

A Formal CHERI-C Memory Model

Seung Hoon Park

March 24, 2023

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	16
3	Memory	21
3.1	Definitions	21
3.2	Properties	23
4	Helper functions and lemmas	28
5	Memory Actions / Operations	66
5.1	Properties of the operations	69
5.1.1	Correctness Properties	69
5.1.2	Good Variable Laws	93
5.1.3	Miscellaneous Laws	99
5.2	Well-formedness of actions	101
5.3	memcpy formalisation	103

6 Miscellaneous Definitions	105
7 Code Generation	107

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 cval corresponds to \mathcal{V}_C in the paper [6]. 'a in this instance must be countable.

```
datatype 'a cval =
  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 cval  $\Rightarrow$  cctype
where
  memval-type v = (case v of
    Uint8-v -  $\Rightarrow$  Uint8
  | Sint8-v -  $\Rightarrow$  Sint8
  | Uint16-v -  $\Rightarrow$  Uint16
  | Sint16-v -  $\Rightarrow$  Sint16
  | Uint32-v -  $\Rightarrow$  Uint32
  | Sint32-v -  $\Rightarrow$  Sint32
  | Uint64-v -  $\Rightarrow$  Uint64
  | Sint64-v -  $\Rightarrow$  Sint64
  | Cap-v -  $\Rightarrow$  Cap
  | Cap-v-frag - -  $\Rightarrow$  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  $\Rightarrow$  8 word
where
  encode-u8 x  $\equiv$  word-of-nat x
```

```
abbreviation decode-u8 :: 8 word  $\Rightarrow$  nat
where
  decode-u8 b  $\equiv$  unat b
```

```
abbreviation encode-u8-list :: 8 word  $\Rightarrow$  8 word list
where
```

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

abbreviation $decode-u8-list :: 8\ word\ list \Rightarrow 8\ word$

where

$decode-u8-list\ ls \equiv hd\ ls$

lemma $encode-decode-u8-list:$

$ls = [b] \implies ls = encode-u8-list\ (decode-u8-list\ ls)$

by $simp$

lemma $decode-encode-u8-list:$

$w = decode-u8-list\ (encode-u8-list\ w)$

by $simp$

lemma $encode-decode-u8:$

$w = encode-u8\ (decode-u8\ w)$

by $simp$

lemma $decode-encode-u8:$

assumes $i \leq 2 \wedge LENGTH(8) - 1$

shows $i = decode-u8\ (encode-u8\ i)$

by $(metis\ assms\ le-unat-voi\ unat-minus-one-word)$

abbreviation $u64-split :: 64\ word \Rightarrow 32\ word\ list$

where

$u64-split\ x \equiv (word-rsplit :: 64\ word \Rightarrow 32\ word\ list)\ x$

abbreviation $u32-split :: 32\ word \Rightarrow 16\ word\ list$

where

$u32-split\ x \equiv (word-rsplit :: 32\ word \Rightarrow 16\ word\ list)\ x$

abbreviation $u16-split :: 16\ word \Rightarrow 8\ word\ list$

where

$u16-split\ x \equiv (word-rsplit :: 16\ word \Rightarrow 8\ word\ list)\ x$

abbreviation $cat-u16 :: 8\ word\ list \Rightarrow 16\ word$

where

$cat-u16\ x \equiv (word-rcat :: 8\ word\ list \Rightarrow 16\ word)\ x$

abbreviation $encode-u16 :: nat \Rightarrow 8\ word\ list$

where

$encode-u16\ x \equiv u16-split\ (word-of-nat\ x)$

abbreviation $decode-u16 :: 8\ word\ list \Rightarrow nat$

where

$decode-u16\ x \equiv unat\ (cat-u16\ x)$

lemma $flatten-u16-length:$

$length\ (u16-split\ vs) = 2$

by (simp add: length-word-rsplit-even-size wsst-TYs(3))

lemma *rsplit-rcat-eq*:

assumes $LENGTH('b::len) \bmod LENGTH('a::len) = 0$

and $length\ w = LENGTH('b) \div LENGTH('a)$

shows $(word-rsplit :: 'b\ word \Rightarrow 'a\ word\ list) ((word-rcat :: 'a\ word\ list \Rightarrow 'b\ word)\ w) = w$

by (simp add: assms mod-0-imp-dvd size-word.rep-eq word-rsplit-rcat-size)

lemma *rsplit-rcat-u16-eq*:

