A Formal CHERI-C Memory Model

Seung Hoon Park

February 6, 2026

Abstract

In this work, we present a formal memory model that provides a memory semantics for CHERI-C programs with uncompressed capabilities in a 'purecap' environment. We present a CHERI-C memory model theory with properties suitable for verification and potentially other types of analyses. Our theory generates an OCaml executable instance of the memory model, which is then used to instantiate the parametric *Gillian* program analysis framework, enabling concrete execution of CHERI-C programs. The tool can run a CHERI-C test suite, demonstrating the correctness of our tool, and catch a good class of safety violations that the CHERI hardware might miss.

The proof is accompanied by a paper [6] that explains the Gillian framework instantiation that uses the OCaml code generated by this work.

Contents

1 Preliminary Theories **3**
- 1.1 Types and Values . . . 3
- 1.2 Primitive Value Conversion and Cast Proof . . . 4

2 CHERI-C Error System **15**

3 Memory **18**
- 3.1 Definitions . . . 18
- 3.2 Properties . . . 20

4 Helper functions and lemmas **23**

5 Memory Actions / Operations **39**
- 5.1 Properties of the operations . . . 42
  - 5.1.1 Correctness Properties . . . 42
  - 5.1.2 Good Variable Laws . . . 57
  - 5.1.3 Miscellaneous Laws . . . 60
- 5.2 Well-formedness of actions . . . 60
- 5.3 memcpy formalisation . . . 61

6 Miscellaneous Definitions **62**

7 Code Generation **64**

Acknowledgements

The author was funded by the UKRI programme on Digital Security by Design (Ref. EP/V000225/1, SCorCH).

theory *Preliminary-Library*
imports *Main HOL−Library.Word Word-Lib.Word-Lib-Sumo HOL−Library.Countable*
begin

1 Preliminary Theories

In this subsection, we provide the type and value system used by the CHERI-C Memory Model. We also provide proofs for the conversion between large words (i.e. bits) and a list of bytes. For primitive bytes that are not $U8_\tau$ or $S8_\tau$, we need to be able to convert between their normal representation and list of bytes so that storing values work as intended. The high-level detail is given in the paper [6].

1.1 Types and Values

We first formalise the capability type. We first define *memory capabilities* as a record, then we define *tagged capabilities* by extending the record. We state the class comp_countable for future work, but countable is sufficient for the block_id type. For the permissions, we present only those used by the memory model.

class *comp-countable* = *countable* + *zero* + *ord*

record ('*a* :: *comp-countable*) *mem-capability* =
  block-id :: '*a*
  offset :: *int*
  base :: *nat*
  len :: *nat*
  perm-load :: *bool*
  perm-cap-load :: *bool*
  perm-store :: *bool*
  perm-cap-store :: *bool*
  perm-cap-store-local :: *bool*
  perm-global :: *bool*

record ('*a* :: *comp-countable*) *capability* = '*a mem-capability* +
  tag :: *bool*

cctype corresponds to τ in the paper [6], where τ is the type system.

datatype *cctype* =
  Uint8
  | *Sint8*
  | *Uint16*
  | *Sint16*
  | *Uint32*
  | *Sint32*
  | *Uint64*

```
  | Sint64
  | Cap
```

'a ccval corresponds to $\mathcal{V}_{\mathcal{C}}$ in the paper [6]. *'a* in this instance must be countable.

```
datatype 'a ccval =
  Uint8-v       8 word
  | Sint8-v     8 sword
  | Uint16-v    16 word
  | Sint16-v    16 sword
  | Uint32-v    32 word
  | Sint32-v    32 sword
  | Uint64-v    64 word
  | Sint64-v    64 sword
  | Cap-v       'a capability
  | Cap-v-frag 'a capability nat
  | Undef
```

memval_type infers the type of a value.

```
fun memval-type :: 'a ccval ⇒ cctype
  where
  memval-type v = (case v of
     Uint8-v  - ⇒ Uint8
   | Sint8-v  - ⇒ Sint8
   | Uint16-v - ⇒ Uint16
   | Sint16-v - ⇒ Sint16
   | Uint32-v - ⇒ Uint32
   | Sint32-v - ⇒ Sint32
   | Uint64-v - ⇒ Uint64
   | Sint64-v - ⇒ Sint64
   | Cap-v    - ⇒ Cap
   | Cap-v-frag - - ⇒ Uint8)
```

1.2 Primitive Value Conversion and Cast Proof

In this subsection, we provide proofs for the conversion between words and list of words. We also provide proofs that casting primitive values is correct. These will be used by the `load` and `store` operations in the memory model.

```
abbreviation encode-u8 :: nat ⇒ 8 word
  where
  encode-u8 x ≡ word-of-nat x

abbreviation decode-u8 :: 8 word ⇒ nat
  where
  decode-u8 b ≡ unat b

abbreviation encode-u8-list :: 8 word ⇒ 8 word list
  where
```

encode-u8-list $w \equiv [w]$

abbreviation *decode-u8-list* :: *8 word list* ⇒ *8 word*
where
decode-u8-list ls ≡ *hd ls*

lemma *encode-decode-u8-list*:
$ls = [b] \Longrightarrow ls =$ *encode-u8-list* (*decode-u8-list ls*)
⟨*proof*⟩

lemma *decode-encode-u8-list*:
$w =$ *decode-u8-list* (*encode-u8-list w*)
⟨*proof*⟩

lemma *encode-decode-u8*:
$w =$ *encode-u8* (*decode-u8 w*)
⟨*proof*⟩

lemma *decode-encode-u8*:
assumes $i \leq 2 \hat{} LENGTH(8) - 1$
shows $i =$ *decode-u8* (*encode-u8 i*)
⟨*proof*⟩

abbreviation *u64-split* :: *64 word* ⇒ *32 word list*
where
u64-split x ≡ (*word-rsplit* :: *64 word* ⇒ *32 word list*) *x*

abbreviation *u32-split* :: *32 word* ⇒ *16 word list*
where
u32-split x ≡ (*word-rsplit* :: *32 word* ⇒ *16 word list*) *x*

abbreviation *u16-split* :: *16 word* ⇒ *8 word list*
where
u16-split x ≡ (*word-rsplit* :: *16 word* ⇒ *8 word list*) *x*

abbreviation *cat-u16* :: *8 word list* ⇒ *16 word*
where
cat-u16 x ≡ (*word-rcat* :: *8 word list* ⇒ *16 word*) *x*

abbreviation *encode-u16* :: *nat* ⇒ *8 word list*
where
encode-u16 x ≡ *u16-split* (*word-of-nat x*)

abbreviation *decode-u16* :: *8 word list* ⇒ *nat*
where
decode-u16 x ≡ *unat* (*cat-u16 x*)

lemma *flatten-u16-length*:
length (*u16-split vs*) = *2*

⟨*proof*⟩

lemma *rsplit-rcat-eq*:
assumes *LENGTH*(('*b*::*len*)) *mod LENGTH*(('*a*::*len*)) = *0*
and *length w* = *LENGTH*('*b*) *div LENGTH*('*a*)
shows (*word-rsplit* :: '*b word* ⇒ '*a word list*) ((*word-rcat* :: '*a word list* ⇒ '*b word*) *w*) = *w*
⟨*proof*⟩

lemma *rsplit-rcat-u16-eq*:
assumes *w* = [*a1*, *a2*]
shows (*word-rsplit* :: *16 word* ⇒ *8 word list*) ((*word-rcat* :: *8 word list* ⇒ *16 word*) *w*) = *w*
⟨*proof*⟩

lemma *encode-decode-u16*:
assumes *w* = [*a*, *b*]
shows *w* = *encode-u16* (*decode-u16 w*)
⟨*proof*⟩

lemma *cat-flatten-u16-eq*:
cat-u16 (*u16-split w*) = *w*
⟨*proof*⟩

lemma *decode-encode-u16*:
assumes *i* ≤ *2* ^ *LENGTH*(*16*) − *1*
shows *i* = *decode-u16* (*encode-u16 i*)
⟨*proof*⟩

abbreviation *flatten-u32* :: *32 word* ⇒ *8 word list*
where
flatten-u32 x ≡ (*word-rsplit* :: *32 word* ⇒ *8 word list*) *x*

abbreviation *cat-u32* :: *8 word list* ⇒ *32 word*
where
cat-u32 x ≡ (*word-rcat* :: *8 word list* ⇒ *32 word*) *x*

abbreviation *encode-u32* :: *nat* ⇒ *8 word list*
where
encode-u32 x ≡ *flatten-u32* (*word-of-nat x*)

abbreviation *decode-u32* :: *8 word list* ⇒ *nat*
where
decode-u32 i ≡ *unat* (*cat-u32 i*)

lemma *flatten-u32-length*:
length (*flatten-u32 vs*) = *4*
⟨*proof*⟩

lemma *rsplit-rcat-u32-eq*:
assumes $w = [a1, a2, b1, b2]$
shows (*word-rsplit* :: *32 word* ⇒ *8 word list*) ((*word-rcat* :: *8 word list* ⇒ *32 word*) *w*) = *w*
⟨*proof*⟩

lemma *encode-decode-u32*:
assumes $w = [a1, a2, b1, b2]$
shows *w* = *encode-u32* (*decode-u32 w*)
⟨*proof*⟩

lemma *cat-flatten-u32-eq*:
cat-u32 (*flatten-u32 w*) = *w*
⟨*proof*⟩

lemma *decode-encode-u32*:
assumes *i* ≤ *2* ^ *LENGTH*(*32*) − *1*
shows *i* = *decode-u32* (*encode-u32 i*)
⟨*proof*⟩

abbreviation *flatten-u64* :: *64 word* ⇒ *8 word list*
where
flatten-u64 x ≡ (*word-rsplit* :: *64 word* ⇒ *8 word list*) *x*

abbreviation *cat-u64* :: *8 word list* ⇒ *64 word*
where
cat-u64 x ≡ *word-rcat x*

abbreviation *encode-u64* :: *nat* ⇒ *8 word list*
where
encode-u64 x ≡ *flatten-u64* (*word-of-nat x*)

abbreviation *decode-u64* :: *8 word list* ⇒ *nat*
where
decode-u64 x ≡ *unat* (*cat-u64 x*)

lemma *flatten-u64-length*:
length (*flatten-u64 vs*) = *8*
⟨*proof*⟩

lemma *encode-decode-u64*:
assumes $w = [a1, a2, b1, b2, c1, c2, d1, d2]$
shows *w* = *encode-u64* (*decode-u64 w*)
⟨*proof*⟩

lemma *cat-flatten-u64-eq*:
cat-u64 (*flatten-u64 w*) = *w*
⟨*proof*⟩

lemma *decode-encode-u64*:
assumes $i \leq 2 \hat{} LENGTH(64) - 1$
shows $i = decode\text{-}u64\ (encode\text{-}u64\ i)$
⟨*proof*⟩

abbreviation *encode-s8* :: *int* ⇒ *8 sword*
where
encode-s8 x ≡ *word-of-int x*

abbreviation *decode-s8* :: *8 sword* ⇒ *int*
where
decode-s8 b ≡ *sint b*

abbreviation *encode-s8-list* :: *8 sword* ⇒ *8 word list*
where
encode-s8-list w ≡ [*SCAST*(*8 signed* → *8*) *w*]

abbreviation *decode-s8-list* :: *8 word list* ⇒ *8 sword*
where
decode-s8-list ls ≡ *UCAST*(*8* → *8 signed*) (*hd ls*)

lemma *encode-decode-s8-list*:
$ls = [b] \Longrightarrow ls = encode\text{-}s8\text{-}list\ (decode\text{-}s8\text{-}list\ ls)$
⟨*proof*⟩

lemma *decode-encode-s8-list*:
$w = decode\text{-}s8\text{-}list\ (encode\text{-}s8\text{-}list\ w)$
⟨*proof*⟩

lemma *encode-decode-s8*:
$w = encode\text{-}s8\ (decode\text{-}s8\ w)$
⟨*proof*⟩

lemma *decode-encode-s8*:
assumes $-(2 \hat{} (LENGTH(8) - 1)) \leq i$
and $i < 2 \hat{} (LENGTH(8) - 1)$
shows $i = decode\text{-}s8\ (encode\text{-}s8\ i)$
⟨*proof*⟩

abbreviation *s64-split* :: *64 sword* ⇒ *32 word list*
where
s64-split x ≡ (*word-rsplit* :: *64 sword* ⇒ *32 word list*) *x*

abbreviation *s32-split* :: *32 sword* ⇒ *16 word list*
where
s32-split x ≡ (*word-rsplit* :: *32 sword* ⇒ *16 word list*) *x*

abbreviation *s16-split* :: *16 sword* ⇒ *8 word list*

where
s16-split x ≡ (*word-rsplit* :: *16 sword* ⇒ *8 word list*) *x*

abbreviation *cat-s16* :: *8 word list* ⇒ *16 sword*
where
cat-s16 x ≡ (*word-rcat* :: *8 word list* ⇒ *16 sword*) *x*

abbreviation *encode-s16* :: *int* ⇒ *8 word list*
where
encode-s16 x ≡ *s16-split* (*word-of-int x*)

abbreviation *decode-s16* :: *8 word list* ⇒ *int*
where
decode-s16 x ≡ *sint* (*cat-s16 x*)

lemma *flatten-s16-length*:
length (*s16-split vs*) = *2*
⟨*proof*⟩

lemma *rsplit-rcat-s16-eq*:
assumes *w* = [*a1*, *a2*]
shows (*word-rsplit* :: *16 sword* ⇒ *8 word list*) ((*word-rcat* :: *8 word list* ⇒ *16 sword*) *w*) = *w*
⟨*proof*⟩

lemma *encode-decode-s16*:
assumes *w* = [*a*, *b*]
shows *w* = *encode-s16* (*decode-s16 w*)
⟨*proof*⟩

lemma *cat-flatten-s16-eq*:
cat-s16 (*s16-split w*) = *w*
⟨*proof*⟩

lemma *decode-encode-s16*:
assumes − (*2* ^ (*LENGTH*(*16*) − *1*)) ≤ *i*
and *i* < *2* ^ (*LENGTH*(*16*) − *1*)
shows *i* = *decode-s16* (*encode-s16 i*)
⟨*proof*⟩

abbreviation *flatten-s32* :: *32 sword* ⇒ *8 word list*
where
flatten-s32 x ≡ (*word-rsplit* :: *32 sword* ⇒ *8 word list*) *x*

abbreviation *cat-s32* :: *8 word list* ⇒ *32 sword*
where
cat-s32 x ≡ (*word-rcat* :: *8 word list* ⇒ *32 sword*) *x*

abbreviation *encode-s32* :: *int* ⇒ *8 word list*
  where
  encode-s32 x ≡ *flatten-s32* (*word-of-int x*)

abbreviation *decode-s32* :: *8 word list* ⇒ *int*
  where
  decode-s32 i ≡ *sint* (*cat-s32 i*)

lemma *flatten-s32-length*:
  length (*flatten-s32 vs*) = *4*
  ⟨*proof*⟩

lemma *rsplit-rcat-s32-eq*:
  assumes *w* = [*a1*, *a2*, *b1*, *b2*]
  shows (*word-rsplit* :: *32 sword* ⇒ *8 word list*) ((*word-rcat* :: *8 word list* ⇒ *32 sword*) *w*) = *w*
  ⟨*proof*⟩

lemma *encode-decode-s32*:
  assumes *w* = [*a1*, *a2*, *b1*, *b2*]
  shows *w* = *encode-s32* (*decode-s32 w*)
  ⟨*proof*⟩

lemma *decode-encode-s32*:
  assumes − (*2* ^ (*LENGTH*(*32*) − *1*)) ≤ *i*
    and *i* < *2* ^ (*LENGTH*(*32*) − *1*)
  shows *i* = *decode-s32* (*encode-s32 i*)
  ⟨*proof*⟩

abbreviation *flatten-s64* :: *64 sword* ⇒ *8 word list*
  where
  flatten-s64 x ≡ (*word-rsplit* :: *64 sword* ⇒ *8 word list*) *x*

lemma *flatten-s64-length*:
  length (*flatten-s64 vs*) = *8*
  ⟨*proof*⟩

abbreviation *cat-s64* :: *8 word list* ⇒ *64 sword*
  where
  cat-s64 x ≡ *word-rcat x*

abbreviation *encode-s64* :: *int* ⇒ *8 word list*
  where
  encode-s64 x ≡ *flatten-s64* (*word-of-int x*)

abbreviation *decode-s64* :: *8 word list* ⇒ *int*
  where
  decode-s64 x ≡ *sint* (*cat-s64 x*)

lemma *encode-decode-s64*:
assumes $w = [a1, a2, b1, b2, c1, c2, d1, d2]$
shows $w = encode\text{-}s64\ (decode\text{-}s64\ w)$
⟨*proof*⟩

lemma *decode-encode-s64*:
assumes $-(2\ \hat{}\ (LENGTH(64) - 1)) \leq i$
and $i < 2\ \hat{}\ (LENGTH(64) - 1)$
shows $i = decode\text{-}s64\ (encode\text{-}s64\ i)$
⟨*proof*⟩

definition *word-of-integer* :: *integer* ⇒ *'a::len word*
where
word-of-integer x ≡ *word-of-int* (*int-of-integer x*)

definition *sword-of-integer* :: *integer* ⇒ *'a::len sword*
where
sword-of-integer x ≡ *word-of-int* (*int-of-integer x*)

definition *integer-of-word* :: *'a::len word* ⇒ *integer*
where
integer-of-word x ≡ *integer-of-int* (*uint x*)

definition *integer-of-sword* :: *'a::len sword* ⇒ *integer*
where
integer-of-sword x ≡ *integer-of-int* (*sint x*)

lemma *word-integer-eq*:
word-of-integer (*integer-of-word w*) = *w*
⟨*proof*⟩

lemma *sword-integer-eq*:
sword-of-integer (*integer-of-sword w*) = *w*
⟨*proof*⟩

lemma *integer-word-bounded-eq*:
assumes *0* ≤ *i*
assumes *i* ≤ *2* ^ *LENGTH*(*'a::len*) − *1*
shows *integer-of-word* ((*word-of-integer* :: *integer* ⇒ *'a word*) *i*) = *i*
⟨*proof*⟩

lemma *integer-sword-bounded-eq*:
assumes $-(2\ \hat{}\ (LENGTH('a::len) - 1)) \leq i$
and $i < 2\ \hat{}\ (LENGTH('a) - 1)$
shows *integer-of-sword* ((*sword-of-integer* :: *integer* ⇒ *'a sword*) *i*) = *i*
⟨*proof*⟩

