fexpr_composition_transformers_ok`:

$$\{1\} \quad \forall (\text{expr1}: [\text{State} \rightarrow \text{ExprResult}[\text{State}, \text{Data1}]],$$

$$\text{fexpr}: [\text{Data1} \rightarrow [\text{State} \rightarrow \text{ExprResult}[\text{State}, \text{Data2}]]], \text{states}: \text{PRED}[\text{State}],$$
$$P: \text{PRED}[\text{Data1}]):$$
$$(\text{transformers_ok}?(states, \text{singleton}(\text{expr_2_super}(\text{expr1}))) \wedge$$
$$\text{transformer_invariant}?(states, \text{singleton}(\text{expr_2_super}(\text{expr1}))) \wedge$$
$$(\forall (s: (states)): \text{OK}?(expr1(s)) \supset P(\text{data}(\text{expr1}(s)))) \wedge$$
$$(\forall (d: (P)):$$
$$\text{transformers_ok}?(states, \text{singleton}(\text{expr_2_super}(\text{fexpr}(d))))))$$
$$\supset \text{transformers_ok}?(states, \text{singleton}(\text{expr_2_super}(\text{expr1} \#\# \text{fexpr})))$$

Expanding the definition of transformers_ok?,
 Repeatedly Skolemizing and flattening,
 Installing automatic rewrites from: singleton ## expr_2_super expr_2_super_res
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 Replacing using formula -6,
 Hiding formulas: -6,
 Instantiating quantified variables,
 Instantiating quantified variables,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 Instantiating the top quantifier in -5 with the terms: (data(expr1!1(s!1))),
 Instantiating the top quantifier in -5 with the terms: (state(expr1!1(s!1)) expr_2_super(fexpr!1(data(expr1!1(s!1))))),
 we get 2 subgoals:
fexpr_composition_transformers_ok.1:

{-1}	states'(s')
{-2}	OK?(expr_2_super_res(expr1'(s')))
{-3}	transformer_invariant?(states', singleton(expr_2_super(expr1')))
{-4}	P'(data(expr1'(s')))
{-5}	states'(state(expr1'(s'))) \supset OK?(expr_2_super(fexpr'(data(expr1'(s'))))(state(expr1'(s'))))
{1} OK?(expr_2_super_res((expr1' ## fexpr')(s')))	

Expanding the definition of ##,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 Using lemma expr_transformer_invariant_next_ok,
 Installing automatic rewrites from: has_next_state
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 This completes the proof of fexpr_composition_transformers_ok.1.
fexpr_composition_transformers_ok.2:

{-1}	states'(s')
{-2}	OK?(expr_2_super_res(expr1'(s')))
{-3}	transformer_invariant?(states', singleton(expr_2_super(expr1')))
{-4}	P'(data(expr1'(s')))
{1} OK?[State, Data1](expr1'(s')) \vee Exception?[State, Data1](expr1'(s'))	
{2}	OK?(expr_2_super_res((expr1' ## fexpr')(s')))

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 This completes the proof of fexpr_composition_transformers_ok.2.
 Q.E.D.

C.231.2 Transformer_Invariant_3.expr_composition_transformers_ok

Terse proof for expr_composition_transformers_ok.

expr_composition_transformers_ok:

{1}	\forall (expr1: [State \rightarrow ExprResult[State, Data1]], expr2: [State \rightarrow ExprResult[State, Data2]], states: PRED[State]): transformers_ok?(states, singleton(expr_2_super(expr1))) \wedge transformer_invariant?(states, singleton(expr_2_super(expr1))) \wedge transformers_ok?(states, singleton(expr_2_super(expr2))) \supset transformers_ok?(states, singleton(expr_2_super(expr1 ## expr2)))
-----	--

Repeatedly Skolemizing and flattening,

C Proof scripts

Using lemma `fexpr_composition_transformers_ok`,
Expanding the definition of `##`,
Expanding the definition of `##`,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `fexpr_composition_transformers_ok`.
Q.E.D.

C.231.3

Transformer_Invariant_3.fexpr_composition_transformer_invariant

Terse proof for `fexpr_composition_transformer_invariant`.

`fexpr_composition_transformer_invariant`:

$\{1\} \quad \forall (\text{expr1}: [\text{State} \rightarrow \text{ExprResult}[\text{State}, \text{Data1}]],$ $\quad \text{fexpr}: [\text{Data1} \rightarrow [\text{State} \rightarrow \text{ExprResult}[\text{State}, \text{Data2}]]], \text{states}: \text{PRED}[\text{State}],$ $\quad P: \text{PRED}[\text{Data1}]):$ $\quad (\text{transformer_invariant}?(\text{states}, \text{singleton}(\text{expr_2_super}(\text{expr1}))) \wedge$ $\quad (\forall (s: (\text{states})): \text{OK}?(\text{expr1}(s)) \supset P(\text{data}(\text{expr1}(s)))) \wedge$ $\quad (\forall (d: (P)):$ $\quad \quad \text{transformer_invariant}?(\text{states}, \text{singleton}(\text{expr_2_super}(\text{fexpr}(d))))$ $\quad \supset \text{transformer_invariant}?(\text{states}, \text{singleton}(\text{expr_2_super}(\text{expr1} \text{ ## } \text{fexpr})))$

Expanding the definition of `transformer_invariant?`,
Repeatedly Skolemizing and flattening,
Expanding the definition of `singleton`,
Replacing using formula -5,
Hiding formulas: -5,
Instantiating quantified variables,
Instantiating quantified variables,
Case splitting on `OK?(expr1!(s!))`,
we get 2 subgoals:

`fexpr_composition_transformer_invariant.1`:

$\{-1\} \quad \text{OK}?(\text{expr1}'(s'))$ $\{-2\} \quad \text{states}'(s') \supset \text{result_pred}(\text{states}')(\text{expr_2_super}(\text{expr1}'(s')))$ $\{-3\} \quad \text{OK}?(\text{expr1}'(s')) \supset P'(\text{data}(\text{expr1}'(s')))$ $\{-4\} \quad \forall (d: (P')):$ $\quad \forall (s: \text{State}, q: [\text{State} \rightarrow \text{SuperResult}[\text{State}]]):$ $\quad \quad \text{states}'(s) \wedge q = \text{expr_2_super}(\text{fexpr}'(d)) \supset \text{result_pred}(\text{states}')(q(s))$ $\{-5\} \quad \text{states}'(s')$	$\{1\} \quad \text{result_pred}(\text{states}')(\text{expr_2_super}(\text{expr1}' \text{ ## } \text{fexpr}')(s'))$
---	---

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Instantiating the top quantifier in -5 with the terms: `(data(expr1!(s!))`),
Instantiating the top quantifier in -5 with the terms: `(state(expr1!(s!)) expr_2_super(fexpr!(data(expr1!(s!))))`),
Installing automatic rewrites from: `## result_pred expr_2_super expr_2_super_res has_next_state`
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `fexpr_composition_transformer_invariant.1`.

fexpr_composition_transformer_invariant.2:

<p>{-1} $states'(s') \supset result_pred(states')(expr_2_super(expr1')(s'))$ {-2} $OK?(expr1'(s')) \supset P'(data(expr1'(s')))$ {-3} $\forall (d: (P')):$ $\forall (s: State, q: [State \rightarrow SuperResult[State]]):$ $states'(s) \wedge q = expr_2_super(fexpr'(d)) \supset result_pred(states')(q(s))$ {-4} $states'(s')$</p>	<hr style="border: 0.5px solid black;"/> <p>{1} $OK?(expr1'(s'))$ {2} $result_pred(states')(expr_2_super(expr1' \#\# fexpr')(s'))$</p>
--	--

Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of fexpr_composition_transformer_invariant.2.
 Q.E.D.