assumes $w = [a1, a2]$

shows $(word-rsplit :: 16\ word \Rightarrow 8\ word\ list) ((word-rcat :: 8\ word\ list \Rightarrow 16\ word)\ w) = w$

proof –

have $l1: length\ w * 8 = 16$

using *assms* by *clarsimp*

moreover have $l2: size\ ((word-rcat :: 8\ word\ list \Rightarrow 16\ word)\ w) = 16$

using *assms*

by (simp add: size-word.rep-eq)

from $l1\ l2$ have $length\ w * 8 = size\ ((word-rcat :: 8\ word\ list \Rightarrow 16\ word)\ w)$

by *argo*

thus *?thesis*

by (*metis* $l1\ l2\ len8\ word-rsplit-rcat-size$)

qed

lemma *encode-decode-u16*:

assumes $w = [a, b]$

shows $w = encode-u16\ (decode-u16\ w)$

by (simp add: assms rsplit-rcat-eq)

lemma *cat-flatten-u16-eq*:

$cat-u16\ (u16-split\ w) = w$

by (simp add: word-rcat-rsplit)

lemma *decode-encode-u16*:

assumes $i \leq 2 \wedge LENGTH(16) - 1$

shows $i = decode-u16\ (encode-u16\ i)$

by (*metis* *assms* *cat-flatten-u16-eq* *le-unat-uoi* *unat-minus-one-word*)

abbreviation *flatten-u32* :: $32\ word \Rightarrow 8\ word\ list$

where

$flatten-u32\ x \equiv (word-rsplit :: 32\ word \Rightarrow 8\ word\ list)\ x$

abbreviation *cat-u32* :: $8\ word\ list \Rightarrow 32\ word$

where

$cat-u32\ x \equiv (word-rcat :: 8\ word\ list \Rightarrow 32\ word)\ x$

abbreviation *encode-u32* :: $nat \Rightarrow 8\ word\ list$

where
 $encode-u32\ x \equiv flatten-u32\ (word-of-nat\ x)$

abbreviation $decode-u32 :: 8\ word\ list \Rightarrow nat$
where
 $decode-u32\ i \equiv unat\ (cat-u32\ i)$

lemma $flatten-u32-length$:
 $length\ (flatten-u32\ vs) = 4$
by (*simp add: length-word-rsplit-even-size wsst-TYs(3)*)

lemma $rsplit-rcat-u32-eq$:
assumes $w = [a1, a2, b1, b2]$
shows $(word-rsplit :: 32\ word \Rightarrow 8\ word\ list) ((word-rcat :: 8\ word\ list \Rightarrow 32\ word)\ w) = w$
using $rsplit-rcat-eq\ assms$
by *force*

lemma $encode-decode-u32$:
assumes $w = [a1, a2, b1, b2]$
shows $w = encode-u32\ (decode-u32\ w)$
using $assms$
by (*simp add: rsplit-rcat-u32-eq*)

lemma $cat-flatten-u32-eq$:
 $cat-u32\ (flatten-u32\ w) = w$
by (*simp add: word-rcat-rsplit*)

lemma $decode-encode-u32$:
assumes $i \leq 2 \wedge LENGTH(32) - 1$
shows $i = decode-u32\ (encode-u32\ i)$
by (*metis assms le-unat-voi unat-minus-one-word word-rcat-rsplit*)

abbreviation $flatten-u64 :: 64\ word \Rightarrow 8\ word\ list$
where
 $flatten-u64\ x \equiv (word-rsplit :: 64\ word \Rightarrow 8\ word\ list)\ x$

abbreviation $cat-u64 :: 8\ word\ list \Rightarrow 64\ word$
where
 $cat-u64\ x \equiv word-rcat\ x$

abbreviation $encode-u64 :: nat \Rightarrow 8\ word\ list$
where
 $encode-u64\ x \equiv flatten-u64\ (word-of-nat\ x)$

abbreviation $decode-u64 :: 8\ word\ list \Rightarrow nat$
where
 $decode-u64\ x \equiv unat\ (cat-u64\ x)$

lemma *flatten-u64-length*:
 $\text{length } (\text{flatten-u64 } vs) = 8$
by (*simp add: length-word-rsplit-even-size wsst-TYs(3)*)

lemma *encode-decode-u64*:
assumes $w = [a1, a2, b1, b2, c1, c2, d1, d2]$
shows $w = \text{encode-u64 } (\text{decode-u64 } w)$
using *assms*
by (*simp add: rsplit-rcat-eq*)

lemma *cat-flatten-u64-eq*:
 $\text{cat-u64 } (\text{flatten-u64 } w) = w$
by (*simp add: word-rcat-rsplit*)