definition *word8-of-integer* :: *integer* ⇒ *8 word*
where

word8-of-integer ≡ *word-of-integer*

definition *word16-of-integer* :: *integer* ⇒ *16 word*
where
word16-of-integer ≡ *word-of-integer*

definition *word32-of-integer* :: *integer* ⇒ *32 word*
where
word32-of-integer ≡ *word-of-integer*

definition *word64-of-integer* :: *integer* ⇒ *64 word*
where
word64-of-integer ≡ *word-of-integer*

definition *integer-of-word8* :: *8 word* ⇒ *integer*
where
integer-of-word8 ≡ *integer-of-word*

definition *integer-of-word16* :: *16 word* ⇒ *integer*
where
integer-of-word16 ≡ *integer-of-word*

definition *integer-of-word32* :: *32 word* ⇒ *integer*
where
integer-of-word32 ≡ *integer-of-word*

definition *integer-of-word64* :: *64 word* ⇒ *integer*
where
integer-of-word64 ≡ *integer-of-word*

lemma *word8-integer-eq*:
word8-of-integer (*integer-of-word8 w*) = *w*
⟨*proof*⟩

lemma *word16-integer-eq*:
word16-of-integer (*integer-of-word16 w*) = *w*
⟨*proof*⟩

lemma *word32-integer-eq*:
word32-of-integer (*integer-of-word32 w*) = *w*
⟨*proof*⟩

lemma *word64-integer-eq*:
word64-of-integer (*integer-of-word64 w*) = *w*
⟨*proof*⟩

lemma *integer-word8-bounded-eq*:
assumes *0* ≤ *i*
and *i* ≤ *2* ^ *LENGTH*(*8*) − *1*

shows *integer-of-word8* (*word8-of-integer i*) = *i*
⟨*proof*⟩

lemma *integer-word16-bounded-eq*:
assumes *0* ≤ *i*
and *i* ≤ *2* ^ *LENGTH*(*16*) − *1*
shows *integer-of-word16* (*word16-of-integer i*) = *i*
⟨*proof*⟩

lemma *integer-word32-bounded-eq*:
assumes *0* ≤ *i*
and *i* ≤ *2* ^ *LENGTH*(*32*) − *1*
shows *integer-of-word32* (*word32-of-integer i*) = *i*
⟨*proof*⟩

lemma *integer-word64-bounded-eq*:
assumes *0* ≤ *i*
and *i* ≤ *2* ^ *LENGTH*(*64*) − *1*
shows *integer-of-word64* (*word64-of-integer i*) = *i*
⟨*proof*⟩

definition *sword8-of-integer* :: *integer* ⇒ *8 sword*
where
sword8-of-integer ≡ *sword-of-integer*

definition *sword16-of-integer* :: *integer* ⇒ *16 sword*
where
sword16-of-integer ≡ *sword-of-integer*

definition *sword32-of-integer* :: *integer* ⇒ *32 sword*
where
sword32-of-integer ≡ *sword-of-integer*

definition *sword64-of-integer* :: *integer* ⇒ *64 sword*
where
sword64-of-integer ≡ *sword-of-integer*

definition *integer-of-sword8* :: *8 sword* ⇒ *integer*
where
integer-of-sword8 ≡ *integer-of-sword*

definition *integer-of-sword16* :: *16 sword* ⇒ *integer*
where
integer-of-sword16 ≡ *integer-of-sword*

definition *integer-of-sword32* :: *32 sword* ⇒ *integer*
where
integer-of-sword32 ≡ *integer-of-sword*

definition *integer-of-sword64* :: *64 sword* ⇒ *integer*
where
integer-of-sword64 ≡ *integer-of-sword*

lemma *sword8-integer-eq*:
sword8-of-integer (*integer-of-sword8 w*) = *w*
⟨*proof*⟩

lemma *sword16-integer-eq*:
sword16-of-integer (*integer-of-sword16 w*) = *w*
⟨*proof*⟩

lemma *sword32-integer-eq*:
sword32-of-integer (*integer-of-sword32 w*) = *w*
⟨*proof*⟩

lemma *sword64-integer-eq*:
sword64-of-integer (*integer-of-sword64 w*) = *w*
⟨*proof*⟩

lemma *integer-sword8-bounded-eq*:
assumes − (2 ^ (*LENGTH*(8) − 1)) ≤ *i*
and *i* < 2 ^ (*LENGTH*(8) − 1)
shows *integer-of-sword8* (*sword8-of-integer i*) = *i*
⟨*proof*⟩

lemma *integer-sword16-bounded-eq*:
assumes − (2 ^ (*LENGTH*(16) − 1)) ≤ *i*
and *i* < 2 ^ (*LENGTH*(16) − 1)
shows *integer-of-sword16* (*sword16-of-integer i*) = *i*
⟨*proof*⟩

lemma *integer-sword32-bounded-eq*:
assumes − (2 ^ (*LENGTH*(32) − 1)) ≤ *i*
and *i* < 2 ^ (*LENGTH*(32) − 1)
shows *integer-of-sword32* (*sword32-of-integer i*) = *i*
⟨*proof*⟩

lemma *integer-sword64-bounded-eq*:
assumes − (2 ^ (*LENGTH*(64) − 1)) ≤ *i*
and *i* < 2 ^ (*LENGTH*(64) − 1)
shows *integer-of-sword64* (*sword64-of-integer i*) = *i*
⟨*proof*⟩

lemmas *flatten-types-length* = *flatten-u16-length flatten-s16-length flatten-u32-length flatten-s32-length flatten-u64-length flatten-s64-length*

cast_val is an executable code that ensures easy casting of values. This value cast function is used within the Gillian framework [7, 5, 4, 1].

definition *cast-val* :: *String.literal* ⇒ *integer* ⇒ *integer*
where
cast-val s i ≡
if s = *STR ''uint8''* *then integer-of-word8* (*word8-of-integer i*)
else if s = *STR ''int8''* *then integer-of-sword8* (*sword8-of-integer i*)
else if s = *STR ''uint16''* *then integer-of-word16* (*word16-of-integer i*)
else if s = *STR ''int16''* *then integer-of-sword16* (*sword16-of-integer i*)
else if s = *STR ''uint32''* *then integer-of-word32* (*word32-of-integer i*)
else if s = *STR ''int32''* *then integer-of-sword32* (*sword32-of-integer i*)
else if s = *STR ''uint64''* *then integer-of-word64* (*word64-of-integer i*)
else if s = *STR ''int64''* *then integer-of-sword64* (*sword64-of-integer i*)
else i

end
theory *CHERI-C-Concrete-Memory-Model*
imports *Preliminary-Library*
Separation-Algebra.Separation-Algebra
Containers.Containers
HOL−Library.Mapping
HOL−Library.Code-Target-Numeral
begin

2 CHERI-C Error System

In this section, we formalise the error system used by the memory model.

Below are coprocessor 2 excessptions thrown by the hardware. BadAddressViolation is not a coprocessor 2 exception but remains one given by the hardware. This corresponds to CapErr in the paper [6].

datatype *c2errtype* =
TagViolation
| *PermitLoadViolation*
| *PermitStoreViolation*
| *PermitStoreCapViolation*
| *PermitStoreLocalCapViolation*
| *LengthViolation*
| *BadAddressViolation*

These are logical errors produced by the language. In practice, Some of these errors would never be caught due to the inherent spatial safety guarantees given by capabilities. This corresponds to LogicErr in the paper [6].
NOTE: Unhandled corresponds to a custom error not mentioned in *logicerrtype*. One can provide the custom error as a string, but here, for custom errors, we leave it empty to simplify the proof. Ultimately, the important point is that the memory model can still catch custom errors.

datatype *logicerrtype* =
UseAfterFree

| *BufferOverrun*
| *MissingResource*
| *WrongMemVal*
| *MemoryNotFreed*
| *Unhandled String.literal*

We make the distinction between the error types. This corresponds to Err in the paper [6].

datatype *errtype* =
C2Err c2errtype
| *LogicErr logicerrtype*

Finally, we have the 'return' type $\mathcal{R}\ \rho$ in the paper [6].

datatype *'a result* =
Success (*res*: *'a*)
| *Error* (*err*: *errtype*)

In this theory, we concretise the notion of blocks

type-synonym *block* = *integer*
type-synonym *memcap* = *block mem-capability*
type-synonym *cap* = *block capability*

Because `sizeof` depends on the architecture, it shall be given via the memory model. We also use uncompressed capabilities.

definition *sizeof* :: *cctype* ⇒ *nat* (‹|-|$_\tau$›)
where
sizeof $\tau \equiv$ *case* τ *of*
Uint8 ⇒ *1*
| *Sint8* ⇒ *1*
| *Uint16* ⇒ *2*
| *Sint16* ⇒ *2*
| *Uint32* ⇒ *4*
| *Sint32* ⇒ *4*
| *Uint64* ⇒ *8*
| *Sint64* ⇒ *8*
| *Cap* ⇒ *32*

We provide some helper lemmas

lemma *size-type-align*:
assumes $|t|_\tau = x$
shows ∃ *n*. *2* ^ *n* = *x*
⟨*proof*⟩

lemma *memval-size-u8*:
$|$*memval-type* (*Uint8-v v*)$|_\tau$ = *1*
⟨*proof*⟩

lemma *memval-size-s8*:

$|memval\text{-}type\ (Sint8\text{-}v\ v)|_\tau = 1$
⟨*proof*⟩

lemma *memval-size-u16*:
$|memval\text{-}type\ (Uint16\text{-}v\ v)|_\tau = 2$
⟨*proof*⟩

lemma *memval-size-s16*:
$|memval\text{-}type\ (Sint16\text{-}v\ v)|_\tau = 2$
⟨*proof*⟩

lemma *memval-size-u32*:
$|memval\text{-}type\ (Uint32\text{-}v\ v)|_\tau = 4$
⟨*proof*⟩

lemma *memval-size-s32*:
$|memval\text{-}type\ (Sint32\text{-}v\ v)|_\tau = 4$
⟨*proof*⟩

lemma *memval-size-u64*:
$|memval\text{-}type\ (Uint64\text{-}v\ v)|_\tau = 8$
⟨*proof*⟩

lemma *memval-size-s64*:
$|memval\text{-}type\ (Sint64\text{-}v\ v)|_\tau = 8$
⟨*proof*⟩

lemma *memval-size-cap*:
$|memval\text{-}type\ (Cap\text{-}v\ v)|_\tau = 32$
⟨*proof*⟩

lemmas *memval-size-types* = *memval-size-u8 memval-size-s8 memval-size-u16 memval-size-s16 memval-size-u32 memval-size-s32 memval-size-u64 memval-size-s64 memval-size-cap*

corollary *memval-size-u16-eq-word-split-len*:
assumes $val = Uint16\text{-}v\ v$
shows $|memval\text{-}type\ val|_\tau = length\ (u16\text{-}split\ v)$
⟨*proof*⟩

corollary *memval-size-s16-eq-word-split-len*:
assumes $val = Sint16\text{-}v\ v$
shows $|memval\text{-}type\ val|_\tau = length\ (s16\text{-}split\ v)$
⟨*proof*⟩

corollary *memval-size-u32-eq-word-split-len*:
assumes $val = Uint32\text{-}v\ v$
shows $|memval\text{-}type\ val|_\tau = length\ (flatten\text{-}u32\ v)$
⟨*proof*⟩

corollary *memval-size-s32-eq-word-split-len*:
assumes *val* = *Sint32-v v*
shows $|$*memval-type val*$|_\tau$ = *length* (*flatten-s32 v*)
⟨*proof*⟩

corollary *memval-size-u64-eq-word-split-len*:
assumes *val* = *Uint64-v v*
shows $|$*memval-type val*$|_\tau$ = *length* (*flatten-u64 v*)
⟨*proof*⟩

corollary *memval-size-s64-eq-word-split-len*:
assumes *val* = *Sint64-v v*
shows $|$*memval-type val*$|_\tau$ = *length* (*flatten-s64 v*)
⟨*proof*⟩

lemma *sizeof-nonzero*:
$|t|_\tau > 0$
⟨*proof*⟩

We prove that integer is a countable type.

instance *int* :: *comp-countable* ⟨*proof*⟩

lemma *integer-encode-eq*: (*int-encode* ∘ *int-of-integer*) *x* = (*int-encode* ∘ *int-of-integer*) *y* ⟷ *x* = *y*
⟨*proof*⟩

instance *integer* :: *countable*
⟨*proof*⟩

instance *integer* :: *comp-countable* ⟨*proof*⟩

3 Memory

In this section, we formalise the heap and prove some initial properties.

3.1 Definitions

First, we provide $\mathcal{V}_\mathcal{M}$—refer to [6] for the definition. We note that this representation allows us to make the distinction between what is a capability and what is a primitive value stored in memory. We can define a tag-preserving `memcpy` by checking ahead whether there are valid capabilities stored in memory or whether there are simply bytes. The downside to this approach is that overwriting primitive values to where capabilities were stored—and vice versa—will lead to an undefined load operation. However, this tends not to be a big problem, as (1) overwritten capabilities are tag-invalidated anyway, so the capabilities cannot be dereferenced even if the

user obtained the capability somehow, and (2) for legacy C programs that do not have access to CHERI library functions, there is no way to access the metadata of the invalidated capabilities. For compatibility purposes, this imposes hardly any problems.

datatype *memval* =
  Byte (*of-byte*: *8 word*)
  | *ACap* (*of-cap*: *memcap*) (*of-nth*: *nat*)

In general, the bound is irrelevant, as capability bound ensures spatial safety. We add bounds in the heap so that we can incorporate *hybrid* CHERI C programs in the future, where pointers and capabilities co-exist, but strictly speaking, this is not required in *purecap* CHERI C programs, which is what this memory model is based on. Ultimately, this is the pair of mapping defined in the paper [6].

record *object* =
  bounds :: *nat* × *nat*
  content :: (*nat*, *memval*) *mapping*
  tags :: (*nat*, *bool*) *mapping*

t is the datatype that allows us to make the distinction between blocks that are freed and blocks that are valid.

datatype *t* =
  Freed
  | *Map* (*the-map*: *object*)

heap_map in heap is essentially $\mathcal{H}$ defined in the paper [6]. We extend the structure and keep track of the next block for the allocator for efficiency—much like how CompCert's C memory model does this [3].

record *heap* =
  next-block :: *block*
  heap-map :: (*block*, *t*) *mapping*

definition *memval-is-byte* :: *memval* ⇒ *bool*
  where
  memval-is-byte m ≡ *case m of Byte* - ⇒ *True* | *ACap* - - ⇒ *False*

abbreviation *memval-is-cap* :: *memval* ⇒ *bool*
  where
  memval-is-cap m ≡ ¬ *memval-is-byte m*

lemma *memval-byte*:
  memval-is-byte m ⟹ ∃ *b*. *m* = *Byte b*
  ⟨*proof*⟩

lemma *memval-byte-not-memcap*:
  memval-is-byte m ⟹ *m* ≠ *ACap c n*
  ⟨*proof*⟩

lemma *memval-memcap*:
memval-is-cap m $\Longrightarrow \exists$ *c n*. *m* = *ACap c n*
⟨*proof*⟩

lemma *memval-memcap-not-byte*:
memval-is-cap m $\Longrightarrow$ *m* $\neq$ *Byte b*
⟨*proof*⟩

3.2 Properties

We prove that the heap is an instance of separation algebra.

instantiation *unit* :: *cancellative-sep-algebra*
begin
definition *0* ≡ ()
definition *u1* + *u2* = ()
definition (*u1*::*unit*) ## *u2* ≡ *True*
instance
⟨*proof*⟩
end

instantiation *nat* :: *cancellative-sep-algebra*
begin
definition (*n1*::*nat*) ## *n2* ≡ *True*
instance
⟨*proof*⟩
end

This proof ultimately shows that heap_map forms a separation algebra.

instantiation *mapping* :: (*type*, *type*) *cancellative-sep-algebra*
begin

definition *zero-map-def*: *0* ≡ *Mapping.empty*
definition *plus-map-def*: *m1* + *m2* ≡ *Mapping* ((*Mapping.lookup m1*) ++ (*Mapping.lookup m2*))
definition *sep-disj-map-def*: *m1* ## *m2* ≡ *Mapping.keys m1* ∩ *Mapping.keys m2* = {}

instance
⟨*proof*⟩
end

instantiation *heap-ext* :: (*cancellative-sep-algebra*) *cancellative-sep-algebra*
begin
definition *0* :: *'a heap-scheme* ≡ (| *next-block* = *0*, *heap-map* = *Mapping.empty*, ... = *0* |)
definition (*m1* :: *'a heap-scheme*) + (*m2* :: *'a heap-scheme*) ≡
(| *next-block* = *next-block m1* + *next-block m2*,

```
                heap-map = Mapping ((Mapping.lookup (heap-map m1)) ++
(Mapping.lookup (heap-map m2))),
             ... = heap.more m1 + heap.more m2 ⦈
definition (m1 :: 'a heap-scheme) ## (m2 :: 'a heap-scheme) ≡
           Mapping.keys (heap-map m1) ∩ Mapping.keys (heap-map m2) = {}
           ∧ heap.more m1 ## heap.more m2
instance
⟨proof⟩
end

instantiation mem-capability-ext :: (comp-countable, zero) zero
begin
definition 0 :: ('a, 'b) mem-capability-scheme ≡
        ⦇ block-id = 0,
          offset = 0,
          base = 0,
          len = 0,
          perm-load = False,
          perm-cap-load = False,
          perm-store = False,
          perm-cap-store = False,
          perm-cap-store-local = False,
          perm-global = False,
          ... =  0⦈
instance ⟨proof⟩
end

subclass (in comp-countable) zero ⟨proof⟩

instantiation capability-ext :: (zero) zero
begin
definition 0 ≡ ⦇ tag = False, ... = 0⦈
instance ⟨proof⟩
end
```