C.231.4

Transformer_Invariant_3.expr_composition_transformer_invariant

Terse proof for expr_composition_transformer_invariant.

expr_composition_transformer_invariant:

<p>{1} $\forall (expr1: [State \rightarrow ExprResult[State, Data1]],$ $expr2: [State \rightarrow ExprResult[State, Data2]], states: PRED[State]):$ $transformer_invariant?(states, singleton(expr_2_super(expr1))) \wedge$ $transformer_invariant?(states, singleton(expr_2_super(expr2)))$ $\supset transformer_invariant?(states, singleton(expr_2_super(expr1 \#\# expr2)))$</p>	<hr style="border: 0.5px solid black;"/>
--	--

Repeatedly Skolemizing and flattening,
 Using lemma fexpr_composition_transformer_invariant,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 Expanding the definition of ##,
 Expanding the definition of ##,
 which is trivially true.
 This completes the proof of expr_composition_transformer_invariant.
 Q.E.D.

C.232 Proofs for Transformer_Super_Embedding (state-transformer.pvs)

This theory contains no provable formal statements.

C.233 Proofs for Transformer_Super_Embedding_Expr (state-transformer.pvs)

C.233.1 Transformer_Super_Embedding_Expr.ok_expr_2_super

Terse proof for ok_expr_2_super.

ok_expr_2_super:

$\{1\} \quad \forall (\text{expr}: [\text{State} \rightarrow \text{ExprResult}[\text{State}, \text{Data}]], s: \text{State}):$ $\text{OK}?(\text{expr_2_super}(\text{expr})(s)) = \text{OK}?(\text{expr}(s))$
--

This completes the proof of ok_expr_2_super.

Q.E.D.

C.233.2

Transformer_Super_Embedding_Expr.has_next_state_expr_2_super

Terse proof for has_next_state_expr_2_super.

has_next_state_expr_2_super:

$\{1\} \quad \forall (\text{expr}: [\text{State} \rightarrow \text{ExprResult}[\text{State}, \text{Data}]], s: \text{State}):$ $\text{has_next_state}(\text{expr_2_super}(\text{expr})(s)) = \text{has_next_state}(\text{expr}(s))$
--

This completes the proof of has_next_state_expr_2_super.

Q.E.D.

C.233.3

Transformer_Super_Embedding_Expr.state_expr_2_super_TCC1

Terse proof for state_expr_2_super_TCC1.

state_expr_2_super_TCC1:

$\{1\} \quad \forall (\text{expr}: [\text{State} \rightarrow \text{ExprResult}[\text{State}, \text{Data}]], s: \text{State}):$ $\text{has_next_state}(\text{expr}(s)) \supset$ $\text{OK}?[\text{State}](\text{expr_2_super}(\text{expr})(s)) \vee \text{abnormal}?[\text{State}](\text{expr_2_super}(\text{expr})(s))$

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of state_expr_2_super_TCC1.

Q.E.D.

C.233.4

Transformer_Super_Embedding_Expr.state_expr_2_super_TCC2

Terse proof for state_expr_2_super_TCC2.

state_expr_2_super_TCC2:

$\{1\} \quad \forall (\text{expr}: [\text{State} \rightarrow \text{ExprResult}[\text{State}, \text{Data}]], s: \text{State}):$ $\text{has_next_state}(\text{expr}(s)) \supset$ $\text{OK}?[\text{State}, \text{Data}](\text{expr}(s)) \vee \text{Exception}?[\text{State}, \text{Data}](\text{expr}(s))$
--

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of state_expr_2_super_TCC2.

Q.E.D.

C.233.5 Transformer_Super_Embedding_Expr.state_expr_2_super

Terse proof for state_expr_2_super.

state_expr_2_super:

$$\frac{\{1\} \quad \forall (\text{expr}: [\text{State} \rightarrow \text{ExprResult}[\text{State}, \text{Data}]], s: \text{State}):}{\text{has_next_state}(\text{expr}(s)) \supset \text{state}(\text{expr_2_super}(\text{expr})(s)) = \text{state}(\text{expr}(s))}$$

This completes the proof of state_expr_2_super.
Q.E.D.

C.234 Proofs for Transformer_Super_Embedding_Stmt (state-transformer.pvs)

C.234.1 Transformer_Super_Embedding_Stmt.ok_stmt_2_super

Terse proof for ok_stmt_2_super.

ok_stmt_2_super:

$$\frac{\{1\} \quad \forall (s: \text{State}, \text{stmt}: [\text{State} \rightarrow \text{StmtResult}[\text{State}]])}{\text{OK?}(\text{stmt_2_super}(\text{stmt})(s)) = \text{OK?}(\text{stmt}(s))}$$

This completes the proof of ok_stmt_2_super.
Q.E.D.

C.234.2

Transformer_Super_Embedding_Stmt.has_next_state_stmt_2_super

Terse proof for has_next_state_stmt_2_super.

has_next_state_stmt_2_super:

$$\frac{\{1\} \quad \forall (s: \text{State}, \text{stmt}: [\text{State} \rightarrow \text{StmtResult}[\text{State}]])}{\text{has_next_state}(\text{stmt_2_super}(\text{stmt})(s)) = \text{has_next_state}(\text{stmt}(s))}$$

This completes the proof of has_next_state_stmt_2_super.
Q.E.D.

C.234.3

Transformer_Super_Embedding_Stmt.state_stmt_2_super_TCC1

Terse proof for state_stmt_2_super_TCC1.

state_stmt_2_super_TCC1:

$$\frac{\{1\} \quad \forall (s: \text{State}, \text{stmt}: [\text{State} \rightarrow \text{StmtResult}[\text{State}]])}{\text{has_next_state}(\text{stmt}(s)) \supset \text{OK?}[\text{State}](\text{stmt_2_super}(\text{stmt})(s)) \vee \text{abnormal?}[\text{State}](\text{stmt_2_super}(\text{stmt})(s))}$$

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `state_stmt_2_super_TCC1`.
Q.E.D.

C.234.4

Transformer_Super_Embedding Stmt.state_stmt_2_super_TCC2

Terse proof for `state_stmt_2_super_TCC2`.

`state_stmt_2_super_TCC2`:

```
{1}  ∀ (s: State, stmt: [State → StmtResult [State]]):
      has_next_state(stmt(s)) ⊃
      OK?[State](stmt(s)) ∨
      Break?[State](stmt(s)) ∨
      Continue?[State](stmt(s)) ∨
      Return?[State](stmt(s)) ∨
      Switch?[State](stmt(s)) ∨
      Default?[State](stmt(s)) ∨ Exception?[State](stmt(s))
```

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `state_stmt_2_super_TCC2`.
Q.E.D.

C.234.5 Transformer_Super_Embedding Stmt.state_stmt_2_super

Terse proof for `state_stmt_2_super`.

`state_stmt_2_super`:

```
{1}  ∀ (s: State, stmt: [State → StmtResult [State]]):
      has_next_state(stmt(s)) ⊃ state(stmt_2_super(stmt)(s)) = state(stmt(s))
```

This completes the proof of `state_stmt_2_super`.
Q.E.D.