lemma *decode-encode-u64*:
assumes $i \leq 2 \wedge \text{LENGTH}(64) - 1$
shows $i = \text{decode-u64 } (\text{encode-u64 } i)$
by (*metis assms le-unat-uoI unat-minus-one-word word-rcat-rsplit*)

abbreviation *encode-s8* :: $\text{int} \Rightarrow 8 \text{ sword}$
where
 $\text{encode-s8 } x \equiv \text{word-of-int } x$

abbreviation *decode-s8* :: $8 \text{ sword} \Rightarrow \text{int}$
where
 $\text{decode-s8 } b \equiv \text{sint } b$

abbreviation *encode-s8-list* :: $8 \text{ sword} \Rightarrow 8 \text{ word list}$
where
 $\text{encode-s8-list } w \equiv [\text{SCAST}(8 \text{ signed} \rightarrow 8) w]$

abbreviation *decode-s8-list* :: $8 \text{ word list} \Rightarrow 8 \text{ sword}$
where
 $\text{decode-s8-list } ls \equiv \text{UCAST}(8 \rightarrow 8 \text{ signed}) (\text{hd } ls)$

lemma *encode-decode-s8-list*:
 $ls = [b] \implies ls = \text{encode-s8-list } (\text{decode-s8-list } ls)$
by *simp*

lemma *decode-encode-s8-list*:
 $w = \text{decode-s8-list } (\text{encode-s8-list } w)$
by *simp*

lemma *encode-decode-s8*:
 $w = \text{encode-s8 } (\text{decode-s8 } w)$
by *simp*

lemma *decode-encode-s8*:
assumes $-(2 \wedge (\text{LENGTH}(8) - 1)) \leq i$

and $i < 2^{\wedge}(LENGTH(8) - 1)$
shows $i = decode-s8 (encode-s8 i)$
by (*metis assms More-Word.sint-of-int-eq len-signed*)

abbreviation $s64-split :: 64\ sword \Rightarrow 32\ word\ list$
where
 $s64-split\ x \equiv (word-rsplit :: 64\ sword \Rightarrow 32\ word\ list)\ x$

abbreviation $s32-split :: 32\ sword \Rightarrow 16\ word\ list$
where
 $s32-split\ x \equiv (word-rsplit :: 32\ sword \Rightarrow 16\ word\ list)\ x$

abbreviation $s16-split :: 16\ sword \Rightarrow 8\ word\ list$
where
 $s16-split\ x \equiv (word-rsplit :: 16\ sword \Rightarrow 8\ word\ list)\ x$

abbreviation $cat-s16 :: 8\ word\ list \Rightarrow 16\ sword$
where
 $cat-s16\ x \equiv (word-rcat :: 8\ word\ list \Rightarrow 16\ sword)\ x$

abbreviation $encode-s16 :: int \Rightarrow 8\ word\ list$
where
 $encode-s16\ x \equiv s16-split (word-of-int\ x)$

abbreviation $decode-s16 :: 8\ word\ list \Rightarrow int$
where
 $decode-s16\ x \equiv sint (cat-s16\ x)$

lemma *flatten-s16-length*:
 $length (s16-split\ vs) = 2$
by (*simp add: length-word-rsplit-even-size wsst-TYs(3)*)

lemma *rsplit-rcat-s16-eq*:
assumes $w = [a1, a2]$
shows $(word-rsplit :: 16\ sword \Rightarrow 8\ word\ list)\ ((word-rcat :: 8\ word\ list \Rightarrow 16\ sword)\ w) = w$
proof –
have $l1: length\ w * 8 = 16$
using *assms by clarsimp*
moreover **have** $l2: size\ ((word-rcat :: 8\ word\ list \Rightarrow 16\ sword)\ w) = 16$
using *assms*
by (*simp add: size-word.rep-eq*)
from $l1\ l2$ **have** $length\ w * 8 = size\ ((word-rcat :: 8\ word\ list \Rightarrow 16\ sword)\ w)$
by *argo*
thus *?thesis*
by (*simp add: word-rsplit-rcat-size*)
qed

lemma *encode-decode-s16*:
assumes $w = [a, b]$
shows $w = \text{encode-s16 } (\text{decode-s16 } w)$
by (*simp add: assms rsplit-rcat-eq*)

lemma *cat-flatten-s16-eq*:
 $\text{cat-s16 } (\text{s16-split } w) = w$
by (*simp add: word-rcat-rsplit*)