Section 4.5 of CHERI C/C++ Programming Guide defines what a `NULL` capability is [8].

```
definition null-capability :: cap (‹NULL›)
  where
  NULL ≡ 0

context
  notes null-capability-def[simp]
begin

lemma null-capability-block-id[simp]:
  block-id NULL = 0
  ⟨proof⟩
```

lemma *null-capability-offset*[*simp*]:
offset NULL = 0
⟨*proof*⟩

lemma *null-capability-base*[*simp*]:
base NULL = 0
⟨*proof*⟩

lemma *null-capability-len*[*simp*]:
len NULL = 0
⟨*proof*⟩

lemma *null-capability-perm-load*[*simp*]:
perm-load NULL = False
⟨*proof*⟩

lemma *null-capability-perm-cap-load*[*simp*]:
perm-cap-load NULL = False
⟨*proof*⟩

lemma *null-capability-perm-store*[*simp*]:
perm-store NULL = False
⟨*proof*⟩

lemma *null-capability-perm-cap-store*[*simp*]:
perm-cap-store NULL = False
⟨*proof*⟩

lemma *null-capability-perm-cap-store-local*[*simp*]:
perm-cap-store-local NULL = False
⟨*proof*⟩

lemma *null-capability-tag*[*simp*]:
tag NULL = False
⟨*proof*⟩

end

Here, we define the initial heap.

definition *init-heap* :: *heap*
where
init-heap ≡ *0* (| *next-block* := *1* |)

abbreviation *cap-offset* :: *nat* ⇒ *nat*
where
cap-offset p ≡ *if p mod* $|Cap|_\tau$ *= 0 then p else p − p mod* $|Cap|_\tau$

We state the well-formedness property $\mathcal{W}_f^{\mathcal{C}}$ stated in the paper [6].

definition *wellformed* :: (*block*, *t*) *mapping* ⇒ *bool* (‹$\mathcal{W}_{\mathfrak{f}}$/(-/)›)

where
$\mathcal{W}_{\mathfrak{f}}(h) \equiv$
$\forall\ b\ obj.\ Mapping.lookup\ h\ b = Some\ (Map\ obj)$
$\longrightarrow Set.filter\ (\lambda x.\ x\ mod\ |Cap|_\tau \neq 0)\ (Mapping.keys\ (tags\ obj)) = \{\}$

lemma *init-heap-empty*:
$Mapping.keys\ (heap\text{-}map\ init\text{-}heap) = \{\}$
⟨*proof*⟩

Below shows $\mathcal{W}_f^{\mathcal{C}}(\mu_0)$

lemma *init-wellformed*:
$\mathcal{W}_{\mathfrak{f}}(heap\text{-}map\ init\text{-}heap)$
⟨*proof*⟩

lemma *mapping-lookup-disj1*:
$m1\ \#\#\ m2 \Longrightarrow Mapping.lookup\ m1\ n = Some\ x \Longrightarrow Mapping.lookup\ (m1 + m2)\ n = Some\ x$
⟨*proof*⟩

lemma *mapping-lookup-disj2*:
$m1\ \#\#\ m2 \Longrightarrow Mapping.lookup\ m2\ n = Some\ x \Longrightarrow Mapping.lookup\ (m1 + m2)\ n = Some\ x$
⟨*proof*⟩

Below shows that well-formedness is composition-compatible

lemma $heap\text{-}map\ h1\ \#\#\ heap\text{-}map\ h2 \Longrightarrow \mathcal{W}_{\mathfrak{f}}(heap\text{-}map\ h1 + heap\text{-}map\ h2)$
$\Longrightarrow \mathcal{W}_{\mathfrak{f}}(heap\text{-}map\ h1) \wedge \mathcal{W}_{\mathfrak{f}}(heap\text{-}map\ h2)$
⟨*proof*⟩

4 Helper functions and lemmas

primrec $is\text{-}memval\text{-}defined :: (nat, memval)\ mapping \Rightarrow nat \Rightarrow nat \Rightarrow bool$
where
$is\text{-}memval\text{-}defined\ \text{-}\ \text{-}\ 0 = True$
$|\ is\text{-}memval\text{-}defined\ m\ off\ (Suc\ siz) = ((off \in Mapping.keys\ m) \wedge is\text{-}memval\text{-}defined\ m\ (Suc\ off)\ siz)$

primrec $is\text{-}contiguous\text{-}bytes :: (nat, memval)\ mapping \Rightarrow nat \Rightarrow nat \Rightarrow bool$
where
$is\text{-}contiguous\text{-}bytes\ \text{-}\ \text{-}\ 0 = True$
$|\ is\text{-}contiguous\text{-}bytes\ m\ off\ (Suc\ siz) = ((off \in Mapping.keys\ m)$
$\wedge\ memval\text{-}is\text{-}byte\ (the\ (Mapping.lookup\ m\ off))$
$\wedge\ is\text{-}contiguous\text{-}bytes\ m\ (Suc\ off)\ siz)$

definition $get\text{-}cap :: (nat, memval)\ mapping \Rightarrow nat \Rightarrow memcap$
where
$get\text{-}cap\ m\ off = of\text{-}cap\ (the\ (Mapping.lookup\ m\ off))$

fun *is-cap* :: (*nat*, *memval*) *mapping* ⇒ *nat* ⇒ *bool*
  where
  is-cap m off = (*off* ∈ *Mapping.keys m* ∧ *memval-is-cap* (*the* (*Mapping.lookup m off*)))

primrec *is-contiguous-cap* :: (*nat*, *memval*) *mapping* ⇒ *memcap* ⇒ *nat* ⇒ *nat* ⇒ *bool*
  where
  is-contiguous-cap - - - 0 = *True*
| *is-contiguous-cap m c off* (*Suc siz*) = ((*off* ∈ *Mapping.keys m*)
∧ *memval-is-cap* (*the* (*Mapping.lookup m off*))
∧ *of-cap* (*the* (*Mapping.lookup m off*)) = *c*
∧ *of-nth* (*the* (*Mapping.lookup m off*)) = *siz*
∧ *is-contiguous-cap m c* (*Suc off*) *siz*)

primrec *is-contiguous-zeros-prim* :: (*nat*, *memval*) *mapping* ⇒ *nat* ⇒ *nat* ⇒ *bool*
  where
  is-contiguous-zeros-prim - - 0 = *True*
| *is-contiguous-zeros-prim m off* (*Suc siz*) = (*Mapping.lookup m off* = *Some* (*Byte 0*)
∧ *is-contiguous-zeros-prim m* (*Suc off*) *siz*)

definition *is-contiguous-zeros* :: (*nat*, *memval*) *mapping* ⇒ *nat* ⇒ *nat* ⇒ *bool*
  where
  is-contiguous-zeros m off siz ≡ ∀ *ofs* ≥ *off*. *ofs* < *off* + *siz* ⟶ *Mapping.lookup m ofs* = *Some* (*Byte 0*)

lemma *is-contiguous-zeros-code*[*code*]:
  is-contiguous-zeros m off siz = *is-contiguous-zeros-prim m off siz*
⟨*proof*⟩

primrec *retrieve-bytes* :: (*nat*, *memval*) *mapping* ⇒ *nat* ⇒ *nat* ⇒ *8 word list*
  where
  retrieve-bytes m - 0 = []
| *retrieve-bytes m off* (*Suc siz*) = *of-byte* (*the* (*Mapping.lookup m off*)) # *retrieve-bytes m* (*Suc off*) *siz*

primrec *is-same-cap* :: (*nat*, *memval*) *mapping* ⇒ *memcap* ⇒ *nat* ⇒ *nat* ⇒ *bool*
  where
  is-same-cap - - - 0 = *True*
| *is-same-cap m c off* (*Suc siz*) = (*of-cap* (*the* (*Mapping.lookup m off*)) = *c* ∧ *is-same-cap m c* (*Suc off*) *siz*)

definition *retrieve-tval* :: *object* ⇒ *nat* ⇒ *cctype* ⇒ *bool* ⇒ *block ccval*
  where
  retrieve-tval obj off typ pcl ≡

```
    if is-contiguous-bytes (content obj) off |typ|_τ then
      (case typ of
         Uint8  ⇒ Uint8-v  (decode-u8-list (retrieve-bytes (content obj) off |typ|_τ))
       | Sint8  ⇒ Sint8-v  (decode-s8-list (retrieve-bytes (content obj) off |typ|_τ))
       | Uint16 ⇒ Uint16-v (cat-u16 (retrieve-bytes (content obj) off |typ|_τ))
       | Sint16 ⇒ Sint16-v (cat-s16 (retrieve-bytes (content obj) off |typ|_τ))
       | Uint32 ⇒ Uint32-v (cat-u32 (retrieve-bytes (content obj) off |typ|_τ))
       | Sint32 ⇒ Sint32-v (cat-s32 (retrieve-bytes (content obj) off |typ|_τ))
       | Uint64 ⇒ Uint64-v (cat-u64 (retrieve-bytes (content obj) off |typ|_τ))
       | Sint64 ⇒ Sint64-v (cat-s64 (retrieve-bytes (content obj) off |typ|_τ))
       | Cap    ⇒ if is-contiguous-zeros (content obj) off |typ|_τ then Cap-v NULL
else Undef)
    else if is-cap (content obj) off then
      let cap = get-cap (content obj) off in
      let tv = the (Mapping.lookup (tags obj) (cap-offset off)) in
      let t = (case pcl of False ⇒ False | True ⇒ tv) in
      let cv = mem-capability.extend cap (| tag = t |) in
      let nth-frag = of-nth (the (Mapping.lookup (content obj) off)) in
      (case typ of
         Uint8 ⇒ Cap-v-frag (mem-capability.extend cap (| tag = False |)) nth-frag
       | Sint8 ⇒ Cap-v-frag (mem-capability.extend cap (| tag = False |)) nth-frag
       | Cap   ⇒ if is-contiguous-cap (content obj) cap off |typ|_τ then Cap-v cv else
Undef
       | -     ⇒ Undef)
    else Undef

primrec store-bytes :: (nat, memval) mapping ⇒ nat ⇒ 8 word list ⇒ (nat, mem-
val) mapping
  where
  store-bytes obj - [] = obj
| store-bytes obj off (v # vs) = store-bytes (Mapping.update off (Byte v) obj) (Suc
off) vs

primrec store-cap :: (nat, memval) mapping ⇒ nat ⇒ cap ⇒ nat ⇒ (nat, memval)
mapping
  where
  store-cap obj - - 0 = obj
| store-cap obj off cap (Suc n) = store-cap (Mapping.update off (ACap (mem-capability.truncate
cap) n) obj) (Suc off) cap n

abbreviation store-tag :: (nat, bool) mapping ⇒ nat ⇒ bool ⇒ (nat, bool) map-
ping
  where
  store-tag obj off tg ≡ Mapping.update off tg obj

definition store-tval :: object ⇒ nat ⇒ block ccval ⇒ object
  where
  store-tval obj off val ≡
     case val of Uint8-v  v      ⇒ obj (| content := store-bytes (content obj) off
```

(*encode-u8-list v*),
tags := *store-tag* (*tags obj*) (*cap-offset off*) *False* ⦈
| *Sint8-v v* ⇒ *obj* (⦇ *content* := *store-bytes* (*content obj*) *off* (*encode-s8-list v*),
tags := *store-tag* (*tags obj*) (*cap-offset off*) *False* ⦈
| *Uint16-v v* ⇒ *obj* (⦇ *content* := *store-bytes* (*content obj*) *off* (*u16-split v*),
tags := *store-tag* (*tags obj*) (*cap-offset off*) *False* ⦈
| *Sint16-v v* ⇒ *obj* (⦇ *content* := *store-bytes* (*content obj*) *off* (*s16-split v*),
tags := *store-tag* (*tags obj*) (*cap-offset off*) *False* ⦈
| *Uint32-v v* ⇒ *obj* (⦇ *content* := *store-bytes* (*content obj*) *off* (*flatten-u32 v*),
tags := *store-tag* (*tags obj*) (*cap-offset off*) *False* ⦈
| *Sint32-v v* ⇒ *obj* (⦇ *content* := *store-bytes* (*content obj*) *off* (*flatten-s32 v*),
tags := *store-tag* (*tags obj*) (*cap-offset off*) *False* ⦈
| *Uint64-v v* ⇒ *obj* (⦇ *content* := *store-bytes* (*content obj*) *off* (*flatten-u64 v*),
tags := *store-tag* (*tags obj*) (*cap-offset off*) *False* ⦈
| *Sint64-v v* ⇒ *obj* (⦇ *content* := *store-bytes* (*content obj*) *off* (*flatten-s64 v*),
tags := *store-tag* (*tags obj*) (*cap-offset off*) *False* ⦈
| *Cap-v c* ⇒ *obj* (⦇ *content* := *store-cap* (*content obj*) *off c* $|Cap|_\tau$,
tags := *store-tag* (*tags obj*) (*cap-offset off*) (*tag c*) ⦈
| *Cap-v-frag c n* ⇒ *obj* (⦇ *content* := *Mapping.update off* (*ACap* (*mem-capability.truncate c*) *n*) (*content obj*),
tags := *store-tag* (*tags obj*) (*cap-offset off*) *False*⦈

lemma *stored-bytes-prev*:
assumes $x < off$
shows *Mapping.lookup* (*store-bytes obj off vs*) *x* = *Mapping.lookup obj x*
⟨*proof*⟩

lemma *stored-tags-prev*:
assumes $x < off$
shows *Mapping.lookup* (*store-tag obj off vs*) *x* = *Mapping.lookup obj x*
⟨*proof*⟩

lemma *stored-cap-prev*:
assumes $x < off$
shows *Mapping.lookup* (*store-cap obj off cap siz*) *x* = *Mapping.lookup obj x*
⟨*proof*⟩

lemma *stored-bytes-instant-correctness*:
Mapping.lookup (*store-bytes obj off* (*v* # *vs*)) *off* = *Some* (*Byte v*)
⟨*proof*⟩

lemma *stored-cap-instant-correctness*:

Mapping.lookup (*store-cap obj off cap* (*Suc siz*)) *off* = *Some* (*ACap* (*mem-capability.truncate cap*) *siz*)
⟨*proof*⟩

lemma *numeral-4-eq-4*: *4* = *Suc* (*Suc* (*Suc* (*Suc 0*)))
⟨*proof*⟩

lemma *numeral-5-eq-5*: *5* = *Suc* (*Suc* (*Suc* (*Suc* (*Suc 0*))))
⟨*proof*⟩

lemma *numeral-6-eq-6*: *6* = *Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc 0*)))))
⟨*proof*⟩

lemma *numeral-7-eq-7*: *7* = *Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc 0*))))))
⟨*proof*⟩

lemma *numeral-8-eq-8*: *8* = *Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc 0*)))))))
⟨*proof*⟩

lemma *list-length-2-realise*:
length ls = *2* ⟹ ∃ *n0 n1*. *ls* = [*n0*, *n1*]
⟨*proof*⟩

lemma *list-length-4-realise*:
length ls = *4* ⟹ ∃ *n0 n1 n2 n3*. *ls* = [*n0*, *n1*, *n2*, *n3*]
⟨*proof*⟩

lemma *list-length-8-realise*:
length ls = *8* ⟹ ∃ *n0 n1 n2 n3 n4 n5 n6 n7*. *ls* = [*n0*, *n1*, *n2*, *n3*, *n4*, *n5*, *n6*, *n7*]
⟨*proof*⟩

lemma *u16-split-realise*:
∃ *b0 b1*. *u16-split v* = [*b0*, *b1*]
⟨*proof*⟩

lemma *s16-split-realise*:
∃ *b0 b1*. *s16-split v* = [*b0*, *b1*]
⟨*proof*⟩

lemma *u32-split-realise*:
∃ *b0 b1 b2 b3*. *flatten-u32 v* = [*b0*, *b1*, *b2*, *b3*]
⟨*proof*⟩

lemma *s32-split-realise*:
∃ *b0 b1 b2 b3*. *flatten-s32 v* = [*b0*, *b1*, *b2*, *b3*]
⟨*proof*⟩

lemma *u64-split-realise*:

∃ *b0 b1 b2 b3 b4 b5 b6 b7*. *flatten-u64 v* = [*b0*, *b1*, *b2*, *b3*, *b4*, *b5*, *b6*, *b7*]
⟨*proof*⟩

lemma *s64-split-realise*:
∃ *b0 b1 b2 b3 b4 b5 b6 b7*. *flatten-s64 v* = [*b0*, *b1*, *b2*, *b3*, *b4*, *b5*, *b6*, *b7*]
⟨*proof*⟩

lemma *store-bytes-u16*:
shows *off* ∈ *Mapping.keys* (*store-bytes m off* (*u16-split v*))
and *Suc off* ∈ *Mapping.keys* (*store-bytes m off* (*u16-split v*))
and ∃ *b0*. *Mapping.lookup* (*store-bytes m off* (*u16-split v*)) *off* = *Some* (*Byte b0*)
and ∃ *b1*. *Mapping.lookup* (*store-bytes m off* (*u16-split v*)) (*Suc off*) = *Some* (*Byte b1*)
⟨*proof*⟩

lemma *store-bytes-s16*:
shows *off* ∈ *Mapping.keys* (*store-bytes m off* (*s16-split v*))
and *Suc off* ∈ *Mapping.keys* (*store-bytes m off* (*s16-split v*))
and ∃ *b0*. *Mapping.lookup* (*store-bytes m off* (*s16-split v*)) *off* = *Some* (*Byte b0*)
and ∃ *b1*. *Mapping.lookup* (*store-bytes m off* (*s16-split v*)) (*Suc off*) = *Some* (*Byte b1*)
⟨*proof*⟩

lemma *store-bytes-u32*:
shows *off* ∈ *Mapping.keys* (*store-bytes m off* (*flatten-u32 v*))
and *Suc off* ∈ *Mapping.keys* (*store-bytes m off* (*flatten-u32 v*))
and *Suc* (*Suc off*) ∈ *Mapping.keys* (*store-bytes m off* (*flatten-u32 v*))
and *Suc* (*Suc* (*Suc off*)) ∈ *Mapping.keys* (*store-bytes m off* (*flatten-u32 v*))
and ∃ *b0*. *Mapping.lookup* (*store-bytes m off* (*flatten-u32 v*)) *off* = *Some* (*Byte b0*)
and ∃ *b1*. *Mapping.lookup* (*store-bytes m off* (*flatten-u32 v*)) (*Suc off*) = *Some* (*Byte b1*)
and ∃ *b2*. *Mapping.lookup* (*store-bytes m off* (*flatten-u32 v*)) (*Suc* (*Suc off*)) = *Some* (*Byte b2*)
and ∃ *b3*. *Mapping.lookup* (*store-bytes m off* (*flatten-u32 v*)) (*Suc* (*Suc* (*Suc off*))) = *Some* (*Byte b3*)
⟨*proof*⟩