C.235 Proofs for UnaryExpressions (expressions.pvs)

C.235.1 UnaryExpressions.deref_TCC1

Terse proof for `deref_TCC1`.

`deref_TCC1`:

```
{1}  ∀ ((p_typ: Cpp_Subtype(cv(pointer?))): pointer?(cv_base(p_typ))
```

Repeatedly Skolemizing and flattening,
Rewriting using `cv_base_result`, matching in `*`,
Keeping (1) and hiding `*`,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `deref_TCC1`.
Q.E.D.

C.235.2 UnaryExpressions.arrow_TCC1

Terse proof for arrow_TCC1.

arrow_TCC1:

<pre>{1} ∀ (t: Cpp_Type): pointer?(t) ⊃ array?(t) ∨ pointer?(t) ∨ reference?(t) ∨ bitfield?(t) ∨ enum?(t) ∨ pointer_to_member?(t) ∨ const?(t) ∨ volatile?(t)</pre>

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of arrow_TCC1.

Q.E.D.

C.235.3 UnaryExpressions.arrow_TCC2

Terse proof for arrow_TCC2.

arrow_TCC2:

<pre>{1} ∀ ((p_typ: Cpp_Subtype(cv(extend[Cpp_Type_, Cpp_Type, bool, FALSE] (λ (t: Cpp_Type): pointer?(t) ∧ (class?(typ(t)) ∨ union?(typ(t)))))))): cv(pointer?)(p_typ)</pre>
--

Repeatedly Skolemizing and flattening,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of arrow_TCC2.

Q.E.D.

C.235.4 UnaryExpressions.unary_minus_TCC1

Terse proof for unary_minus_TCC1.

unary_minus_TCC1:

<pre>{1} ∀ (typ: Cpp_Subtype(cv(extend[Cpp_Type_, Cpp_Type, bool, FALSE] ({t: Cpp_Type (char?(t) ∨ wchar_t?(t) ∨ signed_integer?(t)})))): n: (range(typ)): cv(non_bool_integral_enum?)(typ)</pre>

Repeatedly Skolemizing and flattening,

Keeping (-2 1) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of unary_minus_TCC1.

Q.E.D.

C.235.5 UnaryExpressions.unary_minus_TCC2

Terse proof for unary_minus_TCC2.

unary_minus_TCC2:

$\{1\} \quad \forall (\text{typ}: \text{Cpp_Subtype}(\text{cv}(\text{extend}[\text{Cpp_Type}_, \text{Cpp_Type}, \text{bool}, \text{FALSE}] (\{t: \text{Cpp_Type} \mid (\text{char}? (t) \vee \text{wchar_t}? (t) \vee \text{signed_integer}? (t))\}))))): \text{cv}(\text{non_bool_integral_enum}?)(\text{typ})$
--

Repeatedly Skolemizing and flattening,
 Keeping (-2 1) and hiding *,
 Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of unary_minus_TCC2.
 Q.E.D.

C.235.6 UnaryExpressions.unary_minus_unsigned_TCC1

Terse proof for unary_minus_unsigned_TCC1.

unary_minus_unsigned_TCC1:

$\{1\} \quad \forall (\text{typ}: \text{Cpp_Subtype}(\text{cv}(\text{unsigned_integer}?)), n: (\text{range}(\text{typ}))) : \text{cv}(\text{non_bool_integral_enum}?)(\text{typ})$

Repeatedly Skolemizing and flattening,
 Keeping (-2 1) and hiding *,
 Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of unary_minus_unsigned_TCC1.
 Q.E.D.

C.235.7 UnaryExpressions.unary_minus_unsigned_TCC2

Terse proof for unary_minus_unsigned_TCC2.

unary_minus_unsigned_TCC2:

$\{1\} \quad \forall (\text{typ}: \text{Cpp_Subtype}(\text{cv}(\text{unsigned_integer}?))) : \text{cv}(\text{non_bool_integral_enum}?)(\text{typ})$
--

Repeatedly Skolemizing and flattening,
 Hiding formulas: -1,
 Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of unary_minus_unsigned_TCC2.
 Q.E.D.

C.235.8 UnaryExpressions.preinc_TCC1

Terse proof for preinc_TCC1.

preinc_TCC1:

$\{1\} \quad \forall ((\text{typ}: \text{Cpp_Subtype}(v(\text{non_bool_integral}?)))) : \text{cv}(\text{non_bool_integral_enum}?)(\text{typ})$
--

Repeatedly Skolemizing and flattening,
 Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of `preinc_TCC1`.
 Q.E.D.

C.235.9 UnaryExpressions.sizeof_TCC1

Terse proof for `sizeof_TCC1`.

`sizeof_TCC1`:

{1}	$\text{Cpp_Type?}(\text{uint}) \wedge \text{cv}(\text{non_bool_integral_enum?})(\text{uint})$
-----	---

Trying repeated skolemization, instantiation, and if-lifting,
 Expanding the definition of `Cpp_Type?`,
 Repeatedly Skolemizing and flattening,
 Expanding the definition of subterm,
 Replacing using formula -1,
 Installing automatic rewrites from: `no_pointers_to_bitfield?` `no_pointers_to_references?`
`no_cv_references?` `no_reference_to_reference?` `no_reference_to_bitfields?` `no_pointer_to_member_to_reference?`
`no_pointer_to_member_to_cv_void?` `no_cv_void_parameter?` `no_array_of_references?` `no_array_of_bitfields?`
`no_array_of_cv_void?` `no_array_of_function?` `no_array_of_abstract_class?` `no_pointer_or_ref_to_incomplete_array_parameter?`
`no_array_return_type?` `no_function_return_type?` `bitfield_underlying_integral_or_enum_type?` `cv_array?`
`enum_underlying_integral?` `enum_constants?` `const_volatile?` `const_stutter?` `volatile_stutter?`
`no_cv_class?` `no_cv_union?` `no_cv_function?` `no_reference_to_void?` `no_cv_void?`

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 This completes the proof of `sizeof_TCC1`.
 Q.E.D.

C.236 Proofs for Uninterpreted_Data (abstract_data.pvs)

C.236.1 Uninterpreted_Data.u_data_type_length

Terse proof for `u_data_type_length`.

`u_data_type_length`:

{1}	$\forall (\text{uidt}: (\text{uninterpreted_data_type?}), l: \text{list}[\text{Byte}], a: \text{Address}):$ $\text{valid?}(\text{uidt})(l, a) \supset \text{length}(l) = \text{size}(\text{uidt})$
-----	---

Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of `u_data_type_length`.
 Q.E.D.

C.237 Proofs for Unit (vfiasco-prelude.pvs)

This theory contains no provable formal statements.

C.238 Proofs for Unit_adt (Unit_adt.pvs)

This theory contains no provable formal statements.

C.239 Proofs for Unit_adt_reduce (Unit_adt.pvs)

This theory contains no provable formal statements.

C.240 Proofs for While_Hoare (statement-rewrites.pvs)

C.240.1 While_Hoare.iterate_while_right

Terse proof for `iterate_while_right`.