lemma *decode-encode-s16*:
assumes $-(2 \wedge (\text{LENGTH}(16) - 1)) \leq i$
and $i < 2 \wedge (\text{LENGTH}(16) - 1)$
shows $i = \text{decode-s16 } (\text{encode-s16 } i)$
by (*metis assms cat-flatten-s16-eq len-signed sint-of-int-eq*)

abbreviation *flatten-s32* :: $32 \text{ sword} \Rightarrow 8 \text{ word list}$
where
 $\text{flatten-s32 } x \equiv (\text{word-rsplit} :: 32 \text{ sword} \Rightarrow 8 \text{ word list}) x$

abbreviation *cat-s32* :: $8 \text{ word list} \Rightarrow 32 \text{ sword}$
where
 $\text{cat-s32 } x \equiv (\text{word-rcat} :: 8 \text{ word list} \Rightarrow 32 \text{ sword}) x$

abbreviation *encode-s32* :: $\text{int} \Rightarrow 8 \text{ word list}$
where
 $\text{encode-s32 } x \equiv \text{flatten-s32 } (\text{word-of-int } x)$

abbreviation *decode-s32* :: $8 \text{ word list} \Rightarrow \text{int}$
where
 $\text{decode-s32 } i \equiv \text{sint } (\text{cat-s32 } i)$

lemma *flatten-s32-length*:
 $\text{length } (\text{flatten-s32 } vs) = 4$
by (*simp add: length-word-rsplit-even-size wsst-TYs(3)*)

lemma *rsplit-rcat-s32-eq*:
assumes $w = [a1, a2, b1, b2]$
shows $(\text{word-rsplit} :: 32 \text{ sword} \Rightarrow 8 \text{ word list}) ((\text{word-rcat} :: 8 \text{ word list} \Rightarrow 32 \text{ sword}) w) = w$
using *rsplit-rcat-eq assms*
by *force*

lemma *encode-decode-s32*:
assumes $w = [a1, a2, b1, b2]$
shows $w = \text{encode-s32 } (\text{decode-s32 } w)$
using *assms*
by (*simp add: rsplit-rcat-s32-eq*)

lemma *decode-encode-s32*:

assumes $-(2 \wedge (\text{LENGTH}(32) - 1)) \leq i$
and $i < 2 \wedge (\text{LENGTH}(32) - 1)$
shows $i = \text{decode-s32} (\text{encode-s32} i)$
by (*metis assms len-signed sint-of-int-eq word-rcat-rsplit*)

abbreviation *flatten-s64* :: *64 sword* \Rightarrow *8 word list*
where
flatten-s64 $x \equiv (\text{word-rsplit} :: 64 \text{ sword} \Rightarrow 8 \text{ word list}) x$

lemma *flatten-s64-length*:
 $\text{length} (\text{flatten-s64} vs) = 8$
by (*simp add: length-word-rsplit-even-size wsst-TYs(3)*)

abbreviation *cat-s64* :: *8 word list* \Rightarrow *64 sword*
where
cat-s64 $x \equiv \text{word-rcat} x$

abbreviation *encode-s64* :: *int* \Rightarrow *8 word list*
where
encode-s64 $x \equiv \text{flatten-s64} (\text{word-of-int} x)$

abbreviation *decode-s64* :: *8 word list* \Rightarrow *int*
where
decode-s64 $x \equiv \text{sint} (\text{cat-s64} x)$

lemma *encode-decode-s64*:
assumes $w = [a1, a2, b1, b2, c1, c2, d1, d2]$
shows $w = \text{encode-s64} (\text{decode-s64} w)$
using *assms*
by (*simp add: rsplit-rcat-eq*)

lemma *decode-encode-s64*:
assumes $-(2 \wedge (\text{LENGTH}(64) - 1)) \leq i$
and $i < 2 \wedge (\text{LENGTH}(64) - 1)$
shows $i = \text{decode-s64} (\text{encode-s64} i)$
by (*metis assms len-signed sint-of-int-eq word-rcat-rsplit*)

definition *word-of-integer* :: *integer* \Rightarrow *'a::len word*
where
word-of-integer $x \equiv \text{word-of-int} (\text{int-of-integer} x)$

definition *sword-of-integer* :: *integer* \Rightarrow *'a::len sword*
where
sword-of-integer $x \equiv \text{word-of-int} (\text{int-of-integer} x)$

definition *integer-of-word* :: *'a::len word* \Rightarrow *integer*
where
integer-of-word $x \equiv \text{integer-of-int} (\text{uint} x)$