lemma *store-bytes-s32*:
shows *off* ∈ *Mapping.keys* (*store-bytes m off* (*flatten-s32 v*))
and *Suc off* ∈ *Mapping.keys* (*store-bytes m off* (*flatten-s32 v*))
and *Suc* (*Suc off*) ∈ *Mapping.keys* (*store-bytes m off* (*flatten-s32 v*))
and *Suc* (*Suc* (*Suc off*)) ∈ *Mapping.keys* (*store-bytes m off* (*flatten-s32 v*))
and ∃ *b0*. *Mapping.lookup* (*store-bytes m off* (*flatten-s32 v*)) *off* = *Some* (*Byte b0*)
and ∃ *b1*. *Mapping.lookup* (*store-bytes m off* (*flatten-s32 v*)) (*Suc off*) = *Some* (*Byte b1*)

and ∃ *b2*. *Mapping.lookup* (*store-bytes m off* (*flatten-s32 v*)) (*Suc* (*Suc off*)) = *Some* (*Byte b2*)
and ∃ *b3*. *Mapping.lookup* (*store-bytes m off* (*flatten-s32 v*)) (*Suc* (*Suc* (*Suc off*))) = *Some* (*Byte b3*)
⟨*proof*⟩

lemma *store-bytes-u64*:
shows *off* ∈ *Mapping.keys* (*store-bytes m off* (*flatten-u64 v*))
and *Suc off* ∈ *Mapping.keys* (*store-bytes m off* (*flatten-u64 v*))
and *Suc* (*Suc off*) ∈ *Mapping.keys* (*store-bytes m off* (*flatten-u64 v*))
and *Suc* (*Suc* (*Suc off*)) ∈ *Mapping.keys* (*store-bytes m off* (*flatten-u64 v*))
and *Suc* (*Suc* (*Suc* (*Suc off*))) ∈ *Mapping.keys* (*store-bytes m off* (*flatten-u64 v*))
and *Suc* (*Suc* (*Suc* (*Suc* (*Suc off*)))) ∈ *Mapping.keys* (*store-bytes m off* (*flatten-u64 v*))
and *Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc off*))))) ∈ *Mapping.keys* (*store-bytes m off* (*flatten-u64 v*))
and *Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc off*)))))) ∈ *Mapping.keys* (*store-bytes m off* (*flatten-u64 v*))
and ∃ *b0*. *Mapping.lookup* (*store-bytes m off* (*flatten-u64 v*)) *off* = *Some* (*Byte b0*)
and ∃ *b1*. *Mapping.lookup* (*store-bytes m off* (*flatten-u64 v*)) (*Suc off*) = *Some* (*Byte b1*)
and ∃ *b2*. *Mapping.lookup* (*store-bytes m off* (*flatten-u64 v*)) (*Suc* (*Suc off*)) = *Some* (*Byte b2*)
and ∃ *b3*. *Mapping.lookup* (*store-bytes m off* (*flatten-u64 v*)) (*Suc* (*Suc* (*Suc off*))) = *Some* (*Byte b3*)
and ∃ *b0*. *Mapping.lookup* (*store-bytes m off* (*flatten-u64 v*)) (*Suc* (*Suc* (*Suc* (*Suc off*)))) = *Some* (*Byte b0*)
and ∃ *b1*. *Mapping.lookup* (*store-bytes m off* (*flatten-u64 v*)) (*Suc* (*Suc* (*Suc* (*Suc* (*Suc off*))))) = *Some* (*Byte b1*)
and ∃ *b2*. *Mapping.lookup* (*store-bytes m off* (*flatten-u64 v*)) (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc off*)))))) = *Some* (*Byte b2*)
and ∃ *b3*. *Mapping.lookup* (*store-bytes m off* (*flatten-u64 v*)) (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc off*))))))) = *Some* (*Byte b3*)
⟨*proof*⟩

lemma *store-bytes-s64*:
shows *off* ∈ *Mapping.keys* (*store-bytes m off* (*flatten-s64 v*))
and *Suc off* ∈ *Mapping.keys* (*store-bytes m off* (*flatten-s64 v*))
and *Suc* (*Suc off*) ∈ *Mapping.keys* (*store-bytes m off* (*flatten-s64 v*))
and *Suc* (*Suc* (*Suc off*)) ∈ *Mapping.keys* (*store-bytes m off* (*flatten-s64 v*))
and *Suc* (*Suc* (*Suc* (*Suc off*))) ∈ *Mapping.keys* (*store-bytes m off* (*flatten-s64 v*))
and *Suc* (*Suc* (*Suc* (*Suc* (*Suc off*)))) ∈ *Mapping.keys* (*store-bytes m off* (*flatten-s64 v*))
and *Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc off*))))) ∈ *Mapping.keys* (*store-bytes m off* (*flatten-s64 v*))
and *Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc off*)))))) ∈ *Mapping.keys* (*store-bytes*

m off (*flatten-s64 v*))
and ∃ *b0*. *Mapping.lookup* (*store-bytes m off* (*flatten-s64 v*)) *off* = *Some* (*Byte b0*)
and ∃ *b1*. *Mapping.lookup* (*store-bytes m off* (*flatten-s64 v*)) (*Suc off*) = *Some* (*Byte b1*)
and ∃ *b2*. *Mapping.lookup* (*store-bytes m off* (*flatten-s64 v*)) (*Suc* (*Suc off*)) = *Some* (*Byte b2*)
and ∃ *b3*. *Mapping.lookup* (*store-bytes m off* (*flatten-s64 v*)) (*Suc* (*Suc* (*Suc off*))) = *Some* (*Byte b3*)
and ∃ *b0*. *Mapping.lookup* (*store-bytes m off* (*flatten-s64 v*)) (*Suc* (*Suc* (*Suc* (*Suc off*)))) = *Some* (*Byte b0*)
and ∃ *b1*. *Mapping.lookup* (*store-bytes m off* (*flatten-s64 v*)) (*Suc* (*Suc* (*Suc* (*Suc* (*Suc off*))))) = *Some* (*Byte b1*)
and ∃ *b2*. *Mapping.lookup* (*store-bytes m off* (*flatten-s64 v*)) (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc off*)))))) = *Some* (*Byte b2*)
and ∃ *b3*. *Mapping.lookup* (*store-bytes m off* (*flatten-s64 v*)) (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc off*))))))) = *Some* (*Byte b3*)
⟨*proof*⟩

corollary *u16-store-bytes-imp-is-contiguous-bytes*:
is-contiguous-bytes (*store-bytes m off* (*u16-split v*)) *off 2*
⟨*proof*⟩

corollary *s16-store-bytes-imp-is-contiguous-bytes*:
is-contiguous-bytes (*store-bytes m off* (*s16-split v*)) *off 2*
⟨*proof*⟩

corollary *u32-store-bytes-imp-is-contiguous-bytes*:
is-contiguous-bytes (*store-bytes m off* (*flatten-u32 v*)) *off 4*
⟨*proof*⟩

corollary *s32-store-bytes-imp-is-contiguous-bytes*:
is-contiguous-bytes (*store-bytes m off* (*flatten-s32 v*)) *off 4*
⟨*proof*⟩

corollary *u64-store-bytes-imp-is-contiguous-bytes*:
is-contiguous-bytes (*store-bytes m off* (*flatten-u64 v*)) *off 8*
⟨*proof*⟩

corollary *s64-store-bytes-imp-is-contiguous-bytes*:
is-contiguous-bytes (*store-bytes m off* (*flatten-s64 v*)) *off 8*
⟨*proof*⟩

lemma *stored-tval-contiguous-bytes*:
assumes *val* ≠ *Undef*
and ∀ *v*. *val* ≠ *Cap-v v*
and ∀ *v n*. *val* ≠ *Cap-v-frag v n*
shows *is-contiguous-bytes* (*content* (*store-tval obj off val*)) *off* $|memval\text{-}type\ val|_\tau$
⟨*proof*⟩

lemma *suc-of-32*:
32 = Suc 31
⟨*proof*⟩

lemma *store-cap-correct-dom*:
shows *off* ∈ *Mapping.keys* (*store-cap m off cap 32*)
and *off + 1* ∈ *Mapping.keys* (*store-cap m off cap 32*)
and *off + 2* ∈ *Mapping.keys* (*store-cap m off cap 32*)
and *off + 3* ∈ *Mapping.keys* (*store-cap m off cap 32*)
and *off + 4* ∈ *Mapping.keys* (*store-cap m off cap 32*)
and *off + 5* ∈ *Mapping.keys* (*store-cap m off cap 32*)
and *off + 6* ∈ *Mapping.keys* (*store-cap m off cap 32*)
and *off + 7* ∈ *Mapping.keys* (*store-cap m off cap 32*)
and *off + 8* ∈ *Mapping.keys* (*store-cap m off cap 32*)
and *off + 9* ∈ *Mapping.keys* (*store-cap m off cap 32*)
and *off + 10* ∈ *Mapping.keys* (*store-cap m off cap 32*)
and *off + 11* ∈ *Mapping.keys* (*store-cap m off cap 32*)
and *off + 12* ∈ *Mapping.keys* (*store-cap m off cap 32*)
and *off + 13* ∈ *Mapping.keys* (*store-cap m off cap 32*)
and *off + 14* ∈ *Mapping.keys* (*store-cap m off cap 32*)
and *off + 15* ∈ *Mapping.keys* (*store-cap m off cap 32*)
and *off + 16* ∈ *Mapping.keys* (*store-cap m off cap 32*)
and *off + 17* ∈ *Mapping.keys* (*store-cap m off cap 32*)
and *off + 18* ∈ *Mapping.keys* (*store-cap m off cap 32*)
and *off + 19* ∈ *Mapping.keys* (*store-cap m off cap 32*)
and *off + 20* ∈ *Mapping.keys* (*store-cap m off cap 32*)
and *off + 21* ∈ *Mapping.keys* (*store-cap m off cap 32*)
and *off + 22* ∈ *Mapping.keys* (*store-cap m off cap 32*)
and *off + 23* ∈ *Mapping.keys* (*store-cap m off cap 32*)
and *off + 24* ∈ *Mapping.keys* (*store-cap m off cap 32*)
and *off + 25* ∈ *Mapping.keys* (*store-cap m off cap 32*)
and *off + 26* ∈ *Mapping.keys* (*store-cap m off cap 32*)
and *off + 27* ∈ *Mapping.keys* (*store-cap m off cap 32*)
and *off + 28* ∈ *Mapping.keys* (*store-cap m off cap 32*)
and *off + 29* ∈ *Mapping.keys* (*store-cap m off cap 32*)
and *off + 30* ∈ *Mapping.keys* (*store-cap m off cap 32*)
and *off + 31* ∈ *Mapping.keys* (*store-cap m off cap 32*)
⟨*proof*⟩

lemma *store-cap-correct-val*:
shows *Mapping.lookup* (*store-cap m off cap 32*) *off* =
Some (*ACap* (*mem-capability.truncate cap*) *31*)
and *Mapping.lookup* (*store-cap m off cap 32*) (*off + 1*) =
Some (*ACap* (*mem-capability.truncate cap*) *30*)
and *Mapping.lookup* (*store-cap m off cap 32*) (*off + 2*) =
Some (*ACap* (*mem-capability.truncate cap*) *29*)
and *Mapping.lookup* (*store-cap m off cap 32*) (*off + 3*) =
Some (*ACap* (*mem-capability.truncate cap*) *28*)

and *Mapping.lookup* (*store-cap m off cap 32*) (*off + 4*) =
Some (*ACap* (*mem-capability.truncate cap*) *27*)
and *Mapping.lookup* (*store-cap m off cap 32*) (*off + 5*) =
Some (*ACap* (*mem-capability.truncate cap*) *26*)
and *Mapping.lookup* (*store-cap m off cap 32*) (*off + 6*) =
Some (*ACap* (*mem-capability.truncate cap*) *25*)
and *Mapping.lookup* (*store-cap m off cap 32*) (*off + 7*) =
Some (*ACap* (*mem-capability.truncate cap*) *24*)
and *Mapping.lookup* (*store-cap m off cap 32*) (*off + 8*) =
Some (*ACap* (*mem-capability.truncate cap*) *23*)
and *Mapping.lookup* (*store-cap m off cap 32*) (*off + 9*) =
Some (*ACap* (*mem-capability.truncate cap*) *22*)
and *Mapping.lookup* (*store-cap m off cap 32*) (*off + 10*) =
Some (*ACap* (*mem-capability.truncate cap*) *21*)
and *Mapping.lookup* (*store-cap m off cap 32*) (*off + 11*) =
Some (*ACap* (*mem-capability.truncate cap*) *20*)
and *Mapping.lookup* (*store-cap m off cap 32*) (*off + 12*) =
Some (*ACap* (*mem-capability.truncate cap*) *19*)
and *Mapping.lookup* (*store-cap m off cap 32*) (*off + 13*) =
Some (*ACap* (*mem-capability.truncate cap*) *18*)
and *Mapping.lookup* (*store-cap m off cap 32*) (*off + 14*) =
Some (*ACap* (*mem-capability.truncate cap*) *17*)
and *Mapping.lookup* (*store-cap m off cap 32*) (*off + 15*) =
Some (*ACap* (*mem-capability.truncate cap*) *16*)
and *Mapping.lookup* (*store-cap m off cap 32*) (*off + 16*) =
Some (*ACap* (*mem-capability.truncate cap*) *15*)
and *Mapping.lookup* (*store-cap m off cap 32*) (*off + 17*) =
Some (*ACap* (*mem-capability.truncate cap*) *14*)
and *Mapping.lookup* (*store-cap m off cap 32*) (*off + 18*) =
Some (*ACap* (*mem-capability.truncate cap*) *13*)
and *Mapping.lookup* (*store-cap m off cap 32*) (*off + 19*) =
Some (*ACap* (*mem-capability.truncate cap*) *12*)
and *Mapping.lookup* (*store-cap m off cap 32*) (*off + 20*) =
Some (*ACap* (*mem-capability.truncate cap*) *11*)
and *Mapping.lookup* (*store-cap m off cap 32*) (*off + 21*) =
Some (*ACap* (*mem-capability.truncate cap*) *10*)
and *Mapping.lookup* (*store-cap m off cap 32*) (*off + 22*) =
Some (*ACap* (*mem-capability.truncate cap*) *9*)
and *Mapping.lookup* (*store-cap m off cap 32*) (*off + 23*) =
Some (*ACap* (*mem-capability.truncate cap*) *8*)
and *Mapping.lookup* (*store-cap m off cap 32*) (*off + 24*) =
Some (*ACap* (*mem-capability.truncate cap*) *7*)
and *Mapping.lookup* (*store-cap m off cap 32*) (*off + 25*) =
Some (*ACap* (*mem-capability.truncate cap*) *6*)
and *Mapping.lookup* (*store-cap m off cap 32*) (*off + 26*) =
Some (*ACap* (*mem-capability.truncate cap*) *5*)
and *Mapping.lookup* (*store-cap m off cap 32*) (*off + 27*) =
Some (*ACap* (*mem-capability.truncate cap*) *4*)
and *Mapping.lookup* (*store-cap m off cap 32*) (*off + 28*) =

Some (*ACap* (*mem-capability.truncate cap*) *3*)
and *Mapping.lookup* (*store-cap m off cap 32*) (*off* + *29*) =
Some (*ACap* (*mem-capability.truncate cap*) *2*)
and *Mapping.lookup* (*store-cap m off cap 32*) (*off* + *30*) =
Some (*ACap* (*mem-capability.truncate cap*) *1*)
and *Mapping.lookup* (*store-cap m off cap 32*) (*off* + *31*) =
Some (*ACap* (*mem-capability.truncate cap*) *0*)
⟨*proof*⟩

corollary *store-cap-imp-is-contiguous-cap*:
is-contiguous-cap (*store-cap m off cap 32*) (*mem-capability.truncate cap*) *off 32*
⟨*proof*⟩

lemma *stored-tval-is-cap*:
assumes *val* = *Cap-v v*
shows *is-cap* (*content* (*store-tval obj off val*)) *off*
⟨*proof*⟩

lemma *stored-tval-contiguous-cap*:
assumes *val* = *Cap-v cap*
shows *is-contiguous-cap* (*content* (*store-tval obj off val*)) (*mem-capability.truncate cap*) *off* $|$*memval-type val*$|_\tau$
⟨*proof*⟩

lemma *decode-encoded-u16-in-mem*:
cat-u16 (*retrieve-bytes* (*content* (*store-tval obj off* (*Uint16-v x*))) *off* $|$*Uint16*$|_\tau$) = *x*
⟨*proof*⟩

lemma *decode-encoded-s16-in-mem*:
cat-s16 (*retrieve-bytes* (*content* (*store-tval obj off* (*Sint16-v x*))) *off* $|$*Sint16*$|_\tau$) = *x*
⟨*proof*⟩

lemma *decode-encoded-u32-in-mem*:
cat-u32 (*retrieve-bytes* (*content* (*store-tval obj off* (*Uint32-v x*))) *off* $|$*Uint32*$|_\tau$) = *x*
⟨*proof*⟩

lemma *decode-encoded-s32-in-mem*:
cat-s32 (*retrieve-bytes* (*content* (*store-tval obj off* (*Sint32-v x*))) *off* $|$*Sint32*$|_\tau$) = *x*
⟨*proof*⟩

lemma *cat-flatten-u64-contents-eq*:
cat-u64 [*flatten-u64 vs* ! *0*, *flatten-u64 vs* ! *1*, *flatten-u64 vs* ! *2*, *flatten-u64 vs* ! *3*, *flatten-u64 vs* ! *4*, *flatten-u64 vs* ! *5*, *flatten-u64 vs* ! *6*, *flatten-u64 vs* ! *7*] = *vs*
⟨*proof*⟩

lemma *cat-flatten-s64-contents-eq*:
cat-s64 [*flatten-s64 vs* ! *0*, *flatten-s64 vs* ! *1*, *flatten-s64 vs* ! *2*, *flatten-s64 vs* ! *3*, *flatten-s64 vs* ! *4*, *flatten-s64 vs* ! *5*, *flatten-s64 vs* ! *6*, *flatten-s64 vs* ! *7*] = *vs*
⟨*proof*⟩

lemma *decode-encoded-u64-in-mem*:
cat-u64 (*retrieve-bytes* (*content* (*store-tval obj off* (*Uint64-v x*))) *off* |*Uint64*|$_\tau$) = *x*
⟨*proof*⟩

lemma *decode-encoded-s64-in-mem*:
cat-s64 (*retrieve-bytes* (*content* (*store-tval obj off* (*Sint64-v x*))) *off* |*Sint64*|$_\tau$) = *x*
⟨*proof*⟩

lemma *retrieve-stored-tval-cap*:
assumes *val* = *Cap-v v*
shows *retrieve-tval* (*store-tval obj off val*) *off* (*memval-type val*) *True* = *val*
⟨*proof*⟩

lemma *retrieve-stored-tval-cap-no-perm-cap-load*:
assumes *val* = *Cap-v v*
shows *retrieve-tval* (*store-tval obj off val*) *off* (*memval-type val*) *False* = (*Cap-v* (*v* (| *tag* := *False* |)))
⟨*proof*⟩

lemma *retrieve-stored-tval-u8*:
assumes *val* = *Uint8-v v*
shows *retrieve-tval* (*store-tval obj off val*) *off* (*memval-type val*) *b* = *val*
⟨*proof*⟩

lemma *retrieve-stored-tval-s8*:
assumes *val* = *Sint8-v v*
shows *retrieve-tval* (*store-tval obj off val*) *off* (*memval-type val*) *b* = *val*
⟨*proof*⟩

lemma *retrieve-stored-tval-u16*:
assumes *val* = *Uint16-v v*
shows *retrieve-tval* (*store-tval obj off val*) *off* (*memval-type val*) *b* = *val*
⟨*proof*⟩

lemma *retrieve-stored-tval-s16*:
assumes *val* = *Sint16-v v*
shows *retrieve-tval* (*store-tval obj off val*) *off* (*memval-type val*) *b* = *val*
⟨*proof*⟩

lemma *retrieve-stored-tval-u32*:
assumes *val* = *Uint32-v v*
shows *retrieve-tval* (*store-tval obj off val*) *off* (*memval-type val*) *b* = *val*