`iterate_while_right`:

$$\{1\} \quad \forall (b_ex: [State \rightarrow ExprResult[State, bool]], body: [State \rightarrow StmtResult[State]], n: nat):$$

$$\text{iterate_while}(1 + n, b_ex, body) =$$

$$\text{iterate_while}(n, b_ex, body) \## \text{lift}(e2s(b_ex)) \## \text{body} \## \text{catch_continue}$$

Installing automatic rewrites from: `erplus_ax erminus_ax extended_real_superset_of_number_fields`

Inducting on `n` on formula 1,

we get 2 subgoals:

`iterate_while_right.1`:

$$\{1\} \quad \forall (b_ex: [State \rightarrow ExprResult[State, bool]], body: [State \rightarrow StmtResult[State]]):$$

$$\text{iterate_while}(1 + 0, b_ex, body) =$$

$$\text{iterate_while}(0, b_ex, body) \## \text{lift}(e2s(b_ex)) \## \text{body} \## \text{catch_continue}$$

Repeatedly Skolemizing and flattening,

Applying `decompose-equality`,

Trying repeated skolemization, instantiation, and if-lifting,

Applying eta axiom scheme to `body!1(state(b_ex!1(x!1)))`,

This completes the proof of `iterate_while_right.1`.

`iterate_while_right.2`:

$$\{1\} \quad \forall j:$$

$$(\forall (b_ex: [State \rightarrow ExprResult[State, bool]], body: [State \rightarrow StmtResult[State]]):$$

$$\text{iterate_while}(1 + j, b_ex, body) =$$

$$\text{iterate_while}(j, b_ex, body) \## \text{lift}(e2s(b_ex)) \## \text{body} \## \text{catch_continue}))$$

$$\supset$$

$$(\forall (b_ex: [State \rightarrow ExprResult[State, bool]], body: [State \rightarrow StmtResult[State]]):$$

$$\text{iterate_while}(1 + j + 1, b_ex, body) =$$

$$\text{iterate_while}(j + 1, b_ex, body) \##$$

$$\text{lift}(e2s(b_ex)) \## \text{body} \## \text{catch_continue}))$$

Repeatedly Skolemizing and flattening,

Instantiating quantified variables,

Expanding the definition of `iterate_while`,

Expanding the definition of `##`,

Applying `decompose-equality`,

Expanding the definition of `lift`,

Replacing using formula -2,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `iterate_while_right.2`.

Q.E.D.

C.240.2 While_Hoare.while_variant?_TCC1

Terse proof for `while_variant?_TCC1`.

while_variant?_TCC1:

```

{1}  ∇ (b_ex: [State → ExprResult [State, bool]], body: [State → StmtResult [State]],
      <: PRED[[Ord, Ord]], variant: [State → Ord], Q: PRED[StmtResult [State]]):
while_invariant?(b_ex, body)(Q, Q) ∧ well_founded?(<) ⊃
  (∇ (s: State):
    Q(OK(s)) ∧
    OK?(b_ex(s)) ∧
    data(b_ex(s)) ∧ OK?((e2s(b_ex) ## body ## catch_continue)(s))
  ) ⊃
  OK?[State]
  ((##[State]
    (##[State](e2s[State, boolean](b_ex), body),
    catch_continue[State]))
  (s))
  ∨
  Break?[State]
  ((##[State]
    (##[State](e2s[State, boolean](b_ex), body),
    catch_continue[State]))
  (s))
  ∨
  Continue?[State]
  ((##[State]
    (##[State](e2s[State, boolean](b_ex), body),
    catch_continue[State]))
  (s))
  ∨
  Return?[State]
  ((##[State]
    (##[State](e2s[State, boolean](b_ex), body),
    catch_continue[State]))
  (s))
  ∨
  Switch?[State]
  ((##[State]
    (##[State](e2s[State, boolean](b_ex), body),
    catch_continue[State]))
  (s))
  ∨
  Default?[State]
  ((##[State]
    (##[State](e2s[State, boolean](b_ex), body),
    catch_continue[State]))
  (s))
  ∨
  Exception?[State]
  ((##[State]
    (##[State](e2s[State, boolean](b_ex), body),
    catch_continue[State]))
  (s))

```

Repeatedly Skolemizing and flattening,
 This completes the proof of `while_variant?_TCC1`.
 Q.E.D.

C.240.3 While_Hoare.while_no_cb_TCC1

Terse proof for `while_no_cb_TCC1`.

`while_no_cb_TCC1`:

$$\frac{\{1\} \quad \forall (b_ex: [State \rightarrow ExprResult[State, bool]], body: [State \rightarrow StmtResult[State]], s: State):}{(\exists (n: nat): while_termination_point?(b_ex, body, s)(n)) \supset nonempty?[nat](while_termination_point?[State](b_ex, body, s))}$$

Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of `while_no_cb_TCC1`.
 Q.E.D.

C.240.4 While_Hoare.while_no_cb_cb_while

Terse proof for `while_no_cb_cb_while`.

`while_no_cb_cb_while`:

$$\frac{\{1\} \quad \forall (b_ex: [State \rightarrow ExprResult[State, bool]], body: [State \rightarrow StmtResult[State]]):}{while(b_ex, body) = (while_no_cb(b_ex, body) \#\# catch_break)}$$

Repeatedly Skolemizing and flattening,
 Applying decompose-equality,
 Expanding the definition of `while`,
 Expanding the definition of `while_no_cb`,
 Trying repeated skolemization, instantiation, and if-lifting,
 This completes the proof of `while_no_cb_cb_while`.
 Q.E.D.

C.240.5 While_Hoare.iterate_while_invariant

Terse proof for `iterate_while_invariant`.

`iterate_while_invariant`:

$$\frac{\{1\} \quad \forall (b_ex: [State \rightarrow ExprResult[State, bool]], body: [State \rightarrow StmtResult[State]], s_0: State, inv_abnorm?, inv_norm?: PRED[StmtResult[State]], n: nat):}{(while_invariant?(b_ex, body)(inv_abnorm?, inv_norm?) \wedge inv_abnorm?(OK(s_0)) \wedge (\forall (m: nat): m < n \supset \neg while_termination_point?(b_ex, body, s_0)(m))) \supset inv_abnorm?(iterate_while(n, b_ex, body)(s_0))}$$

Inducting on `n` on formula 1,
 we get 2 subgoals:

iterate_while_invariant.1:

$\{1\} \quad \forall (b_ex: [State \rightarrow ExprResult[State, bool]], body: [State \rightarrow StmtResult[State]], s_0: State, inv_abnorm?, inv_norm?: PRED[StmtResult[State]]):$ $(while_invariant?(b_ex, body)(inv_abnorm?, inv_norm?) \wedge$ $inv_abnorm?(OK(s_0)) \wedge$ $(\forall (m: nat): m < 0 \supset \neg while_termination_point?(b_ex, body, s_0)(m)))$ $\supset inv_abnorm?(iterate_while(0, b_ex, body)(s_0))$

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `iterate_while_invariant.1`.

iterate_while_invariant.2:

$\{1\} \quad \forall j:$ $(\forall (b_ex: [State \rightarrow ExprResult[State, bool]], body: [State \rightarrow StmtResult[State]], s_0: State, inv_abnorm?, inv_norm?: PRED[StmtResult[State]]):$ $(while_invariant?(b_ex, body)(inv_abnorm?, inv_norm?) \wedge$ $inv_abnorm?(OK(s_0)) \wedge$ $(\forall (m: nat): m < j \supset \neg while_termination_point?(b_ex, body, s_0)(m)))$ $\supset inv_abnorm?(iterate_while(j, b_ex, body)(s_0)))$ \supset $(\forall (b_ex: [State \rightarrow ExprResult[State, bool]], body: [State \rightarrow StmtResult[State]], s_0: State, inv_abnorm?, inv_norm?: PRED[StmtResult[State]]):$ $(while_invariant?(b_ex, body)(inv_abnorm?, inv_norm?) \wedge$ $inv_abnorm?(OK(s_0)) \wedge$ $(\forall (m: nat):$ $m < j + 1 \supset \neg while_termination_point?(b_ex, body, s_0)(m)))$ $\supset inv_abnorm?(iterate_while(j + 1, b_ex, body)(s_0)))$
--

Repeatedly Skolemizing and flattening,
Instantiating quantified variables,
Instantiating the top quantifier in -2 with the terms: (s0!1),
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
we get 2 subgoals:

iterate_while_invariant.2.1:

$\{-1\} \quad j' \geq 0$ $\{-2\} \quad while_invariant?(b_ex', body')(inv_abnorm?', inv_norm?')$ $\{-3\} \quad inv_abnorm?(OK(s'_0))$ $\{-4\} \quad inv_abnorm?(iterate_while(j', b_ex', body')(s'_0))$ $\{-5\} \quad \forall (m: nat): m < 1 + j' \supset \neg while_termination_point?(b_ex', body', s'_0)(m)$ <hr/> $\{1\} \quad inv_abnorm?(iterate_while(1 + j', b_ex', body')(s'_0))$

Expanding the definition of `while_invariant?`,
Using lemma `iterate_while_right`,
Simplifying, rewriting, and recording with decision procedures,
Replacing using formula -1,
Instantiating the top quantifier in -6 with the terms: (j!1),
Expanding the definition of `while_termination_point?`,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Instantiating the top quantifier in -3 with the terms: (state(iterate_while(j!1, b_ex!1, body!1)(s0!1))),

Applying eta axiom scheme to `iterate_while(j!1, b_ex!1, body!1)(s0!1)`,
 Expanding the definition of `##`,
 Expanding the definition of `lift`,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 Expanding the definition of `skip`,
 Replacing using formula -1,
 which is trivially true.

This completes the proof of `iterate_while_invariant.2.1`.

`iterate_while_invariant.2.2`:

$\{-1\} \quad j' \geq 0$	$\{-2\} \quad \text{while_invariant?}(\text{b_ex}', \text{body}')(\text{inv_abnorm?}', \text{inv_norm?}')$
$\{-3\} \quad \text{inv_abnorm?}'(\text{OK}(s'_0))$	$\{-4\} \quad \forall (m: \text{nat}): m < 1 + j' \supset \neg \text{while_termination_point?}(\text{b_ex}', \text{body}', s'_0)(m)$
$\{1\} \quad \forall (m: \text{nat}): m < j' \supset \neg \text{while_termination_point?}(\text{b_ex}', \text{body}', s'_0)(m)$	$\{2\} \quad \text{inv_abnorm?}'(\text{iterate_while}(1 + j', \text{b_ex}', \text{body}') (s'_0))$

Repeatedly Skolemizing and flattening,
 Instantiating quantified variables,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 This completes the proof of `iterate_while_invariant.2.2`.
 Q.E.D.

C.240.6 While_Hoare.min_termination_point_no_less

Terse proof for `min_termination_point_no_less`.

`min_termination_point_no_less`:

$\{1\} \quad \forall (\text{b_ex}: [\text{State} \rightarrow \text{ExprResult}[\text{State}, \text{bool}]], \text{body}: [\text{State} \rightarrow \text{StmtResult}[\text{State}]],$	$s_0: \text{State}, m: \text{nat}):$
$(\exists (n: \text{nat}): \text{while_termination_point?}(\text{b_ex}, \text{body}, s_0)(n)) \wedge$	$m < \min[\text{nat}] (\text{while_termination_point?}(\text{b_ex}, \text{body}, s_0))$
$\supset \neg \text{while_termination_point?}(\text{b_ex}, \text{body}, s_0)(m)$	

Repeatedly Skolemizing and flattening,
 Adding type constraints for `min[nat](while_termination_point?(b_ex!1, body!1, s0!1))`,
 Instantiating quantified variables,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 This completes the proof of `min_termination_point_no_less`.
 Q.E.D.

C.240.7 While_Hoare.while_terminates

The \LaTeX code for this proof is broken.

C.240.8 While_Hoare.while

The \LaTeX code for this proof is broken.

C.241 Proofs for While_Rewrites (statement-rewrites.pvs)

C.241.1 While_Rewrites.while_to_while_no_cb

Terse proof for while_to_while_no_cb.

while_to_while_no_cb:

$$\{1\} \quad \forall (<: \text{PRED}[[\text{Ord}, \text{Ord}]], P, Q, R: [\text{State} \rightarrow \text{PRED}[\text{StmtResult}[\text{State}]]], \\ \text{b_ex}: [\text{State} \rightarrow \text{ExprResult}[\text{State}, \text{bool}]], \text{body}: [\text{State} \rightarrow \text{StmtRe-} \\ \text{sult}[\text{State}]], \\ \text{variant}: [\text{State} \rightarrow [\text{State} \rightarrow \text{Ord}]]): \\ \text{while}[\text{State}, \text{Ord}](\text{b_ex}, \text{body})(P, Q, R, \text{variant}, <) = \\ (\text{while_no_cb}(\text{b_ex}, \text{body})(P, Q, R, \text{variant}, <) \text{ ## } \text{catch_break})$$

Expanding the definition of while_no_cb,
Expanding the definition of while,
Repeatedly Skolemizing and flattening,
Using lemma while_no_cb_cb_while[State, Ord],
This completes the proof of while_to_while_no_cb.
Q.E.D.

C.241.2 While_Rewrites.while_inv_rewrite_termination_result

Terse proof for while_inv_rewrite_termination_result.

while_inv_rewrite_termination_result:

$$\{1\} \quad \forall (<: \text{PRED}[[\text{Ord}, \text{Ord}]], P, Q, R: [\text{State} \rightarrow \text{PRED}[\text{StmtResult}[\text{State}]]], \\ S: \text{PRED}[\text{StmtResult}[\text{State}]], \text{b_ex}: [\text{State} \rightarrow \text{ExprResult}[\text{State}, \text{bool}]], \\ \text{body}: [\text{State} \rightarrow \text{StmtResult}[\text{State}]], s: \text{State}, \text{vari-} \\ \text{ant}: [\text{State} \rightarrow [\text{State} \rightarrow \text{Ord}]]): \\ (P(s)(\text{skip}(s)) \wedge \\ R(s)(\text{skip}(s)) \wedge \\ \text{while_variant?}(\text{b_ex}, \text{body}, <)(\text{variant}(s), R(s)) \wedge \\ \text{while_invariant?}(\text{b_ex}, \text{body})(P(s), Q(s)) \wedge \\ (\forall (\text{res}: \text{StmtResult}[\text{State}]): R(s)(\text{res}) \supset S(\text{res}))) \\ \supset S(\text{while_no_cb}(\text{b_ex}, \text{body})(P, Q, R, \text{variant}, <)(s))$$

Repeatedly Skolemizing and flattening,
Using lemma hoare_while[State, Ord],
Installing automatic rewrites from: skip while_no_cb
Repeatedly simplifying with decision procedures, rewriting, propositional reasoning, quantifier instantiation, skolemization, if-lifting and equality replacement,
This completes the proof of while_inv_rewrite_termination_result.
Q.E.D.