definition *integer-of-sword* :: 'a::len sword \Rightarrow integer

where

integer-of-sword $x \equiv$ *integer-of-int* (*sint* x)

lemma *word-integer-eq*:

word-of-integer (*integer-of-word* w) = w

unfolding *word-of-integer-def* *integer-of-word-def*

by (*metis int-of-integer-of-int integer-of-int-eq-of-int word-uint.Rep-inverse'*)

lemma *sword-integer-eq*:

sword-of-integer (*integer-of-sword* w) = w

unfolding *sword-of-integer-def* *integer-of-sword-def*

by (*metis int-of-integer-of-int integer-of-int-eq-of-int word-sint.Rep-inverse'*)

lemma *integer-word-bounded-eq*:

assumes $0 \leq i$

assumes $i \leq 2 \wedge \text{LENGTH}('a::\text{len}) - 1$

shows *integer-of-word* ((*word-of-integer* :: integer \Rightarrow 'a word) i) = i

unfolding *integer-of-word-def* *word-of-integer-def*

using *assms*

by (*metis integer-less-eq-iff integer-of-int-eq-of-int minus-integer.rep-eq of-int-0-le-iff of-int-eq-1-iff of-int-eq-numeral-power-cancel-iff of-int-integer-of word-of-int-inverse zle-diff1-eq*)

lemma *integer-sword-bounded-eq*:

assumes $-(2 \wedge (\text{LENGTH}('a::\text{len}) - 1)) \leq i$

and $i < 2 \wedge (\text{LENGTH}('a) - 1)$

shows *integer-of-sword* ((*sword-of-integer* :: integer \Rightarrow 'a sword) i) = i

unfolding *integer-of-sword-def* *sword-of-integer-def*

using *signed-take-bit-int-eq-self assms*

by (*smt (verit) diff-numeral-special(11) int-of-integer-numeral integer-less-eq-iff integer-of-int-eq-of-int len-signed minus-integer.rep-eq numeral-power-eq-of-int-cancel-iff of-int-integer-of of-int-power-less-of-int-cancel-iff one-integer.rep-eq sint-of-int-eq uminus-integer.rep-eq*)

definition *word8-of-integer* :: integer \Rightarrow 8 word

where

word8-of-integer \equiv *word-of-integer*

definition *word16-of-integer* :: integer \Rightarrow 16 word

where

word16-of-integer \equiv *word-of-integer*

definition *word32-of-integer* :: integer \Rightarrow 32 word

where

word32-of-integer \equiv *word-of-integer*

definition *word64-of-integer* :: integer \Rightarrow 64 word

where

$word64\text{-of-integer} \equiv word\text{-of-integer}$

definition $integer\text{-of-word8} :: 8\ word \Rightarrow integer$
where
 $integer\text{-of-word8} \equiv integer\text{-of-word}$

definition $integer\text{-of-word16} :: 16\ word \Rightarrow integer$
where
 $integer\text{-of-word16} \equiv integer\text{-of-word}$

definition $integer\text{-of-word32} :: 32\ word \Rightarrow integer$
where
 $integer\text{-of-word32} \equiv integer\text{-of-word}$

definition $integer\text{-of-word64} :: 64\ word \Rightarrow integer$
where
 $integer\text{-of-word64} \equiv integer\text{-of-word}$

lemma $word8\text{-integer-eq}$:
 $word8\text{-of-integer} (integer\text{-of-word8}\ w) = w$
unfolding $word8\text{-of-integer-def}$ $integer\text{-of-word8-def}$
using $word\text{-integer-eq}$
by $blast$

lemma $word16\text{-integer-eq}$:
 $word16\text{-of-integer} (integer\text{-of-word16}\ w) = w$
unfolding $word16\text{-of-integer-def}$ $integer\text{-of-word16-def}$
using $word\text{-integer-eq}$
by $blast$

lemma $word32\text{-integer-eq}$:
 $word32\text{-of-integer} (integer\text{-of-word32}\ w) = w$
unfolding $word32\text{-of-integer-def}$ $integer\text{-of-word32-def}$
using $word\text{-integer-eq}$
by $blast$

lemma $word64\text{-integer-eq}$:
 $word64\text{-of-integer} (integer\text{-of-word64}\ w) = w$
unfolding $word64\text{-of-integer-def}$ $integer\text{-of-word64-def}$
using $word\text{-integer-eq}$
by $blast$