⟨*proof*⟩

lemma *retrieve-stored-tval-s32*:
assumes *val* = *Sint32-v v*
shows *retrieve-tval* (*store-tval obj off val*) *off* (*memval-type val*) *b* = *val*
⟨*proof*⟩

lemma *retrieve-stored-tval-u64*:
assumes *val* = *Uint64-v v*
shows *retrieve-tval* (*store-tval obj off val*) *off* (*memval-type val*) *b* = *val*
⟨*proof*⟩

lemma *retrieve-stored-tval-s64*:
assumes *val* = *Sint64-v v*
shows *retrieve-tval* (*store-tval obj off val*) *off* (*memval-type val*) *b* = *val*
⟨*proof*⟩

lemma *memcap-truncate-extend-equiv*:
mem-capability.extend (*mem-capability.truncate c*) (| *tag* = *tag c* |) = *c*
⟨*proof*⟩

corollary *Acap-truncate-extend-equiv*:
mem-capability.extend (*of-cap* (*ACap* (*mem-capability.truncate c*) *n*)) (| *tag* = *tag c* |) = *c*
⟨*proof*⟩

lemma *memcap-truncate-extend-gen*:
mem-capability.extend (*mem-capability.truncate c*) (| *tag* = *b* |) = *c* (| *tag* := *b* |)
⟨*proof*⟩

corollary *Acap-truncate-extend-gen*:
mem-capability.extend (*of-cap* (*ACap* (*mem-capability.truncate c*) *n*)) (| *tag* = *b* |) = *c* (| *tag* := *b* |)
⟨*proof*⟩

lemma *retrieve-stored-tval-cap-frag*:
assumes *val* = *Cap-v-frag c n*
shows *retrieve-tval* (*store-tval obj off val*) *off* (*memval-type val*) *b* =
Cap-v-frag (*c* (| *tag* := *False* |)) *n*
⟨*proof*⟩

lemmas *retrieve-stored-tval-prim* = *retrieve-stored-tval-u8 retrieve-stored-tval-s8*
retrieve-stored-tval-u16 retrieve-stored-tval-s16
retrieve-stored-tval-u32 retrieve-stored-tval-s32
retrieve-stored-tval-u64 retrieve-stored-tval-s64

lemma *retrieve-stored-tval-any-perm*:
assumes *val* ≠ *Undef*
and ∀ *v*. *val* ≠ *Cap-v v*

and $\forall\ v\ n.\ val \neq Cap\text{-}v\text{-}frag\ v\ n$
shows *retrieve-tval* (*store-tval obj off val*) *off* (*memval-type val*) *b* = *val*
⟨*proof*⟩

lemma *retrieve-stored-tval-with-perm-cap-load*:
assumes *val* ≠ *Undef*
and $\forall\ v\ n.\ val \neq Cap\text{-}v\text{-}frag\ v\ n$
shows *retrieve-tval* (*store-tval obj off val*) *off* (*memval-type val*) *True* = *val*
⟨*proof*⟩

lemma *store-bytes-domain-1*:
assumes *x* + *length vs* ≤ *n*
shows *Mapping.lookup* (*store-bytes m n vs*) *x* = *Mapping.lookup m x*
⟨*proof*⟩

lemma *store-bytes-domain-2*:
assumes *n* + *length vs* ≤ *x*
shows *Mapping.lookup* (*store-bytes m n vs*) *x* = *Mapping.lookup m x*
⟨*proof*⟩

lemma *store-bytes-keys-1*:
Set.filter (λ *x*. *x* + *length vs* ≤ *n*) (*Mapping.keys m*) =
Set.filter (λ *x*. *x* + *length vs* ≤ *n*) (*Mapping.keys* (*store-bytes m n vs*))
⟨*proof*⟩

lemma *store-bytes-keys-2*:
Set.filter (λ *x*. *n* + *length vs* ≤ *x*) (*Mapping.keys m*) =
Set.filter (λ *x*. *n* + *length vs* ≤ *x*) (*Mapping.keys* (*store-bytes m n vs*))
⟨*proof*⟩

lemma *store-cap-domain-1*:
assumes *x* + *n* ≤ *p*
shows *Mapping.lookup* (*store-cap m p c n*) *x* = *Mapping.lookup m x*
⟨*proof*⟩

lemma *store-cap-domain-2*:
assumes *p* + *n* ≤ *x*
shows *Mapping.lookup* (*store-cap m p c n*) *x* = *Mapping.lookup m x*
⟨*proof*⟩

lemma *store-cap-keys-1*:
Set.filter (λ *x*. *x* + *n* ≤ *p*) (*Mapping.keys m*) =
Set.filter (λ *x*. *x* + *n* ≤ *p*) (*Mapping.keys* (*store-cap m p c n*))
⟨*proof*⟩

lemma *store-cap-keys-2*:
Set.filter (λ *x*. *p* + *n* ≤ *x*) (*Mapping.keys m*) =
Set.filter (λ *x*. *p* + *n* ≤ *x*) (*Mapping.keys* (*store-cap m p c n*))
⟨*proof*⟩

lemma *store-tags-domain-1*:
assumes $x < n$
shows *Mapping.lookup* (*store-tag m n b*) x = *Mapping.lookup m x*
⟨*proof*⟩

lemma *store-tags-domain-2*:
assumes $n < x$
shows *Mapping.lookup* (*store-tag m n b*) x = *Mapping.lookup m x*
⟨*proof*⟩

lemma *store-tags-keys-1*:
Set.filter (λ *x*. $x < n$) (*Mapping.keys m*) =
Set.filter (λ *x*. $x < n$) (*Mapping.keys* (*store-tag m n b*))
⟨*proof*⟩

lemma *store-tags-keys-2*:
Set.filter (λ *x*. $n < x$) (*Mapping.keys m*) =
Set.filter (λ *x*. $n < x$) (*Mapping.keys* (*store-tag m n b*))
⟨*proof*⟩

lemma *cap-offset-aligned*:
(*cap-offset n*) *mod* $|Cap|_\tau$ = *0*
⟨*proof*⟩

lemma *store-tags-offset*:
assumes *Set.filter* (λ *x*. *x mod* $|Cap|_\tau \neq 0$) (*Mapping.keys m*) = {}
shows *Set.filter* (λ *x*. *x mod* $|Cap|_\tau \neq 0$) (*Mapping.keys* (*store-tag m* (*cap-offset n*) *b*)) = {}
⟨*proof*⟩

lemma *store-tval-disjoint-bounds*:
assumes *store-tval obj off val* = *obj'*
and *val* ≠ *Undef*
shows *bounds obj* = *bounds obj'*
⟨*proof*⟩

lemma *store-tval-disjoint-1-content*:
assumes *store-tval obj off val* = *obj'*
and *val* ≠ *Undef*
and *off'* < *off*
shows *Mapping.lookup* (*content obj*) *off'* = *Mapping.lookup* (*content obj'*) *off'*
⟨*proof*⟩

lemma *store-tval-disjoint-1-content-bytes*:
assumes *store-tval obj off val* = *obj'*
and *val* ≠ *Undef*
and *off'* + *n* ≤ *off*
shows *retrieve-bytes* (*content obj*) *off' n* = *retrieve-bytes* (*content obj'*) *off' n*

⟨*proof*⟩

lemma *store-tval-disjoint-1-content-contiguous-bytes*:
assumes *store-tval obj off val* = *obj′*
and *val* ≠ *Undef*
and *off′* + *n* ≤ *off*
shows *is-contiguous-bytes* (*content obj*) *off′ n* = *is-contiguous-bytes* (*content obj′*) *off′ n*
⟨*proof*⟩

lemma *store-tval-disjoint-1-content-contiguous-caps*:
assumes *store-tval obj off val* = *obj′*
and *val* ≠ *Undef*
and *off′* + *n* ≤ *off*
shows *is-contiguous-cap* (*content obj*) *cap off′ n* = *is-contiguous-cap* (*content obj′*) *cap off′ n*
⟨*proof*⟩

lemma *store-tval-disjoint-1-tags*:
assumes *store-tval obj off val* = *obj′*
and *val* ≠ *Undef*
and *off′* + |*Cap*|$_\tau$ ≤ *off*
shows *Mapping.lookup* (*tags obj*) *off′* = *Mapping.lookup* (*tags obj′*) *off′*
⟨*proof*⟩

lemma *store-tval-disjoint-2-content*:
assumes *store-tval obj off val* = *obj′*
and *val* ≠ *Undef*
and *off* + |*memval-type val*|$_\tau$ ≤ *off′*
shows *Mapping.lookup* (*content obj*) *off′* = *Mapping.lookup* (*content obj′*) *off′*
⟨*proof*⟩

lemma *store-tval-disjoint-2-content-bytes*:
assumes *store-tval obj off val* = *obj′*
and *val* ≠ *Undef*
and *off* + |*memval-type val*|$_\tau$ ≤ *off′*
shows *retrieve-bytes* (*content obj*) *off′ n* = *retrieve-bytes* (*content obj′*) *off′ n*
⟨*proof*⟩

lemma *store-tval-disjoint-2-content-contiguous-bytes*:
assumes *store-tval obj off val* = *obj′*
and *val* ≠ *Undef*
and *off* + |*memval-type val*|$_\tau$ ≤ *off′*
shows *is-contiguous-bytes* (*content obj*) *off′ n* = *is-contiguous-bytes* (*content obj′*) *off′ n*
⟨*proof*⟩

lemma *store-tval-disjoint-2-content-contiguous-caps*:
assumes *store-tval obj off val = obj'*
and *val ≠ Undef*
and *off + |memval-type val|$_\tau$ ≤ off'*
shows *is-contiguous-cap (content obj) cap off' n = is-contiguous-cap (content obj') cap off' n*
⟨*proof*⟩

lemma *store-tval-disjoint-2-tags*:
assumes *store-tval obj off val = obj'*
and *val ≠ Undef*
and *off + |memval-type val|$_\tau$ ≤ off'*
shows *Mapping.lookup (tags obj) off' = Mapping.lookup (tags obj') off'*
⟨*proof*⟩

lemma *zero-imp-bytes*:
is-contiguous-zeros obj off n ⟹ ¬ is-contiguous-bytes obj off n ⟹ False
⟨*proof*⟩

lemma *retrieve-stored-tval-disjoint-1*:
assumes *store-tval obj off val = obj'*
and *val ≠ Undef*
and *off' + |t|$_\tau$ ≤ off*
shows *retrieve-tval obj off' t b = retrieve-tval obj' off' t b*
⟨*proof*⟩

lemma *retrieve-stored-tval-disjoint-2*:
assumes *store-tval obj off val = obj'*
and *val ≠ Undef*
and *off + |memval-type val|$_\tau$ ≤ off'*
and *t = Cap ⟹ off' mod |Cap|$_\tau$ = 0*
shows *retrieve-tval obj off' t b = retrieve-tval obj' off' t b*
⟨*proof*⟩

lemma *type-uniq*:
assumes *∃ x n. ret = Cap-v-frag x n*
shows *ret ≠ Uint8-v v1 ret ≠ Sint8-v v2 ret ≠ Uint16-v v3 ret ≠ Sint16-v v4*
ret ≠ Uint32-v v5 ret ≠ Sint32-v v6 ret ≠ Uint64-v v7 ret ≠ Sint64-v v8
ret ≠ Cap-v v9
⟨*proof*⟩

5 Memory Actions / Operations

definition *alloc :: heap ⇒ bool ⇒ nat ⇒ (heap × cap) result*
where
alloc h c s ≡
let cap = (| block-id = (next-block h),
offset = 0,

```
                base = 0,
                len = s,
                perm-load = True,
                perm-cap-load = c,
                perm-store = True,
                perm-cap-store = c,
                perm-cap-store-local = c,
                perm-global = False,
                tag = True
              |) in
    let h' = h (| next-block := (next-block h) + 1,
                  heap-map := Mapping.update
                                (next-block h)
                                (Map (| bounds = (0, s),
                                        content = Mapping.empty,
                                        tags = Mapping.empty
                                     |)
                                ) (heap-map h)
               |) in
    Success (h', cap)

definition free :: heap ⇒ cap ⇒ (heap × cap) result
  where
  free h c ≡
    if c = NULL then Success (h, c) else
    if tag c = False then Error (C2Err (TagViolation)) else
    if perm-global c = True then Error (LogicErr (Unhandled 0)) else
    let obj = Mapping.lookup (heap-map h) (block-id c) in
    (case obj of None      ⇒ Error (LogicErr (MissingResource))
               | Some cobj ⇒
      (case cobj of Freed ⇒ Error (LogicErr (UseAfterFree))
                  | Map m ⇒
        if offset c ≠ 0 then Error (LogicErr (Unhandled 0))
        else if offset c > base c + len c then Error (LogicErr (Unhandled 0)) else
      let cap-bound = (base c, base c + len c) in
      if cap-bound ≠ bounds m then Error (LogicErr (Unhandled 0)) else
      let h' = h (| heap-map := Mapping.update (block-id c) Freed (heap-map h) |)
in
      let cap = c (| tag := False |) in
      Success (h', cap)))
```

How load works: The hardware would perform a CL[C] operation on the given capability first. An invalid capability for load would be caught by the hardware. Once all the hardware checks are performed, we then proceed to the logical checks.

```
definition load :: heap ⇒ cap ⇒ cctype ⇒ block ccval result
  where
  load h c t ≡
    if tag c = False then
```

```
    Error (C2Err TagViolation)
  else if perm-load c = False then
    Error (C2Err PermitLoadViolation)
  else if offset c + |t|_τ > base c + len c then
    Error (C2Err LengthViolation)
  else if offset c < base c then
    Error (C2Err LengthViolation)
  else if offset c mod |t|_τ ≠ 0 then
    Error (C2Err BadAddressViolation)
  else
    let obj = Mapping.lookup (heap-map h) (block-id c) in
   (case obj of None      ⇒ Error (LogicErr (MissingResource))
              | Some cobj ⇒
     (case cobj of Freed ⇒ Error (LogicErr (UseAfterFree))
                 | Map m ⇒ if offset c < fst (bounds m) ∨ offset c + |t|_τ > snd
(bounds m) then
                           Error (LogicErr BufferOverrun) else
                           Success (retrieve-tval m (nat (offset c)) t (perm-cap-load
c))))

definition store :: heap ⇒ cap ⇒ block ccval ⇒ heap result
  where
  store h c v ≡
    if tag c = False then
      Error (C2Err TagViolation)
    else if perm-store c = False then
      Error (C2Err PermitStoreViolation)
    else if (case v of Cap-v cv ⇒ ¬ perm-cap-store c ∧ tag cv | - ⇒ False) then
      Error (C2Err PermitStoreCapViolation)
   else if (case v of Cap-v cv ⇒ ¬ perm-cap-store-local c ∧ tag cv ∧ ¬ perm-global
cv | - ⇒ False) then
      Error (C2Err PermitStoreLocalCapViolation)
    else if offset c + |memval-type v|_τ > base c + len c then
      Error (C2Err LengthViolation)
    else if offset c < base c then
      Error (C2Err LengthViolation)
    else if offset c mod |memval-type v|_τ ≠ 0 then
      Error (C2Err BadAddressViolation)
    else if v = Undef then
      Error (LogicErr (Unhandled 0))
    else
      let obj = Mapping.lookup (heap-map h) (block-id c) in
     (case obj of None      ⇒ Error (LogicErr (MissingResource))
                | Some cobj ⇒
       (case cobj of Freed ⇒ Error (LogicErr (UseAfterFree))
                   | Map m ⇒ if offset c < fst (bounds m) ∨ offset c + |memval-type
v|_τ > snd (bounds m) then
                             Error (LogicErr BufferOverrun) else
                             Success (h (| heap-map := Mapping.update
```

(*block-id c*)
(*Map* (*store-tval m* (*nat* (*offset c*)) *v*))
(*heap-map h*) ⦇⦈))))

5.1 Properties of the operations

Here we provide all the properties the operations satisfy. In general, you may find the following forms of proofs:

- If we have valid input, the operation will succeed
- If we have invalid inputs, the operations will return the appropriate error
- If the operation succeeds, we have a valid input

good variable laws are also proven at the next subsubsection.