C.241.3 While_Rewrites.while_inv_rewrite_data_ok

Terse proof for while_inv_rewrite_data_ok.

while_inv_rewrite_data_ok:

$$\begin{array}{l}
 \{1\} \quad \forall (<: \text{PRED}[[\text{Ord}, \text{Ord}]], P, Q, R: [\text{State} \rightarrow \text{PRED}[\text{StmtResult}[\text{State}]]], \\
 \quad S: \text{PRED}[\text{StmtResult}[\text{State}]], b_ex: [\text{State} \rightarrow \text{ExprResult}[\text{State}, \text{bool}]], \\
 \quad \text{body}: [\text{State} \rightarrow \text{StmtResult}[\text{State}]], s: \text{State}, \text{vari-} \\
 \text{ant}: [\text{State} \rightarrow [\text{State} \rightarrow \text{Ord}]]): \\
 \quad (P(s)(\text{skip}(s)) \wedge \\
 \quad R(s)(\text{skip}(s)) \wedge \\
 \quad \text{while_variant?}(b_ex, \text{body}, <)(\text{variant}(s), R(s)) \wedge \\
 \quad \text{while_invariant?}(b_ex, \text{body})(P(s), Q(s)) \wedge \\
 \quad \text{OK?}(\text{while_no_cb}(b_ex, \text{body})(P, Q, R, \text{variant}, <)(s)) \wedge \\
 \quad (\forall (\text{res}: \text{StmtResult}[\text{State}]): Q(s)(\text{res}) \supset S(\text{res}))) \\
 \quad \supset S(\text{while_no_cb}(b_ex, \text{body})(P, Q, R, \text{variant}, <)(s))
 \end{array}$$

Installing automatic rewrites from: skip while_no_cb

Repeatedly Skolemizing and flattening,

Using lemma hoare_while[State, Ord],

Repeatedly simplifying with decision procedures, rewriting, propositional reasoning, quantifier instantiation, skolemization, if-lifting and equality replacement,

This completes the proof of while_inv_rewrite_data_ok.

Q.E.D.

C.241.4 While_Rewrites.while_inv_rewrite_data_break

Terse proof for while_inv_rewrite_data_break.

while_inv_rewrite_data_break:

$$\begin{array}{l}
 \{1\} \quad \forall (<: \text{PRED}[[\text{Ord}, \text{Ord}]], P, Q, R: [\text{State} \rightarrow \text{PRED}[\text{StmtResult}[\text{State}]]], \\
 \quad S: \text{PRED}[\text{StmtResult}[\text{State}]], b_ex: [\text{State} \rightarrow \text{ExprResult}[\text{State}, \text{bool}]], \\
 \quad \text{body}: [\text{State} \rightarrow \text{StmtResult}[\text{State}]], s: \text{State}, \text{vari-} \\
 \text{ant}: [\text{State} \rightarrow [\text{State} \rightarrow \text{Ord}]]): \\
 \quad (P(s)(\text{skip}(s)) \wedge \\
 \quad R(s)(\text{skip}(s)) \wedge \\
 \quad \text{while_variant?}(b_ex, \text{body}, <)(\text{variant}(s), R(s)) \wedge \\
 \quad \text{while_invariant?}(b_ex, \text{body})(P(s), Q(s)) \wedge \\
 \quad \text{Break?}(\text{while_no_cb}(b_ex, \text{body})(P, Q, R, \text{variant}, <)(s)) \wedge \\
 \quad (\forall (\text{res}: \text{StmtResult}[\text{State}]): \\
 \quad \quad (P(s)(\text{res}) \vee Q(s)(\text{res})) \wedge \text{Break?}(\text{res}) \supset S(\text{res}))) \\
 \quad \supset S(\text{while_no_cb}(b_ex, \text{body})(P, Q, R, \text{variant}, <)(s))
 \end{array}$$

Installing automatic rewrites from: skip while_no_cb

Repeatedly Skolemizing and flattening,

Using lemma hoare_while[State, Ord],

Repeatedly simplifying with decision procedures, rewriting, propositional reasoning, quantifier instantiation, skolemization, if-lifting and equality replacement,

This completes the proof of while_inv_rewrite_data_break.

Q.E.D.

C.241.5 While_Rewrites.while_inv_rewrite_data_return

Terse proof for while_inv_rewrite_data_return.

while_inv_rewrite_data_return:

$$\{1\} \quad \forall (<: \text{PRED}[[\text{Ord}, \text{Ord}]], P, Q, R: [\text{State} \rightarrow \text{PRED}[\text{StmtResult}[\text{State}]]], \\ S: \text{PRED}[\text{StmtResult}[\text{State}]], \text{b_ex}: [\text{State} \rightarrow \text{ExprResult}[\text{State}, \text{bool}]], \\ \text{body}: [\text{State} \rightarrow \text{StmtResult}[\text{State}]], s: \text{State}, \text{vari-} \\ \text{ant}: [\text{State} \rightarrow [\text{State} \rightarrow \text{Ord}]]): \\ (P(s)(\text{skip}(s)) \wedge \\ R(s)(\text{skip}(s)) \wedge \\ \text{while_variant?}(\text{b_ex}, \text{body}, <)(\text{variant}(s), R(s)) \wedge \\ \text{while_invariant?}(\text{b_ex}, \text{body})(P(s), Q(s)) \wedge \\ \text{Return?}(\text{while_no_cb}(\text{b_ex}, \text{body}))(P, Q, R, \text{variant}, <)(s) \wedge \\ (\forall (\text{res}: \text{StmtResult}[\text{State}]): \\ (P(s)(\text{res}) \vee Q(s)(\text{res})) \wedge \text{Return?}(\text{res}) \supset S(\text{res}))) \\ \supset S(\text{while_no_cb}(\text{b_ex}, \text{body}))(P, Q, R, \text{variant}, <)(s))$$

Installing automatic rewrites from: skip while_no_cb

Repeatedly Skolemizing and flattening,

Using lemma hoare_while[State, Ord],

Repeatedly simplifying with decision procedures, rewriting, propositional reasoning, quantifier instantiation, skolemization, if-lifting and equality replacement,

This completes the proof of while_inv_rewrite_data_return.

Q.E.D.

C.241.6

While_Rewrites.stmt_while_inv_rewrite_termination_result_TCC1

Terse proof for stmt_while_inv_rewrite_termination_result_TCC1.

stmt_while_inv_rewrite_termination_result_TCC1:

$$\{1\} \quad \forall (s: \text{State}, \text{stmt}: [\text{State} \rightarrow \text{StmtResult}[\text{State}]]): \\ \text{OK?}(\text{stmt}(s)) \supset \\ \text{OK?}[\text{State}](\text{stmt}(s)) \vee \\ \text{Break?}[\text{State}](\text{stmt}(s)) \vee \\ \text{Continue?}[\text{State}](\text{stmt}(s)) \vee \\ \text{Return?}[\text{State}](\text{stmt}(s)) \vee \\ \text{Switch?}[\text{State}](\text{stmt}(s)) \vee \\ \text{Default?}[\text{State}](\text{stmt}(s)) \vee \text{Exception?}[\text{State}](\text{stmt}(s))$$

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of stmt_while_inv_rewrite_termination_result_TCC1.