lemma $integer\text{-word8-bounded-eq}$:
assumes $0 \leq i$
and $i \leq 2 \wedge LENGTH(8) - 1$
shows $integer\text{-of-word8} (word8\text{-of-integer}\ i) = i$
unfolding $word8\text{-of-integer-def}$ $integer\text{-of-word8-def}$
using $integer\text{-word-bounded-eq}$ $assms$
by $blast$

lemma *integer-word16-bounded-eq*:
assumes $0 \leq i$
and $i \leq 2^{\wedge} \text{LENGTH}(16) - 1$
shows *integer-of-word16* (*word16-of-integer* i) = i
unfolding *word16-of-integer-def* *integer-of-word16-def*
using *integer-word-bounded-eq* *assms*
by *blast*

lemma *integer-word32-bounded-eq*:
assumes $0 \leq i$
and $i \leq 2^{\wedge} \text{LENGTH}(32) - 1$
shows *integer-of-word32* (*word32-of-integer* i) = i
unfolding *word32-of-integer-def* *integer-of-word32-def*
using *integer-word-bounded-eq* *assms*
by *blast*

lemma *integer-word64-bounded-eq*:
assumes $0 \leq i$
and $i \leq 2^{\wedge} \text{LENGTH}(64) - 1$
shows *integer-of-word64* (*word64-of-integer* i) = i
unfolding *word64-of-integer-def* *integer-of-word64-def*
using *integer-word-bounded-eq* *assms*
by *blast*

definition *word8-of-integer* :: *integer* \Rightarrow 8 *word*
where
word8-of-integer \equiv *word-of-integer*

definition *word16-of-integer* :: *integer* \Rightarrow 16 *word*
where
word16-of-integer \equiv *word-of-integer*

definition *word32-of-integer* :: *integer* \Rightarrow 32 *word*
where
word32-of-integer \equiv *word-of-integer*

definition *word64-of-integer* :: *integer* \Rightarrow 64 *word*
where
word64-of-integer \equiv *word-of-integer*

definition *integer-of-word8* :: 8 *word* \Rightarrow *integer*
where
integer-of-word8 \equiv *integer-of-word*

definition *integer-of-word16* :: 16 *word* \Rightarrow *integer*
where
integer-of-word16 \equiv *integer-of-word*

definition *integer-of-sword32* :: 32 sword \Rightarrow integer
 where
 integer-of-sword32 \equiv *integer-of-sword*

definition *integer-of-sword64* :: 64 sword \Rightarrow integer
 where
 integer-of-sword64 \equiv *integer-of-sword*

lemma *sword8-integer-eq*:
 sword8-of-integer (*integer-of-sword8* w) = w
 unfolding *sword8-of-integer-def integer-of-sword8-def*
 using *sword-integer-eq*
 by *blast*

lemma *sword16-integer-eq*:
 sword16-of-integer (*integer-of-sword16* w) = w
 unfolding *sword16-of-integer-def integer-of-sword16-def*
 using *sword-integer-eq*
 by *blast*

lemma *sword32-integer-eq*:
 sword32-of-integer (*integer-of-sword32* w) = w
 unfolding *sword32-of-integer-def integer-of-sword32-def*
 using *sword-integer-eq*
 by *blast*

lemma *sword64-integer-eq*:
 sword64-of-integer (*integer-of-sword64* w) = w
 unfolding *sword64-of-integer-def integer-of-sword64-def*
 using *sword-integer-eq*
 by *blast*

lemma *integer-sword8-bounded-eq*:
 assumes $-(2 \wedge (\text{LENGTH}(8) - 1)) \leq i$
 and $i < 2 \wedge (\text{LENGTH}(8) - 1)$
 shows *integer-of-sword8* (*sword8-of-integer* i) = i
 unfolding *sword8-of-integer-def integer-of-sword8-def*
 using *integer-sword-bounded-eq assms*
 by *metis*

lemma *integer-sword16-bounded-eq*:
 assumes $-(2 \wedge (\text{LENGTH}(16) - 1)) \leq i$
 and $i < 2 \wedge (\text{LENGTH}(16) - 1)$
 shows *integer-of-sword16* (*sword16-of-integer* i) = i
 unfolding *sword16-of-integer-def integer-of-sword16-def*
 using *integer-sword-bounded-eq assms*
 by *metis*

lemma *integer-sword32-bounded-eq*:

```

assumes  $-(2 \wedge (\text{LENGTH}(32) - 1)) \leq i$ 
and  $i < 2 \wedge (\text{LENGTH}(32) - 1)$ 
shows integer-of-sword32 (sword32-of-integer i) = i
unfolding sword32-of-integer-def integer-of-sword32-def
using integer-sword-bounded-eq assms
by metis