5.1.1 Correctness Properties

lemma *alloc-always-success*:
∃! *res*. *alloc h c s* = *Success res*
⟨*proof*⟩

schematic-goal *alloc-updated-heap-and-cap*:
alloc h c s = *Success* (*?h'*, *?cap*)
⟨*proof*⟩

lemma *alloc-never-fails*:
alloc h c s = *Error e* ⟹ *False*
⟨*proof*⟩

In practice, malloc may actually return NULL when allocation fails. However, this still complies with The C Standard.

lemma *alloc-no-null-ret*:
assumes *alloc h c s* = *Success* (*h'*, *cap*)
shows *cap* ≠ *NULL*
⟨*proof*⟩

lemma *alloc-correct*:
assumes *alloc h c s* = *Success* (*h'*, *cap*)
shows *next-block h'* = *next-block h* + *1*
and *Mapping.lookup* (*heap-map h'*) (*next-block h*)
= *Some* (*Map* ⦇ *bounds* = (*0*, *s*), *content* = *Mapping.empty*, *tags* = *Mapping.empty*⦈)
⟨*proof*⟩

Section 7.20.3.2 of The C Standard states free(NULL) results in no action occurring [2].

lemma *free-null*:
free h NULL = Success (h, NULL)
⟨*proof*⟩

lemma *free-false-tag*:
assumes *c ≠ NULL*
and *tag c = False*
shows *free h c = Error (C2Err (TagViolation))*
⟨*proof*⟩

lemma *free-global-cap*:
assumes *c ≠ NULL*
and *tag c = True*
and *perm-global c = True*
shows *free h c = Error (LogicErr (Unhandled 0))*
⟨*proof*⟩

lemma *free-nonexistant-obj*:
assumes *c ≠ NULL*
and *tag c = True*
and *perm-global c = False*
and *Mapping.lookup (heap-map h) (block-id c) = None*
shows *free h c = Error (LogicErr (MissingResource))*
⟨*proof*⟩

This case may arise if there are copies of the same capability, where only one was freed. It is worth noting that due to this, temporal safety is not guaranteed by the CHERI hardware.

lemma *free-double-free*:
assumes *c ≠ NULL*
and *tag c = True*
and *perm-global c = False*
and *Mapping.lookup (heap-map h) (block-id c) = Some Freed*
shows *free h c = Error (LogicErr (UseAfterFree))*
⟨*proof*⟩

An incorrect offset implies the actual ptr value is not that returned by alloc. Section 7.20.3.2 of The C Standard states this leads to undefined behaviour [2]. Clang, in practice, however, terminates the C program with an invalid pointer error.

lemma *free-incorrect-cap-offset*:
assumes *c ≠ NULL*
and *tag c = True*
and *perm-global c = False*
and *Mapping.lookup (heap-map h) (block-id c) = Some (Map m)*
and *offset c ≠ 0*
shows *free h c = Error (LogicErr (Unhandled 0))*
⟨*proof*⟩

lemma *free-incorrect-bounds*:
assumes $c \neq NULL$
and *tag* $c = True$
and *perm-global* $c = False$
and *Mapping.lookup* (*heap-map* h) (*block-id* c) = *Some* (*Map* m)
and *offset* $c = 0$
and *bounds* $m \neq$ (*base* c, *base* c + *len* c)
shows *free* h c = *Error* (*LogicErr* (*Unhandled* 0))
⟨*proof*⟩

lemma *free-non-null-correct*:
assumes $c \neq NULL$
and *valid-tag*: *tag* $c = True$
and *perm-global* $c = False$
and *map-has-contents*: *Mapping.lookup* (*heap-map* h) (*block-id* c) = *Some* (*Map* m)
and *offset-correct*: *offset* $c = 0$
and *bounds-correct*: *bounds* m = (*base* c, *base* c + *len* c)
shows *free* h c = *Success* (h (| *heap-map* := *Mapping.update* (*block-id* c) *Freed* (*heap-map* h) |), c (| *tag* := *False* |))
⟨*proof*⟩

lemma *free-cond*:
assumes *free* h c = *Success* (h', *cap*)
shows $c \neq NULL \Longrightarrow$ *tag* $c = True$
and $c \neq NULL \Longrightarrow$ *perm-global* $c = False$
and $c \neq NULL \Longrightarrow$ *offset* $c = 0$
and $c \neq NULL \Longrightarrow \exists\ m.$ *Mapping.lookup* (*heap-map* h) (*block-id* c) = *Some* (*Map* m) $\wedge$
bounds m = (*base* c, *base* c + *len* c)
and $c \neq NULL \Longrightarrow$ *Mapping.lookup* (*heap-map* h') (*block-id* c) = *Some Freed*
and $c \neq NULL \Longrightarrow$ *cap* = c (| *tag* := *False* |)
and $c = NULL \Longrightarrow$ (h, c) = (h', *cap*)
⟨*proof*⟩

lemmas *free-cond-non-null* = *free-cond*(*1*) *free-cond*(*2*) *free-cond*(*3*) *free-cond*(*4*) *free-cond*(*5*) *free-cond*(*6*)

lemma *double-free*:
assumes *free* h c = *Success* (h', *cap*)
and *cap* $\neq$ *NULL*
shows *free* h' *cap* = *Error* (*C2Err TagViolation*)
⟨*proof*⟩

lemma *free-next-block*:
assumes *free* h *cap* = *Success* (h', *cap*$'$)
shows *next-block* h = *next-block* h'

⟨*proof*⟩

lemma *load-null-error*:
load h NULL t = *Error* (*C2Err TagViolation*)
⟨*proof*⟩

lemma *load-false-tag*:
assumes *tag c* = *False*
shows *load h c t* = *Error* (*C2Err TagViolation*)
⟨*proof*⟩

lemma *load-false-perm-load*:
assumes *tag c* = *True*
and *perm-load c* = *False*
shows *load h c t* = *Error* (*C2Err PermitLoadViolation*)
⟨*proof*⟩

lemma *load-bound-over*:
assumes *tag c* = *True*
and *perm-load c* = *True*
and *offset c* + |*t*|$_\tau$ > *base c* + *len c*
shows *load h c t* = *Error* (*C2Err LengthViolation*)
⟨*proof*⟩

lemma *load-bound-under*:
assumes *tag c* = *True*
and *perm-load c* = *True*
and *offset c* + |*t*|$_\tau$ ≤ *base c* + *len c*
and *offset c* < *base c*
shows *load h c t* = *Error* (*C2Err LengthViolation*)
⟨*proof*⟩

lemma *load-misaligned*:
assumes *tag c* = *True*
and *perm-load c* = *True*
and *offset c* + |*t*|$_\tau$ ≤ *base c* + *len c*
and *offset c* ≥ *base c*
and *offset c mod* |*t*|$_\tau$ ≠ *0*
shows *load h c t* = *Error* (*C2Err BadAddressViolation*)
⟨*proof*⟩

lemma *load-nonexistant-obj*:
assumes *tag c* = *True*
and *perm-load c* = *True*
and *offset c* + |*t*|$_\tau$ ≤ *base c* + *len c*
and *offset c* ≥ *base c*
and *offset c mod* |*t*|$_\tau$ = *0*
and *Mapping.lookup* (*heap-map h*) (*block-id c*) = *None*
shows *load h c t* = *Error* (*LogicErr MissingResource*)

⟨*proof*⟩

lemma *load-load-after-free*:
assumes *tag c* = *True*
and *perm-load c* = *True*
and *offset c* + |*t*|$_\tau$ ≤ *base c* + *len c*
and *offset c* ≥ *base c*
and *offset c mod* |*t*|$_\tau$ = *0*
and *Mapping.lookup* (*heap-map h*) (*block-id c*) = *Some Freed*
shows *load h c t* = *Error* (*LogicErr UseAfterFree*)
⟨*proof*⟩

lemma *load-cap-on-heap-bounds-fail-1*:
assumes *tag c* = *True*
and *perm-load c* = *True*
and *offset c* + |*t*|$_\tau$ ≤ *base c* + *len c*
and *offset c* ≥ *base c*
and *offset c mod* |*t*|$_\tau$ = *0*
and *Mapping.lookup* (*heap-map h*) (*block-id c*) = *Some* (*Map m*)
and *is-contiguous-bytes* (*content m*) (*nat* (*offset c*)) |*t*|$_\tau$
and *t* = *Cap*
and ¬ *is-contiguous-zeros* (*content m*) (*nat* (*offset c*)) |*t*|$_\tau$
and *offset c* < *fst* (*bounds m*)
shows *load h c t* = *Error* (*LogicErr BufferOverrun*)
⟨*proof*⟩

lemma *load-cap-on-heap-bounds-fail-2*:
assumes *tag c* = *True*
and *perm-load c* = *True*
and *offset c* + |*t*|$_\tau$ ≤ *base c* + *len c*
and *offset c* ≥ *base c*
and *offset c mod* |*t*|$_\tau$ = *0*
and *Mapping.lookup* (*heap-map h*) (*block-id c*) = *Some* (*Map m*)
and *is-contiguous-bytes* (*content m*) (*nat* (*offset c*)) |*t*|$_\tau$
and *t* = *Cap*
and ¬ *is-contiguous-zeros* (*content m*) (*nat* (*offset c*)) |*t*|$_\tau$
and *offset c* + |*t*|$_\tau$ > *snd* (*bounds m*)
shows *load h c t* = *Error* (*LogicErr BufferOverrun*)
⟨*proof*⟩

lemma *load-cap-on-membytes-fail*:
assumes *tag c* = *True*
and *perm-load c* = *True*
and *offset c* + |*t*|$_\tau$ ≤ *base c* + *len c*
and *offset c* ≥ *base c*
and *offset c mod* |*t*|$_\tau$ = *0*
and *Mapping.lookup* (*heap-map h*) (*block-id c*) = *Some* (*Map m*)
and *is-contiguous-bytes* (*content m*) (*nat* (*offset c*)) |*t*|$_\tau$
and *t* = *Cap*

and ¬ *is-contiguous-zeros* (*content m*) (*nat* (*offset c*)) $|t|_\tau$
and *offset c* ≥ *fst* (*bounds m*)
and *offset c* + $|t|_\tau$ ≤ *snd* (*bounds m*)
shows *load h c t* = *Success Undef*
⟨*proof*⟩

lemma *load-null-cap-on-membytes*:
assumes *tag c* = *True*
and *perm-load c* = *True*
and *offset c* + $|t|_\tau$ ≤ *base c* + *len c*
and *offset c* ≥ *base c*
and *offset c mod* $|t|_\tau$ = *0*
and *Mapping.lookup* (*heap-map h*) (*block-id c*) = *Some* (*Map m*)
and *is-contiguous-bytes* (*content m*) (*nat* (*offset c*)) $|t|_\tau$
and *t* = *Cap*
and *offset c* ≥ *fst* (*bounds m*)
and *offset c* + $|t|_\tau$ ≤ *snd* (*bounds m*)
and *is-contiguous-zeros* (*content m*) (*nat* (*offset c*)) $|t|_\tau$
shows *load h c t* = *Success* (*Cap-v NULL*)
⟨*proof*⟩

lemma *load-u8-on-membytes*:
assumes *tag c* = *True*
and *perm-load c* = *True*
and *offset c* + $|t|_\tau$ ≤ *base c* + *len c*
and *offset c* ≥ *base c*
and *offset c mod* $|t|_\tau$ = *0*
and *Mapping.lookup* (*heap-map h*) (*block-id c*) = *Some* (*Map m*)
and *offset c* ≥ *fst* (*bounds m*)
and *offset c* + $|t|_\tau$ ≤ *snd* (*bounds m*)
and *is-contiguous-bytes* (*content m*) (*nat* (*offset c*)) $|t|_\tau$
and *t* = *Uint8*
shows *load h c t* = *Success* (*Uint8-v* (*decode-u8-list* (*retrieve-bytes* (*content m*) (*nat* (*offset c*)) $|t|_\tau$)))
⟨*proof*⟩

lemma *load-s8-on-membytes*:
assumes *tag c* = *True*
and *perm-load c* = *True*
and *offset c* + $|t|_\tau$ ≤ *base c* + *len c*
and *offset c* ≥ *base c*
and *offset c mod* $|t|_\tau$ = *0*
and *Mapping.lookup* (*heap-map h*) (*block-id c*) = *Some* (*Map m*)
and *offset c* ≥ *fst* (*bounds m*)
and *offset c* + $|t|_\tau$ ≤ *snd* (*bounds m*)
and *is-contiguous-bytes* (*content m*) (*nat* (*offset c*)) $|t|_\tau$
and *t* = *Sint8*
shows *load h c t* = *Success* (*Sint8-v* (*decode-s8-list* (*retrieve-bytes* (*content m*) (*nat* (*offset c*)) $|t|_\tau$)))

⟨*proof*⟩

lemma *load-u16-on-membytes*:
assumes *tag c* = *True*
and *perm-load c* = *True*
and *offset c* + |*t*|$_\tau$ ≤ *base c* + *len c*
and *offset c* ≥ *base c*
and *offset c mod* |*t*|$_\tau$ = *0*
and *Mapping.lookup* (*heap-map h*) (*block-id c*) = *Some* (*Map m*)
and *offset c* ≥ *fst* (*bounds m*)
and *offset c* + |*t*|$_\tau$ ≤ *snd* (*bounds m*)
and *is-contiguous-bytes* (*content m*) (*nat* (*offset c*)) |*t*|$_\tau$
and *t* = *Uint16*
shows *load h c t* = *Success* (*Uint16-v* (*cat-u16* (*retrieve-bytes* (*content m*) (*nat* (*offset c*)) |*t*|$_\tau$)))
⟨*proof*⟩

lemma *load-s16-on-membytes*:
assumes *tag c* = *True*
and *perm-load c* = *True*
and *offset c* + |*t*|$_\tau$ ≤ *base c* + *len c*
and *offset c* ≥ *base c*
and *offset c mod* |*t*|$_\tau$ = *0*
and *Mapping.lookup* (*heap-map h*) (*block-id c*) = *Some* (*Map m*)
and *offset c* ≥ *fst* (*bounds m*)
and *offset c* + |*t*|$_\tau$ ≤ *snd* (*bounds m*)
and *is-contiguous-bytes* (*content m*) (*nat* (*offset c*)) |*t*|$_\tau$
and *t* = *Sint16*
shows *load h c t* = *Success* (*Sint16-v* (*cat-s16* (*retrieve-bytes* (*content m*) (*nat* (*offset c*)) |*t*|$_\tau$)))
⟨*proof*⟩

lemma *load-u32-on-membytes*:
assumes *tag c* = *True*
and *perm-load c* = *True*
and *offset c* + |*t*|$_\tau$ ≤ *base c* + *len c*
and *offset c* ≥ *base c*
and *offset c mod* |*t*|$_\tau$ = *0*
and *Mapping.lookup* (*heap-map h*) (*block-id c*) = *Some* (*Map m*)
and *offset c* ≥ *fst* (*bounds m*)
and *offset c* + |*t*|$_\tau$ ≤ *snd* (*bounds m*)
and *is-contiguous-bytes* (*content m*) (*nat* (*offset c*)) |*t*|$_\tau$
and *t* = *Uint32*
shows *load h c t* = *Success* (*Uint32-v* (*cat-u32* (*retrieve-bytes* (*content m*) (*nat* (*offset c*)) |*t*|$_\tau$)))
⟨*proof*⟩

lemma *load-s32-on-membytes*:
assumes *tag c* = *True*

and *perm-load c* = *True*
and *offset c* + $|t|_\tau$ ≤ *base c* + *len c*
and *offset c* ≥ *base c*
and *offset c mod* $|t|_\tau$ = *0*
and *Mapping.lookup* (*heap-map h*) (*block-id c*) = *Some* (*Map m*)
and *offset c* ≥ *fst* (*bounds m*)
and *offset c* + $|t|_\tau$ ≤ *snd* (*bounds m*)
and *is-contiguous-bytes* (*content m*) (*nat* (*offset c*)) $|t|_\tau$
and *t* = *Sint32*
shows *load h c t* = *Success* (*Sint32-v* (*cat-s32* (*retrieve-bytes* (*content m*) (*nat* (*offset c*)) $|t|_\tau$)))
⟨*proof*⟩

lemma *load-u64-on-membytes*:
assumes *tag c* = *True*
and *perm-load c* = *True*
and *offset c* + $|t|_\tau$ ≤ *base c* + *len c*
and *offset c* ≥ *base c*
and *offset c mod* $|t|_\tau$ = *0*
and *Mapping.lookup* (*heap-map h*) (*block-id c*) = *Some* (*Map m*)
and *offset c* ≥ *fst* (*bounds m*)
and *offset c* + $|t|_\tau$ ≤ *snd* (*bounds m*)
and *is-contiguous-bytes* (*content m*) (*nat* (*offset c*)) $|t|_\tau$
and *t* = *Uint64*
shows *load h c t* = *Success* (*Uint64-v* (*cat-u64* (*retrieve-bytes* (*content m*) (*nat* (*offset c*)) $|t|_\tau$)))
⟨*proof*⟩

lemma *load-s64-on-membytes*:
assumes *tag c* = *True*
and *perm-load c* = *True*
and *offset c* + $|t|_\tau$ ≤ *base c* + *len c*
and *offset c* ≥ *base c*
and *offset c mod* $|t|_\tau$ = *0*
and *Mapping.lookup* (*heap-map h*) (*block-id c*) = *Some* (*Map m*)
and *offset c* ≥ *fst* (*bounds m*)
and *offset c* + $|t|_\tau$ ≤ *snd* (*bounds m*)
and *is-contiguous-bytes* (*content m*) (*nat* (*offset c*)) $|t|_\tau$
and *t* = *Sint64*
shows *load h c t* = *Success* (*Sint64-v* (*cat-s64* (*retrieve-bytes* (*content m*) (*nat* (*offset c*)) $|t|_\tau$)))
⟨*proof*⟩

lemma *load-not-cap-in-mem*:
assumes *tag c* = *True*
and *perm-load c* = *True*
and *offset c* + $|t|_\tau$ ≤ *base c* + *len c*
and *offset c* ≥ *base c*
and *offset c mod* $|t|_\tau$ = *0*

and *Mapping.lookup* (*heap-map h*) (*block-id c*) = *Some* (*Map m*)
and *offset c* ≥ *fst* (*bounds m*)
and *offset c* + |*t*|$_\tau$ ≤ *snd* (*bounds m*)
and ¬ *is-contiguous-bytes* (*content m*) (*nat* (*offset c*)) |*t*|$_\tau$
and ¬ *is-cap* (*content m*) (*nat* (*offset c*))
shows *load h c t* = *Success Undef*
⟨*proof*⟩