Q.E.D.

C.241.7 While_Rewrites.stmt_while_inv_rewrite_termination_result

Terse proof for stmt_while_inv_rewrite_termination_result.

stmt_while_inv_rewrite_termination_result:

$\{1\} \quad \forall (<: \text{PRED}[[\text{Ord}, \text{Ord}]], P, Q, R: [\text{State} \rightarrow \text{PRED}[\text{StmtResult}[\text{State}]]], \\ S: \text{PRED}[\text{StmtResult}[\text{State}]], b_ex: [\text{State} \rightarrow \text{ExprResult}[\text{State}, \text{bool}]], \\ \text{body}: [\text{State} \rightarrow \text{StmtResult}[\text{State}]], s: \text{State}, \text{stmt}: [\text{State} \rightarrow \text{StmtResult}[\text{State}]], \\ \text{variant}: [\text{State} \rightarrow [\text{State} \rightarrow \text{Ord}]]): \\ (\text{OK}?(s) \wedge \\ P(\text{state}(\text{stmt}(s)))(\text{stmt}(s)) \wedge \\ R(\text{state}(\text{stmt}(s)))(\text{stmt}(s)) \wedge \\ \text{while_variant?}(b_ex, \text{body}, <) \\ (\text{variant}(\text{state}(\text{stmt}(s))), R(\text{state}(\text{stmt}(s)))) \\ \wedge \\ \text{while_invariant?}(b_ex, \text{body})(P(\text{state}(\text{stmt}(s))), Q(\text{state}(\text{stmt}(s)))) \wedge \\ (\forall (\text{res}: \text{StmtResult}[\text{State}]): R(\text{state}(\text{stmt}(s)))(\text{res}) \supset S(\text{res}))) \\ \supset S((\text{stmt} \ \#\# \ \text{while_no_cb}(b_ex, \text{body})(P, Q, R, \text{variant}, <))(s))$
--

Repeatedly Skolemizing and flattening,
 Installing automatic rewrites from: ## skip lift!
 Using lemma while_inv_rewrite_termination_result,
 Applying eta axiom scheme to stmt!1(s!1),
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 This completes the proof of stmt_while_inv_rewrite_termination_result.
 Q.E.D.

C.241.8 While_Rewrites.stmt_while_inv_rewrite_data_ok

Terse proof for stmt_while_inv_rewrite_data_ok.

stmt_while_inv_rewrite_data_ok:

$\{1\} \quad \forall (<: \text{PRED}[[\text{Ord}, \text{Ord}]], P, Q, R: [\text{State} \rightarrow \text{PRED}[\text{StmtResult}[\text{State}]]], \\ S: \text{PRED}[\text{StmtResult}[\text{State}]], b_ex: [\text{State} \rightarrow \text{ExprResult}[\text{State}, \text{bool}]], \\ \text{body}: [\text{State} \rightarrow \text{StmtResult}[\text{State}]], s: \text{State}, \text{stmt}: [\text{State} \rightarrow \text{StmtResult}[\text{State}]], \\ \text{variant}: [\text{State} \rightarrow [\text{State} \rightarrow \text{Ord}]]): \\ (\text{OK}?(s) \wedge \\ P(\text{state}(\text{stmt}(s)))(\text{stmt}(s)) \wedge \\ R(\text{state}(\text{stmt}(s)))(\text{stmt}(s)) \wedge \\ \text{while_variant?}(b_ex, \text{body}, <) \\ (\text{variant}(\text{state}(\text{stmt}(s))), R(\text{state}(\text{stmt}(s)))) \\ \wedge \\ \text{while_invariant?}(b_ex, \text{body})(P(\text{state}(\text{stmt}(s))), Q(\text{state}(\text{stmt}(s)))) \wedge \\ \text{OK}?(s) \wedge \\ (\forall (\text{res}: \text{StmtResult}[\text{State}]): Q(\text{state}(\text{stmt}(s)))(\text{res}) \supset S(\text{res}))) \\ \supset S((\text{stmt} \ \#\# \ \text{while_no_cb}(b_ex, \text{body})(P, Q, R, \text{variant}, <))(s))$
--

Repeatedly Skolemizing and flattening,
 Installing automatic rewrites from: ## skip lift!
 Applying eta axiom scheme to stmt!1(s!1),
 Using lemma while_inv_rewrite_data_ok,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 This completes the proof of stmt_while_inv_rewrite_data_ok.

Q.E.D.

C.241.9 While_Rewrites.stmt_while_inv_rewrite_data_break

Terse proof for `stmt_while_inv_rewrite_data_break`.

`stmt_while_inv_rewrite_data_break`:

$$\begin{array}{l}
 \{1\} \quad \forall (<: \text{PRED}[[\text{Ord}, \text{Ord}]], P, Q, R: [\text{State} \rightarrow \text{PRED}[\text{StmtResult}[\text{State}]]], \\
 \quad S: \text{PRED}[\text{StmtResult}[\text{State}]], b_ex: [\text{State} \rightarrow \text{ExprResult}[\text{State}, \text{bool}]], \\
 \quad \text{body}: [\text{State} \rightarrow \text{StmtResult}[\text{State}]], s: \text{State}, \text{stmt}: [\text{State} \rightarrow \text{StmtRe-} \\
 \quad \text{sult}[\text{State}]], \\
 \quad \text{variant}: [\text{State} \rightarrow [\text{State} \rightarrow \text{Ord}]]): \\
 \quad (\text{OK?}(\text{stmt}(s)) \wedge \\
 \quad \quad P(\text{state}(\text{stmt}(s)))(\text{stmt}(s)) \wedge \\
 \quad \quad R(\text{state}(\text{stmt}(s)))(\text{stmt}(s)) \wedge \\
 \quad \quad \text{while_variant?}(b_ex, \text{body}, <) \\
 \quad \quad \quad (\text{variant}(\text{state}(\text{stmt}(s))), R(\text{state}(\text{stmt}(s)))) \\
 \quad \quad \wedge \\
 \quad \quad \text{while_invariant?}(b_ex, \text{body})(P(\text{state}(\text{stmt}(s))), Q(\text{state}(\text{stmt}(s)))) \wedge \\
 \quad \quad \text{Break?}((\text{stmt} \## \text{while_no_cb}(b_ex, \text{body})(P, Q, R, \text{variant}, <))(s)) \wedge \\
 \quad \quad (\forall (\text{res}: \text{StmtResult}[\text{State}]): \\
 \quad \quad \quad (P(\text{state}(\text{stmt}(s)))(\text{res}) \vee Q(\text{state}(\text{stmt}(s)))(\text{res})) \wedge \\
 \quad \quad \quad \text{Break?}(\text{res}) \\
 \quad \quad \quad \supset S(\text{res}))) \\
 \quad \supset S((\text{stmt} \## \text{while_no_cb}(b_ex, \text{body})(P, Q, R, \text{variant}, <))(s))
 \end{array}$$

Repeatedly Skolemizing and flattening,

Installing automatic rewrites from: `## skip lift!`

Applying eta axiom scheme to `stmt!1(s!1)`,

Using lemma `while_inv_rewrite_data_break`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `stmt_while_inv_rewrite_data_break`.

Q.E.D.