```

```

lemma integer-sword64-bounded-eq:
assumes  $-(2 \wedge (\text{LENGTH}(64) - 1)) \leq i$ 
and  $i < 2 \wedge (\text{LENGTH}(64) - 1)$ 
shows integer-of-sword64 (sword64-of-integer i) = i
unfolding sword64-of-integer-def integer-of-sword64-def
using integer-sword-bounded-eq assms
by metis

```

```

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  $\Rightarrow$  integer  $\Rightarrow$  integer
where
cast-val s i  $\equiv$ 
  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  $\Rightarrow$  nat (|-| $\tau$ )
where
  sizeof  $\tau \equiv$  case  $\tau$  of
    Uint8  $\Rightarrow$  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  $\exists n. 2^n = x$ 
proof (cases t)
  case Uint8
  then show ?thesis
    using assms
    unfolding sizeof-def
    by fastforce
  next
  case Sint8
  then show ?thesis
    using assms
    unfolding sizeof-def
    by fastforce
  next
  case Uint16
  then show ?thesis
    using assms
    unfolding sizeof-def
    by (rule-tac x=1 in exI) fastforce
  next
  case Sint16
  then show ?thesis
    using assms
    unfolding sizeof-def
    by (rule-tac x=1 in exI) fastforce
  next
  case Uint32
  then show ?thesis
    using assms
    unfolding sizeof-def
    by (rule-tac x=2 in exI) fastforce
  next
  case Sint32
  then show ?thesis
    using assms
    unfolding sizeof-def
    by (rule-tac x=2 in exI) fastforce

```

```

next
  case Uint64
  then show ?thesis
    using assms
    unfolding sizeof-def
    by (rule-tac  $x=3$  in exI) fastforce
next
  case Sint64
  then show ?thesis
    using assms
    unfolding sizeof-def
    by (rule-tac  $x=3$  in exI) fastforce
next
  case Cap
  then show ?thesis
    using assms
    unfolding sizeof-def
    by (rule-tac  $x=5$  in exI) fastforce
qed

```

```

lemma memval-size-u8:
  |memval-type (Uint8-v v)| $\tau$  = 1
  unfolding sizeof-def
  by fastforce

```

```

lemma memval-size-s8:
  |memval-type (Sint8-v v)| $\tau$  = 1
  unfolding sizeof-def
  by fastforce

```

```

lemma memval-size-u16:
  |memval-type (Uint16-v v)| $\tau$  = 2
  unfolding sizeof-def
  by fastforce

```

```

lemma memval-size-s16:
  |memval-type (Sint16-v v)| $\tau$  = 2
  unfolding sizeof-def
  by fastforce

```

```

lemma memval-size-u32:
  |memval-type (Uint32-v v)| $\tau$  = 4
  unfolding sizeof-def
  by fastforce

```

```

lemma memval-size-s32:
  |memval-type (Sint32-v v)| $\tau$  = 4
  unfolding sizeof-def
  by fastforce

```

lemma *memval-size-u64*:
|*memval-type* (*Uint64-v v*)|_τ = 8
unfolding *sizeof-def*
by *fastforce*

lemma *memval-size-s64*:
|*memval-type* (*Sint64-v v*)|_τ = 8
unfolding *sizeof-def*
by *fastforce*

lemma *memval-size-cap*:
|*memval-type* (*Cap-v v*)|_τ = 32
unfolding *sizeof-def*
by *fastforce*

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-v v*
shows |*memval-type val*|_τ = *length (u16-split v)*
using *assms memval-size-u16 flatten-u16-length*
by *force*

corollary *memval-size-s16-eq-word-split-len*:
assumes *val = Sint16-v v*
shows |*memval-type val*|_τ = *length (s16-split v)*
using *assms memval-size-s16 flatten-s16-length*
by *force*

corollary *memval-size-u32-eq-word-split-len*:
assumes *val = Uint32-v v*
shows |*memval-type val*|_τ = *length (flatten-u32 v)*
using *assms memval-size-u32 flatten-u32-length*
by *force*

corollary *memval-size-s32-eq-word-split-len*:
assumes *val = Sint32-v v*
shows |*memval-type val*|_τ = *length (flatten-s32 v)*
using *assms memval-size-s32 flatten-s32-length*
by *force*

corollary *memval-size-u64-eq-word-split-len*:
assumes *val = Uint64-v v*
shows |*memval-type val*|_τ = *length (flatten-u64 v)*
using *assms memval-size-u64 flatten-u64-length*
by *force*

corollary *memval-size-s64-eq-word-split-len*:
assumes $val = Sint64\text{-}v\ v$
shows $|memval\text{-}type\ val|_\tau = length\ (flatten\text{-}s64\ v)$
using *assms memval-size-s64 flatten-s64-length*
by *force*

lemma *sizeof-nonzero*:
 $|t|_\tau > 0$
by (*simp add: sizeof-def split: ctype.split*)