lemma *load-not-contiguous-cap-in-mem*:
assumes *tag c* = *True*
and *perm-load c* = *True*
and *offset c* + |*t*|$_\tau$ ≤ *base c* + *len c*
and *offset c* ≥ *base c*
and *offset c mod* |*t*|$_\tau$ = *0*
and *Mapping.lookup* (*heap-map h*) (*block-id c*) = *Some* (*Map m*)
and *offset c* ≥ *fst* (*bounds m*)
and *offset c* + |*t*|$_\tau$ ≤ *snd* (*bounds m*)
and ¬ *is-contiguous-bytes* (*content m*) (*nat* (*offset c*)) |*t*|$_\tau$
and *is-cap* (*content m*) (*nat* (*offset c*))
and *mc* = *get-cap* (*content m*) (*nat* (*offset c*))
and ¬ *is-contiguous-cap* (*content m*) *mc* (*nat* (*offset c*)) |*t*|$_\tau$
and *t* ≠ *Uint8*
and *t* ≠ *Sint8*
shows *load h c t* = *Success Undef*
⟨*proof*⟩

lemma *load-cap-frag-u8*:
assumes *tag c* = *True*
and *perm-load c* = *True*
and *offset c* + |*t*|$_\tau$ ≤ *base c* + *len c*
and *offset c* ≥ *base c*
and *offset c mod* |*t*|$_\tau$ = *0*
and *Mapping.lookup* (*heap-map h*) (*block-id c*) = *Some* (*Map m*)
and *offset c* ≥ *fst* (*bounds m*)
and *offset c* + |*t*|$_\tau$ ≤ *snd* (*bounds m*)
and ¬ *is-contiguous-bytes* (*content m*) (*nat* (*offset c*)) |*t*|$_\tau$
and *is-cap* (*content m*) (*nat* (*offset c*))
and *mc* = *get-cap* (*content m*) (*nat* (*offset c*))
and *t* = *Uint8*
and *tagval* = *the* (*Mapping.lookup* (*tags m*) (*cap-offset* (*nat* (*offset c*))))
and *tg* = (*case perm-cap-load c of False* ⇒ *False* | *True* ⇒ *tagval*)
and *nth-frag* = *of-nth* (*the* (*Mapping.lookup* (*content m*) (*nat* (*offset c*))))
shows *load h c t* = *Success* (*Cap-v-frag* (*mem-capability.extend mc* (‖ *tag* = *False* ‖)) *nth-frag*)
⟨*proof*⟩

lemma *load-cap-frag-s8*:
assumes *tag c* = *True*

and *perm-load c* = *True*
and *offset c* + $|t|_\tau$ ≤ *base c* + *len c*
and *offset c* ≥ *base c*
and *offset c mod* $|t|_\tau$ = *0*
and *Mapping.lookup* (*heap-map h*) (*block-id c*) = *Some* (*Map m*)
and *offset c* ≥ *fst* (*bounds m*)
and *offset c* + $|t|_\tau$ ≤ *snd* (*bounds m*)
and ¬ *is-contiguous-bytes* (*content m*) (*nat* (*offset c*)) $|t|_\tau$
and *is-cap* (*content m*) (*nat* (*offset c*))
and *mc* = *get-cap* (*content m*) (*nat* (*offset c*))
and ¬ *is-contiguous-cap* (*content m*) *mc* (*nat* (*offset c*)) $|t|_\tau$
and *t* = *Sint8*
and *tagval* = *the* (*Mapping.lookup* (*tags m*) (*cap-offset* (*nat* (*offset c*))))
and *tg* = (*case perm-cap-load c of False* ⇒ *False* | *True* ⇒ *tagval*)
and *nth-frag* = *of-nth* (*the* (*Mapping.lookup* (*content m*) (*nat* (*offset c*))))
shows *load h c t* = *Success* (*Cap-v-frag* (*mem-capability.extend mc* (‖ *tag* = *False* ‖)) *nth-frag*)
⟨*proof*⟩

lemma *load-bytes-on-capbytes-fail*:
assumes *tag c* = *True*
and *perm-load c* = *True*
and *offset c* + $|t|_\tau$ ≤ *base c* + *len c*
and *offset c* ≥ *base c*
and *offset c mod* $|t|_\tau$ = *0*
and *Mapping.lookup* (*heap-map h*) (*block-id c*) = *Some* (*Map m*)
and *offset c* ≥ *fst* (*bounds m*)
and *offset c* + $|t|_\tau$ ≤ *snd* (*bounds m*)
and ¬ *is-contiguous-bytes* (*content m*) (*nat* (*offset c*)) $|t|_\tau$
and *is-cap* (*content m*) (*nat* (*offset c*))
and *mc* = *get-cap* (*content m*) (*nat* (*offset c*))
and *is-contiguous-cap* (*content m*) *mc* (*nat* (*offset c*)) $|t|_\tau$
and *t* ≠ *Cap*
and *t* ≠ *Uint8*
and *t* ≠ *Sint8*
shows *load h c t* = *Success Undef*
⟨*proof*⟩

lemma *load-cap-on-capbytes*:
assumes *tag c* = *True*
and *perm-load c* = *True*
and *offset c* + $|t|_\tau$ ≤ *base c* + *len c*
and *offset c* ≥ *base c*
and *offset c mod* $|t|_\tau$ = *0*
and *Mapping.lookup* (*heap-map h*) (*block-id c*) = *Some* (*Map m*)
and *offset c* ≥ *fst* (*bounds m*)
and *offset c* + $|t|_\tau$ ≤ *snd* (*bounds m*)
and ¬ *is-contiguous-bytes* (*content m*) (*nat* (*offset c*)) $|t|_\tau$
and *is-cap* (*content m*) (*nat* (*offset c*))

and $mc = get\text{-}cap\ (content\ m)\ (nat\ (offset\ c))$
and $is\text{-}contiguous\text{-}cap\ (content\ m)\ mc\ (nat\ (offset\ c))\ |t|_\tau$
and $t = Cap$
and $tagval = the\ (Mapping.lookup\ (tags\ m)\ (nat\ (offset\ c)))$
and $tg = (case\ perm\text{-}cap\text{-}load\ c\ of\ False \Rightarrow False \mid True \Rightarrow tagval)$
shows $load\ h\ c\ t = Success\ (Cap\text{-}v\ (mem\text{-}capability.extend\ mc\ (\!|\ tag = tg\ |\!)))$
⟨*proof*⟩

lemma *load-cond-hard-cap*:
assumes $load\ h\ c\ t = Success\ ret$
shows $tag\ c = True$
and $perm\text{-}load\ c = True$
and $offset\ c + |t|_\tau \leq base\ c + len\ c$
and $offset\ c \geq base\ c$
and $offset\ c\ mod\ |t|_\tau = 0$
⟨*proof*⟩

lemma *load-cond-bytes*:
assumes $load\ h\ c\ t = Success\ ret$
and $ret \neq Undef$
and $\forall\ x.\ ret \neq Cap\text{-}v\ x$
and $\forall\ x\ n\ .\ ret \neq Cap\text{-}v\text{-}frag\ x\ n$
shows $\exists\ m.\ Mapping.lookup\ (heap\text{-}map\ h)\ (block\text{-}id\ c) = Some\ (Map\ m)$
$\wedge\ offset\ c \geq fst\ (bounds\ m)$
$\wedge\ offset\ c + |t|_\tau \leq snd\ (bounds\ m)$
$\wedge\ is\text{-}contiguous\text{-}bytes\ (content\ m)\ (nat\ (offset\ c))\ |t|_\tau$
⟨*proof*⟩

lemma *load-cond-cap*:
assumes $load\ h\ c\ t = Success\ ret$
and $\exists\ x.\ ret = Cap\text{-}v\ x$
shows $\exists\ m\ mc\ tagval\ tg.$
$Mapping.lookup\ (heap\text{-}map\ h)\ (block\text{-}id\ c) = Some\ (Map\ m) \wedge$
$offset\ c \geq fst\ (bounds\ m) \wedge$
$offset\ c + |t|_\tau \leq snd\ (bounds\ m) \wedge$
$(is\text{-}contiguous\text{-}bytes\ (content\ m)\ (nat\ (offset\ c))\ |t|_\tau \longrightarrow$
$is\text{-}contiguous\text{-}zeros\ (content\ m)\ (nat\ (offset\ c))\ |t|_\tau \wedge$
$ret = Cap\text{-}v\ NULL) \wedge$
$(\neg\ is\text{-}contiguous\text{-}bytes\ (content\ m)\ (nat\ (offset\ c))\ |t|_\tau \longrightarrow$
$is\text{-}cap\ (content\ m)\ (nat\ (offset\ c)) \wedge$
$mc = get\text{-}cap\ (content\ m)\ (nat\ (offset\ c)) \wedge$
$is\text{-}contiguous\text{-}cap\ (content\ m)\ mc\ (nat\ (offset\ c))\ |t|_\tau \wedge$
$t = Cap \wedge$
$tagval = the\ (Mapping.lookup\ (tags\ m)\ (nat\ (offset\ c))) \wedge$
$tg = (case\ perm\text{-}cap\text{-}load\ c\ of\ False \Rightarrow False \mid True \Rightarrow tagval))$
⟨*proof*⟩

lemma *load-cond-cap-frag*:

assumes *load h c t = Success ret*
and $\exists$ *x n. ret = Cap-v-frag x n*
shows $\exists$ *m mc tagval tg nth-frag.*
Mapping.lookup (heap-map h) (block-id c) = Some (Map m) ∧
offset c ≥ fst (bounds m) ∧
offset c + |t|$_\tau$ ≤ snd (bounds m) ∧
(is-contiguous-bytes (content m) (nat (offset c)) |t|$_\tau$ ⟶
is-contiguous-zeros (content m) (nat (offset c)) |t|$_\tau$ ∧
ret = Cap-v NULL) ∧
(¬ is-contiguous-bytes (content m) (nat (offset c)) |t|$_\tau$ ⟶
is-cap (content m) (nat (offset c)) ∧
mc = get-cap (content m) (nat (offset c)) ∧
(t = Uint8 ∨ t = Sint8) ∧
tagval = the (Mapping.lookup (tags m) (nat (offset c))) ∧
tg = (case perm-cap-load c of False ⇒ False | True ⇒ tagval) ∧
nth-frag = of-nth (the (Mapping.lookup (content m) (nat (offset c)))))
⟨*proof*⟩

lemma *store-null-error*:
store h NULL v = Error (C2Err TagViolation)
⟨*proof*⟩

lemma *store-false-tag*:
assumes *tag c = False*
shows *store h c v = Error (C2Err TagViolation)*
⟨*proof*⟩

lemma *store-false-perm-store*:
assumes *tag c = True*
and *perm-store c = False*
shows *store h c v = Error (C2Err PermitStoreViolation)*
⟨*proof*⟩

lemma *store-cap-false-perm-cap-store*:
assumes *tag c = True*
and *perm-store c = True*
and *perm-cap-store c = False*
and $\exists$ *cv. v = Cap-v cv ∧ tag cv = True*
shows *store h c v = Error (C2Err PermitStoreCapViolation)*
⟨*proof*⟩

lemma *store-cap-false-perm-cap-store-local*:
assumes *tag c = True*
and *perm-store c = True*
and *perm-cap-store c = True*
and *perm-cap-store-local c = False*
and $\exists$ *cv. v = Cap-v cv ∧ tag cv = True ∧ perm-global cv = False*
shows *store h c v = Error (C2Err PermitStoreLocalCapViolation)*

⟨*proof*⟩

lemma *store-bound-over*:
assumes *tag c* = *True*
and *perm-store c* = *True*
and ⋀ *cv*. ⟦ *v* = *Cap-v cv*; *tag cv* ⟧ ⟹ *perm-cap-store c* ∧ (*perm-cap-store-local c* ∨ *perm-global cv*)
and *offset c* + |*memval-type v*|$_\tau$ > *base c* + *len c*
shows *store h c v* = *Error* (*C2Err LengthViolation*)
⟨*proof*⟩

lemma *store-bound-under*:
assumes *tag c* = *True*
and *perm-store c* = *True*
and ⋀ *cv*. ⟦ *v* = *Cap-v cv*; *tag cv* ⟧ ⟹ *perm-cap-store c* ∧ (*perm-cap-store-local c* ∨ *perm-global cv*)
and *offset c* + |*memval-type v*|$_\tau$ ≤ *base c* + *len c*
and *offset c* < *base c*
shows *store h c v* = *Error* (*C2Err LengthViolation*)
⟨*proof*⟩

lemma *store-misaligned*:
assumes *tag c* = *True*
and *perm-store c* = *True*
and ⋀ *cv*. ⟦ *v* = *Cap-v cv*; *tag cv* ⟧ ⟹ *perm-cap-store c* ∧ (*perm-cap-store-local c* ∨ *perm-global cv*)
and *offset c* + |*memval-type v*|$_\tau$ ≤ *base c* + *len c*
and *offset c* ≥ *base c*
and *offset c mod* |*memval-type v*|$_\tau$ ≠ *0*
shows *store h c v* = *Error* (*C2Err BadAddressViolation*)
⟨*proof*⟩

lemma *store-undef-val*:
assumes *tag c* = *True*
and *perm-store c* = *True*
and ⋀ *cv*. ⟦ *v* = *Cap-v cv*; *tag cv* ⟧ ⟹ *perm-cap-store c* ∧ (*perm-cap-store-local c* ∨ *perm-global cv*)
and *offset c* + |*memval-type v*|$_\tau$ ≤ *base c* + *len c*
and *offset c* ≥ *base c*
and *offset c mod* |*memval-type v*|$_\tau$ = *0*
and *v* = *Undef*
shows *store h c v* = *Error* (*LogicErr* (*Unhandled 0*))
⟨*proof*⟩

lemma *store-nonexistant-obj*:
assumes *tag c* = *True*
and *perm-store c* = *True*
and ⋀ *cv*. ⟦ *v* = *Cap-v cv*; *tag cv* ⟧ ⟹ *perm-cap-store c* ∧ (*perm-cap-store-local c* ∨ *perm-global cv*)

and *offset c* + |*memval-type v*|$_\tau$ ≤ *base c* + *len c*
and *offset c* ≥ *base c*
and *offset c mod* |*memval-type v*|$_\tau$ = *0*
and *v* ≠ *Undef*
and *Mapping.lookup* (*heap-map h*) (*block-id c*) = *None*
shows *store h c v* = *Error* (*LogicErr MissingResource*)
⟨*proof*⟩

lemma *store-store-after-free*:
assumes *tag c* = *True*
and *perm-store c* = *True*
and ⋀ *cv*. ⟦ *v* = *Cap-v cv*; *tag cv* ⟧ ⟹ *perm-cap-store c* ∧ (*perm-cap-store-local c* ∨ *perm-global cv*)
and *offset c* + |*memval-type v*|$_\tau$ ≤ *base c* + *len c*
and *offset c* ≥ *base c*
and *offset c mod* |*memval-type v*|$_\tau$ = *0*
and *v* ≠ *Undef*
and *Mapping.lookup* (*heap-map h*) (*block-id c*) = *Some Freed*
shows *store h c v* = *Error* (*LogicErr UseAfterFree*)
⟨*proof*⟩

lemma *store-bound-violated-1*:
assumes *tag c* = *True*
and *perm-store c* = *True*
and ⋀ *cv*. ⟦ *v* = *Cap-v cv*; *tag cv* ⟧ ⟹ *perm-cap-store c* ∧ (*perm-cap-store-local c* ∨ *perm-global cv*)
and *offset c* + |*memval-type v*|$_\tau$ ≤ *base c* + *len c*
and *offset c* ≥ *base c*
and *offset c mod* |*memval-type v*|$_\tau$ = *0*
and *v* ≠ *Undef*
and *Mapping.lookup* (*heap-map h*) (*block-id c*) = *Some* (*Map m*)
and *offset c* < *fst* (*bounds m*)
shows *store h c v* = *Error* (*LogicErr BufferOverrun*)
⟨*proof*⟩

lemma *store-bound-violated-2*:
assumes *tag c* = *True*
and *perm-store c* = *True*
and ⋀ *cv*. ⟦ *v* = *Cap-v cv*; *tag cv* ⟧ ⟹ *perm-cap-store c* ∧ (*perm-cap-store-local c* ∨ *perm-global cv*)
and *offset c* + |*memval-type v*|$_\tau$ ≤ *base c* + *len c*
and *offset c* ≥ *base c*
and *offset c mod* |*memval-type v*|$_\tau$ = *0*
and *v* ≠ *Undef*
and *Mapping.lookup* (*heap-map h*) (*block-id c*) = *Some* (*Map m*)
and *offset c* + |*memval-type v*|$_\tau$ > *snd* (*bounds m*)
shows *store h c v* = *Error* (*LogicErr BufferOverrun*)
⟨*proof*⟩

lemma *store-success*:
assumes *tag c = True*
and *perm-store c = True*
and $\bigwedge$ *cv*. ⟦ *v = Cap-v cv*; *tag cv* ⟧ ⟹ *perm-cap-store c* ∧ (*perm-cap-store-local c* ∨ *perm-global cv*)
and *offset c* + |*memval-type v*|$_\tau$ ≤ *base c* + *len c*
and *offset c* ≥ *base c*
and *offset c mod* |*memval-type v*|$_\tau$ = *0*
and *v* ≠ *Undef*
and *Mapping.lookup* (*heap-map h*) (*block-id c*) = *Some* (*Map m*)
and *offset c* ≥ *fst* (*bounds m*)
and *offset c* + |*memval-type v*|$_\tau$ ≤ *snd* (*bounds m*)
shows ∃ *ret*. *store h c v = Success ret* ∧
next-block ret = next-block h ∧
heap-map ret = Mapping.update (*block-id c*) (*Map* (*store-tval m* (*nat* (*offset c*)) *v*)) (*heap-map h*)
⟨*proof*⟩

lemma *store-cond-hard-cap*:
assumes *store h c v = Success ret*
shows *tag c = True*
and *perm-store c = True*
and $\bigwedge$ *cv*. ⟦ *v = Cap-v cv*; *tag cv* ⟧ ⟹ *perm-cap-store c* ∧ (*perm-cap-store-local c* ∨ *perm-global cv*)
and *offset c* + |*memval-type v*|$_\tau$ ≤ *base c* + *len c*
and *offset c* ≥ *base c*
and *offset c mod* |*memval-type v*|$_\tau$ = *0*
⟨*proof*⟩

lemma *store-cond-bytes-bounds*:
assumes *store h c val = Success h′*
and ∀ *x*. *val* ≠ *Cap-v x*
shows ∃ *m*. *Mapping.lookup* (*heap-map h*) (*block-id c*) = *Some* (*Map m*)
∧ *offset c* ≥ *fst* (*bounds m*)
∧ *offset c* + |*memval-type val*|$_\tau$ ≤ *snd* (*bounds m*)
⟨*proof*⟩

lemma *store-cond-bytes*:
assumes *store h c val = Success h′*
and ∀ *x*. *val* ≠ *Cap-v x*
shows ∃ *m*. *Mapping.lookup* (*heap-map h′*) (*block-id c*) = *Some* (*Map m*)
∧ *offset c* ≥ *fst* (*bounds m*)
∧ *offset c* + |*memval-type val*|$_\tau$ ≤ *snd* (*bounds m*)
⟨*proof*⟩

lemma *store-cond-cap-bounds*:
assumes *store h c val = Success h′*
and *val = Cap-v x*
shows ∃ *m*. *Mapping.lookup* (*heap-map h*) (*block-id c*) = *Some* (*Map m*)