C.241.10 While_Rewrites.stmt_while_inv_rewrite_data_return

Terse proof for `stmt_while_inv_rewrite_data_return`.

stmt_while_inv_rewrite_data_return:

$\{1\} \quad \forall (<: \text{PRED}[[\text{Ord}, \text{Ord}]], P, Q, R: [\text{State} \rightarrow \text{PRED}[\text{StmtResult}[\text{State}]]], \\ S: \text{PRED}[\text{StmtResult}[\text{State}]], \text{b_ex}: [\text{State} \rightarrow \text{ExprResult}[\text{State}, \text{bool}]], \\ \text{body}: [\text{State} \rightarrow \text{StmtResult}[\text{State}]], s: \text{State}, \text{stmt}: [\text{State} \rightarrow \text{StmtRe-} \\ \text{sult}[\text{State}]], \\ \text{variant}: [\text{State} \rightarrow [\text{State} \rightarrow \text{Ord}]]): \\ (\text{OK}?(\text{stmt}(s)) \wedge \\ P(\text{state}(\text{stmt}(s)))(\text{stmt}(s)) \wedge \\ R(\text{state}(\text{stmt}(s)))(\text{stmt}(s)) \wedge \\ \text{while_variant}?(\text{b_ex}, \text{body}, <) \\ (\text{variant}(\text{state}(\text{stmt}(s))), R(\text{state}(\text{stmt}(s)))) \\ \wedge \\ \text{while_invariant}?(\text{b_ex}, \text{body})(P(\text{state}(\text{stmt}(s))), Q(\text{state}(\text{stmt}(s)))) \wedge \\ \text{Return}?((\text{stmt} \#\# \text{while_no_cb}(\text{b_ex}, \text{body})(P, Q, R, \text{variant}, <))(s)) \wedge \\ (\forall (\text{res}: \text{StmtResult}[\text{State}]): \\ (P(\text{state}(\text{stmt}(s)))(\text{res}) \vee Q(\text{state}(\text{stmt}(s)))(\text{res})) \wedge \\ \text{Return}?(\text{res}) \\ \supset S(\text{res}))) \\ \supset S((\text{stmt} \#\# \text{while_no_cb}(\text{b_ex}, \text{body})(P, Q, R, \text{variant}, <))(s))$
--

Repeatedly Skolemizing and flattening,
 Installing automatic rewrites from: ## skip lift!
 Applying eta axiom scheme to stmt!1(s!1),
 Using lemma while_inv_rewrite_data_return,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 This completes the proof of stmt_while_inv_rewrite_data_return.
 Q.E.D.

C.241.11 While_Rewrites.pm_q_prop_while

Terse proof for pm_q_prop_while.

pm_q_prop_while:

$\{1\} \quad \forall (<: \text{PRED}[[\text{Ord}, \text{Ord}]], P, Q, R: [\text{State} \rightarrow \text{PRED}[\text{StmtResult}[\text{State}]]], \\ \text{b_ex}: [\text{State} \rightarrow \text{ExprResult}[\text{State}, \text{bool}]], \text{body}: [\text{State} \rightarrow \text{StmtRe-} \\ \text{sult}[\text{State}]], \\ \text{pm}: \text{Plain_Memory}[\text{State}], s: \text{State}, \text{variant}: [\text{State} \rightarrow [\text{State} \rightarrow \text{Ord}]]): \\ \text{LET } \text{pm_q_prop_inv} = \\ \quad \lambda (s_1: \text{State}): \lambda (\text{res}: \text{StmtResult}[\text{State}]): \text{pm_q_prop}(\text{pm})(\text{catch_break}(\text{res})) \\ \text{IN} \\ \text{pm}'\text{states}(s) \wedge \\ R(s)(\text{skip}(s)) \wedge \\ \text{while_variant}?(\text{b_ex}, \text{body}, <)(\text{variant}(s), R(s)) \wedge \\ \text{while_invariant}?(\text{b_ex}, \text{body})(\text{pm_q_prop_inv}(s), \text{pm_q_prop_inv}(s)) \\ \supset \text{pm_q_prop}(\text{pm})(\text{while}(\text{b_ex}, \text{body})(P, Q, R, \text{variant}, <))(s)$

Repeatedly Skolemizing and flattening,
 Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
 Using lemma hoare_while[State, Ord],
 Rewriting using while_to_while_no_cb, matching in *,
 Expanding the definition of ##,

C Proof scripts

Expanding the definition of `while_no_cb`,

Expanding the definition of `skip`,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Expanding the definition of `pm_q_prop`,

Expanding the definition of `catch_break`,

which is trivially true.

This completes the proof of `pm_q_prop_while`.

Q.E.D.

C.241.12 While_Rewrites.pm_q_prop_stmt_while_TCC1

Terse proof for `pm_q_prop_stmt_while_TCC1`.

`pm_q_prop_stmt_while_TCC1`:

$$\begin{array}{l} \{1\} \quad \forall (pm: \text{Plain_Memory}[State], s: State, stmt: [State \rightarrow \text{StmtResult}[State]], \\ \quad pm_q_prop_inv: [State \rightarrow [StmtResult[State] \rightarrow \text{bool}]]) : \\ (pm_q_prop(pm)(stmt(s)) \wedge \\ \quad pm_q_prop_inv = \\ \quad (\lambda (s_1: State): \\ \quad \quad \lambda (res: \text{StmtResult}[State]): pm_q_prop(pm)(catch_break(res)))) \\ \supset \\ \text{OK?}[State](stmt(s)) \vee \\ \text{Break?}[State](stmt(s)) \vee \\ \text{Continue?}[State](stmt(s)) \vee \\ \text{Return?}[State](stmt(s)) \vee \\ \text{Switch?}[State](stmt(s)) \vee \\ \text{Default?}[State](stmt(s)) \vee \text{Exception?}[State](stmt(s)) \end{array}$$

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `pm_q_prop_stmt_while_TCC1`.

Q.E.D.

C.241.13 While_Rewrites.pm_q_prop_stmt_while

Terse proof for `pm_q_prop_stmt_while`.

pm_q_prop_stmt_while:

```

{1}  ∀ (<: PRED[[Ord, Ord]], P, Q, R: [State → PRED[StmtResult[State]]],
      b_ex: [State → ExprResult[State, bool]], body: [State → StmtRe-
      sult[State]],
      pm: Plain_Memory[State], s: State, stmt: [State → StmtResult[State]],
      variant: [State → [State → Ord]]):
  LET pm_q_prop_inv =
      λ (s_1: State): λ (res: StmtResult[State]): pm_q_prop(pm)(catch_break(res))
  IN
  pm_q_prop(pm)(stmt(s)) ∧
  R(state(stmt(s)))(stmt(s)) ∧
  while_variant?(b_ex, body, <)(variant(state(stmt(s))), R(state(stmt(s))))
  ∧
  while_invariant?(b_ex, body)
  (pm_q_prop_inv(state(stmt(s))),
   pm_q_prop_inv(state(stmt(s))))
  ⊃ pm_q_prop(pm)((stmt ## while(b_ex, body))(P, Q, R, variant, <))(s)

```

Repeatedly Skolemizing and flattening,

Installing automatic rewrites from: ## skip lift!

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Using lemma pm_q_prop_while,

Applying eta axiom scheme to stmt!1(s!1),

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Expanding the definition of pm_q_prop,

which is trivially true.

This completes the proof of pm_q_prop_stmt_while.

Q.E.D.