$\wedge$ *offset c* $\geq$ *fst* (*bounds m*)
$\wedge$ *offset c* + |*memval-type val*|$_\tau$ $\leq$ *snd* (*bounds m*)
⟨*proof*⟩

lemma *store-cond-cap*:
assumes *store h c val* = *Success h'*
and *val* = *Cap-v v*
shows $\exists$ *m*. *Mapping.lookup* (*heap-map h'*) (*block-id c*) = *Some* (*Map m*)
$\wedge$ *offset c* $\geq$ *fst* (*bounds m*)
$\wedge$ *offset c* + |*memval-type val*|$_\tau$ $\leq$ *snd* (*bounds m*)
⟨*proof*⟩

lemma *store-cond*:
assumes *store h c val* = *Success h'*
shows $\exists$ *m*. *Mapping.lookup* (*heap-map h'*) (*block-id c*) = *Some* (*Map m*)
$\wedge$ *offset c* $\geq$ *fst* (*bounds m*)
$\wedge$ *offset c* + |*memval-type val*|$_\tau$ $\leq$ *snd* (*bounds m*)
⟨*proof*⟩

lemma *store-bounds-preserved*:
assumes *store h c v* = *Success h'*
and *Mapping.lookup* (*heap-map h*) (*block-id c*) = *Some* (*Map m*)
and *Mapping.lookup* (*heap-map h'*) (*block-id c*) = *Some* (*Map m'*)
shows *bounds m* = *bounds m'*
⟨*proof*⟩

lemma *store-cond-cap-frag*:
assumes *store h c val* = *Success h'*
and *val* = *Cap-v-frag v n*
shows $\exists$ *m*. *Mapping.lookup* (*heap-map h'*) (*block-id c*) = *Some* (*Map m*)
⟨*proof*⟩

lemma *store-undef-false*:
assumes *store h c Undef* = *Success ret*
shows *False*
⟨*proof*⟩

lemma *load-after-alloc-size-fail*:
assumes *alloc h c s* = *Success* (*h'*, *cap*)
and |*t*|$_\tau$ > *s*
shows *load h' cap t* = *Error* (*C2Err LengthViolation*)
⟨*proof*⟩

5.1.2 Good Variable Laws

The properties defined above are intermediate results. Properties that govern the correctness while executing are the *good variable* laws. The most important ones are:

- load after alloc
- load after free
- load after store

The *load after store* case requires particular attention. For disjoint cases within the same block (refer to load_after_store_disjoint_2 and load_after_store_disjoint_3), extra attention must be paid to the tagged memory, where the updated tag may occur at a location specified *before* whatever was given by the capability offset. This is why the lemma retrieve_stored_tval_disjoint_2 requires an additional constraint where capability values are aligned. Of course, this is not a problem for load_after_store_disjoint_3 since the capability conditions state that offsets must be aligned.

For the compatible case, as stated in the paper [6], extra care has to be put in the cases where we load capabilities and capability fragments. For this, we have three cases:

- load_after_store_prim
- load_after_store_cap
- load_after_store_cap_frag

The load_after_store_prim case returns the exact value that was stored. The load_after_store_cap case returns the stored capability with the tag bit dependent on the permissions of the capability provided to load. Finally, the load_after_store_cap_frag case returns the capability fragment with the tag bit falsifed.

Finally, we note that there are slight differences to the CompCert version of the Good Variable Law due to the differences in the type and value system. Thus, there are cases in the CompCert version that are trivial in our case.

theorem *load-after-alloc-1*:
  assumes *alloc h c s* = *Success* (*h′*, *cap*)
    and $|t|_\tau \leq s$
  shows *load h′ cap t* = *Success Undef*
⟨*proof*⟩

theorem *load-after-alloc-2*:
  assumes *alloc h c s* = *Success* (*h′*, *cap*)
    and $|t|_\tau \leq s$
    and *block-id cap* ≠ *block-id cap′*
  shows *load h′ cap′ t* = *load h cap′ t*
  ⟨*proof*⟩

theorem *load-after-free-1*:

assumes *free h c = Success (h', cap)*
shows *load h cap t = Error (C2Err TagViolation)*
⟨*proof*⟩

theorem *load-after-free-2*:
assumes *free h c = Success (h', cap)*
and *block-id cap ≠ block-id cap'*
shows *load h cap' t = load h' cap' t*
⟨*proof*⟩

theorem *load-after-store-disjoint-1*:
assumes *store h c val = Success h'*
and *block-id c ≠ block-id c'*
shows *load h c' t = load h' c' t*
⟨*proof*⟩

theorem *load-after-store-disjoint-2*:
assumes *store h c v = Success h'*
and *offset c' + |t|$_\tau$ ≤ offset c*
shows *load h' c' t = load h c' t*
⟨*proof*⟩

theorem *load-after-store-disjoint-3*:
assumes *store h c v = Success h'*
and *offset c + |memval-type v|$_\tau$ ≤ offset c'*
shows *load h' c' t = load h c' t*
⟨*proof*⟩

theorem *load-after-store-prim*:
assumes *store h c val = Success h'*
and *∀ v. val ≠ Cap-v v*
and *∀ v n. val ≠ Cap-v-frag v n*
and *perm-load c = True*
shows *load h' c (memval-type val) = Success val*
⟨*proof*⟩

theorem *load-after-store-cap*:
assumes *store h c (Cap-v v) = Success h'*
and *perm-load c = True*
shows *load h' c (memval-type (Cap-v v)) = Success (Cap-v (v (| tag := case perm-cap-load c of False => False | True => tag v |)))*
⟨*proof*⟩

theorem *load-after-store-cap-frag*:
assumes *store h c (Cap-v-frag c' n) = Success h'*
and *perm-load c*
shows *load h' c (memval-type (Cap-v-frag c' n)) = Success (Cap-v-frag (c' (| tag := False |)) n)*
⟨*proof*⟩

5.1.3 Miscellaneous Laws

lemma *free-after-alloc*:
assumes *alloc h c s = Success (h', cap)*
shows ∃! *ret. free h' cap = Success ret*
⟨*proof*⟩

lemma *store-after-alloc*:
assumes *alloc h True s = Success (h', cap)*
and $|memval\text{-}type\ v|_\tau \leq s$
and $v \neq Undef$
shows ∃ *h''. store h' cap v = Success h''*
⟨*proof*⟩

lemma *store-after-alloc-gen*:
assumes *alloc h True s = Success (h', cap)*
and $|memval\text{-}type\ v|_\tau \leq s$
and $v \neq Undef$
and $n\ mod\ |memval\text{-}type\ v|_\tau = 0$
and $offset\ cap + n + |memval\text{-}type\ v|_\tau \leq base\ cap + len\ cap$
shows ∃ *h''. store h' (cap (| offset := offset cap + n |)) v = Success h''*
⟨*proof*⟩

5.2 Well-formedness of actions

lemma *alloc-wellformed*:
assumes $\mathcal{W}_\mathfrak{f}(heap\text{-}map\ h)$
and *alloc h True s = Success (h', cap)*
shows $\mathcal{W}_\mathfrak{f}(heap\text{-}map\ h')$
⟨*proof*⟩

lemma *free-wellformed*:
assumes $\mathcal{W}_\mathfrak{f}(heap\text{-}map\ h)$
and *free h cap = Success (h', cap')*
shows $\mathcal{W}_\mathfrak{f}(heap\text{-}map\ h')$
⟨*proof*⟩

lemma *load-wellformed*:
assumes $\mathcal{W}_\mathfrak{f}(heap\text{-}map\ h)$
and *load h c t = Success v*
shows $\mathcal{W}_\mathfrak{f}(heap\text{-}map\ h)$
⟨*proof*⟩

lemma *store-wellformed*:
assumes $\mathcal{W}_\mathfrak{f}(heap\text{-}map\ h)$
and *store h c v = Success h'*
shows $\mathcal{W}_\mathfrak{f}(heap\text{-}map\ h')$
⟨*proof*⟩

5.3 memcpy formalisation

We also formalise memcpy in Isabelle/HOL. While other higher level operations are defined in the GIL level, we formalise memcpy here and prove basic properties. memcpy works as follows: we define a mutually recursive function memcpy_prim and memcpy_cap. memcpy_prim attempts byte copies, where tags are invalidated, and memcpy_cap attempts capability copies. memcpy initially calls memcpy_cap. If either load or store fails, perhaps due to misalignment or other issues, memcpy_prim will be called instead. If memcpy_prim also fails from load or store, the operation will fail.

function *memcpy-prim* :: *heap* ⇒ *cap* ⇒ *cap* ⇒ *nat* ⇒ *heap result*
and *memcpy-cap* :: *heap* ⇒ *cap* ⇒ *cap* ⇒ *nat* ⇒ *heap result*
where
memcpy-prim h - - 0 = Success h
| *memcpy-cap h - - 0 = Success h*
| *memcpy-prim h dst src (Suc n) =*
(*let x = load h src Uint8 in*
if ¬ is-Success x then Error (err x) else
let xs = res x in
if xs = Undef then Error (LogicErr (Unhandled 0)) else
let y = store h dst xs in
if ¬ is-Success y then Error (err y) else
let ys = res y in
memcpy-cap ys (dst (| offset := (offset dst + 1) |)) (src (| offset := (offset src) + 1 |)) n)
| *memcpy-cap h dst src (Suc n) =*
(*if (Suc n) < |Cap|$_\tau$ then memcpy-prim h dst src (Suc n)*
else
let x = load h src Cap in
if ¬ is-Success x then memcpy-prim h dst src (Suc n) else
let xs = res x in
if xs = Undef then memcpy-prim h dst src (Suc n) else
let y = store h dst xs in
if ¬ is-Success y then memcpy-prim h dst src (Suc n) else
let ys = res y in
memcpy-cap ys (dst (| offset := (offset dst + |Cap|$_\tau$) |)) (src (| offset := (offset src + |Cap|$_\tau$) |)) (Suc n − |Cap|$_\tau$))
⟨*proof*⟩

We prove that the mutually recursive function terminates.

context
notes *sizeof-def*[*simp*]
begin
termination ⟨*proof*⟩
end

This is the definition of memcpy. We also check that src and dst do not

overlap.

definition *memcpy* :: *heap* ⇒ *cap* ⇒ *cap* ⇒ *nat* ⇒ *heap result*
where
memcpy h dst src n ≡
if n = *0 then*
Success h
else if block-id dst = *block-id src* ∧
((*offset src* ≥ *offset dst* ∧ *offset src* < *offset dst* + *n*) ∨
(*offset dst* ≥ *offset src* ∧ *offset dst* < *offset src* + *n*)) *then*
Error (*LogicErr* (*Unhandled 0*))
else memcpy-cap h dst src n

lemma *memcpy-rec-wellformed*:
assumes $\mathcal{W}_{\mathfrak{f}}$(*heap-map h*)
shows *memcpy-prim h dst src n* = *Success h*′ ⟹ $\mathcal{W}_{\mathfrak{f}}$(*heap-map h*′)
and *memcpy-cap h dst src n* = *Success h*′ ⟹ $\mathcal{W}_{\mathfrak{f}}$(*heap-map h*′)
⟨*proof*⟩

We also prove that memcpy preserves well-formedness.

lemma *memcpy-wellformed*:
assumes $\mathcal{W}_{\mathfrak{f}}$(*heap-map h*)
and *memcpy h dst src n* = *Success h*′
shows $\mathcal{W}_{\mathfrak{f}}$(*heap-map h*′)
⟨*proof*⟩

lemma *memcpy-cond*:
assumes *memcpy h dst src n* = *Success h*′
shows *n* > *0* ⟶ ¬ (*block-id dst* = *block-id src* ∧
((*offset src* ≥ *offset dst* ∧ *offset src* < *offset dst* + *n*) ∨
(*offset dst* ≥ *offset src* ∧ *offset dst* < *offset src* + *n*)))
⟨*proof*⟩

6 Miscellaneous Definitions

The following are used for catching memory leaks in Gillian.

definition *get-block-size* :: *heap* ⇒ *block* ⇒ *nat option*
where
get-block-size h b ≡
let ex = *Mapping.lookup* (*heap-map h*) *b in*
(*case ex of None* ⇒ *None* | *Some m* ⇒
(*case m of Freed* ⇒ *None* | - ⇒ *Some* (*snd* (*bounds* (*the-map m*)))))

primrec *get-memory-leak-size* :: *heap* ⇒ *nat* ⇒ *nat*
where
get-memory-leak-size - 0 = *0*
| *get-memory-leak-size h* (*Suc n*) = *get-memory-leak-size h n* +
(*case get-block-size h* (*integer-of-nat* (*Suc n*)) *of*

```
    None ⇒ 0
  | Some n ⇒ n)

primrec get-unfreed-blocks :: heap ⇒ nat ⇒ block list
  where
  get-unfreed-blocks - 0 = []
| get-unfreed-blocks h (Suc n) =
    (let ex = Mapping.lookup (heap-map h) (integer-of-nat (Suc n)) in
    (case ex of None ⇒ get-unfreed-blocks h n | Some m ⇒
     (case m of Freed ⇒ get-unfreed-blocks h n | - ⇒ integer-of-nat (Suc n) #
get-unfreed-blocks h n)))

end
theory CHERI-C-Global-Environment
  imports CHERI-C-Concrete-Memory-Model
begin
```

Here, we define the global environment. The Global Environment does the following:

1. Creates a mapping from variables to locations (or rather, the capabilities)

2. Sets global variables by invoking alloc. These variables cannot be freed by design

```
type-synonym genv = (String.literal, cap) mapping

definition alloc-glob-var :: heap ⇒ bool ⇒ nat ⇒ (heap × cap) result
  where
  alloc-glob-var h c s ≡
    let h' = alloc h c s in
    Success (fst (res h'), snd (res h') (| perm-global := True |))

definition set-glob-var :: heap ⇒ bool ⇒ nat ⇒ String.literal ⇒ genv ⇒ (heap ×
cap × genv) result
  where
  set-glob-var h c s v g ≡
    let (h', cap) = res (alloc-glob-var h c s) in
    let g' = Mapping.update v cap g in
    Success (h', cap, g')

lemma set-glob-var-glob-bit:
  assumes alloc-glob-var h c s = Success (h', cap)
  shows perm-global cap
  ⟨proof⟩

lemma set-glob-var-glob-bit-lift:
  assumes set-glob-var h c s v g = Success (h', cap, g')
```

shows *perm-global cap*
⟨*proof*⟩

lemma *free-fails-on-glob-var*:
assumes *alloc-glob-var h c s* = *Success* (*h′*, *cap*)
shows *free h′ cap* = *Error* (*LogicErr* (*Unhandled 0*))
⟨*proof*⟩

lemma *free-fails-on-glob-lift*:
assumes *set-glob-var h c s v g* = *Success* (*h′*, *cap*, *g′*)
shows *free h′ cap* = *Error* (*LogicErr* (*Unhandled 0*))
⟨*proof*⟩

7 Code Generation

Here we generate an OCaml instance of the memory model that will be used for Gillian.

export-code
null-capability init-heap next-block get-memory-leak-size get-unfreed-blocks

alloc free load store
memcpy
set-glob-var
word8-of-integer word16-of-integer word32-of-integer word64-of-integer

integer-of-word8 integer-of-word16 integer-of-word32 integer-of-word64
sword8-of-integer sword16-of-integer sword32-of-integer sword64-of-integer
integer-of-sword8 integer-of-sword16 integer-of-sword32 integer-of-sword64
integer-of-nat cast-val
C2Err LogicErr
TagViolation PermitLoadViolation PermitStoreViolation PermitStoreCapViolation
PermitStoreLocalCapViolation LengthViolation BadAddressViolation
UseAfterFree BufferOverrun MissingResource WrongMemVal MemoryNotFreed Unhandled
in *OCaml*
file-prefix *CHERI-C-Memory-Model*

end

References

[1] J. Fragoso Santos, P. Maksimović, S.-E. Ayoun, and P. Gardner. Gillian, Part i: A Multi-Language Platform for Symbolic Execution. In *Proceed-*

ings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2020, page 927942, New York, NY, USA, 2020. Association for Computing Machinery.

[2] B. S. Institution and B. T. B. S. Institution). *The C Standard: Incorporating Technical Corrigendum 1*. Wiley, 2003.

[3] X. Leroy, A. W. Appel, S. Blazy, and G. Stewart. The CompCert Memory Model, Version 2. Research Report RR-7987, INRIA, Jun 2012.

[4] P. Maksimovic, S.-E. Ayoun, J. F. Santos, and P. Gardner. Gillian, part II: real-world verification for javascript and C. In A. Silva and K. R. M. Leino, editors, *Proceedings of the 33rd Computer Aided Verification International Conference, CAV 2021, Virtual Event, July 20-23, 2021, Part II*, volume 12760 of *Lecture Notes in Computer Science*, pages 827–850. Springer, 2021.

[5] P. Maksimovic, J. F. Santos, S.-E. Ayoun, and P. Gardner. Gillian: A Multi-Language Platform for Unified Symbolic Analysis, 2021.

[6] S. H. Park, R. Pai, and T. Melham. A Formal CHERI-C Semantics for Verification, 2022. Submitted to TACAS 2023.

[7] J. F. Santos, P. Maksimovic, S.-E. Ayoun, and P. Gardner. Gillian: Compositional Symbolic Execution for All. *CoRR*, abs/2001.05059, 2020.

[8] R. N. M. Watson, A. Richardson, B. Davis, J. Baldwin, D. Chisnall, J. Clarke, N. Filardo, S. M. Moore, E. Napierala, P. Sewell, and P. G. Neumann. CHERI C/C++ Programming Guide. Technical report, University of Cambridge, Cambridge, England, Jun 2020.