We prove that integer is a countable type.

instance *int :: comp-countable ..*

lemma *integer-encode-eq*: $(int\text{-}encode \circ int\text{-}of\text{-}integer)\ x = (int\text{-}encode \circ int\text{-}of\text{-}integer)\ y \iff x = y$
using *int-encode-eq integer-eq-iff*
by *auto*

instance *integer :: countable*
by (*rule countable-classI[of int-encode \circ int-of-integer]*) (*simp only: integer-encode-eq*)

instance *integer :: comp-countable ..*

3 Memory

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

3.1 Definitions

First, we provide \mathcal{V}_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* =
 $Byte\ (of\text{-}byte: 8\ word)$
 $| ACap\ (of\text{-}cap: memcap)\ (of\text{-}nth: nat)$

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
by (simp add: memval-is-byte-def split: memval.split-asm)
```

```
lemma memval-byte-not-memcap:
  memval-is-byte m ⇒ m ≠ ACap c n
by (simp add: memval-is-byte-def split: memval.split-asm)
```

```
lemma memval-memcap:
  memval-is-cap m ⇒ ∃ c n. m = ACap c n
by (simp add: memval-is-cap-def split: memval.split-asm)
```

```
lemma memval-memcap-not-byte:
  memval-is-cap m ⇒ m ≠ Byte b
by (simp add: memval-is-cap-def split: memval.split-asm)
```

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
  by (standard; (blast | simp add: sep-disj-unit-def))
end

```

```

instantiation nat :: cancellative-sep-algebra
begin
definition (n1::nat) ## n2 ≡ True
instance
  by (standard; (blast | simp add: sep-disj-nat-def))
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
  show  $\bigwedge x :: ('a, 'b) \text{ mapping}. x \## 0$ 
    by (simp add: sep-disj-map-def zero-map-def)
next
  show  $\bigwedge x :: ('a, 'b) \text{ mapping}. x + 0 = x$ 
    by (simp add: sep-disj-map-def Mapping.keys-def zero-map-def plus-map-def Mapping.empty.abs-eq Mapping.lookup.abs-eq mapping-eqI)
next
  show  $\bigwedge x y :: ('a, 'b) \text{ mapping}. x \## y \implies y \## x$ 
    using sep-disj-map-def
    by auto
next
  show  $\bigwedge x y z :: ('a, 'b) \text{ mapping}. x \## y \implies x + y = y + x$ 
    using map-add-comm
    by (fastforce simp add: sep-disj-map-def Mapping.keys-def zero-map-def plus-map-def Mapping.lookup-def)
next
  show  $\bigwedge x y z :: ('a, 'b) \text{ mapping}. [x \## y; y \## z; x \## z] \implies x + y + z = x + (y + z)$ 

```

```

    by (simp add: sep-disj-map-def Mapping.keys-def zero-map-def plus-map-def
Mapping.lookup-def Mapping-inverse)
next
  show  $\bigwedge x y z :: ('a, 'b) \text{ mapping}. \llbracket x \#\# y + z; y \#\# z \rrbracket \implies x \#\# y$ 
    by (simp add: sep-disj-map-def Mapping.keys-def zero-map-def plus-map-def
Mapping.lookup-def map-add-comm)
      (metis (no-types, opaque-lifting) Mapping.keys.abs-eq Mapping.keys.rep-eq
disjoint-iff domIff map-add-dom-app-simps(3))
next
  show  $\bigwedge x y z :: ('a, 'b) \text{ mapping}. \llbracket x \#\# y + z; y \#\# z \rrbracket \implies x + y \#\# z$ 
    by (simp add: sep-disj-map-def Mapping.keys-def zero-map-def plus-map-def
Mapping.lookup-def map-add-comm Mapping-inverse inf-commute inf-sup-distrib1)
next
  show  $\bigwedge x z y :: ('a, 'b) \text{ mapping}. \llbracket x + z = y + z; x \#\# z; y \#\# z \rrbracket \implies x = y$ 
    by (simp add: plus-map-def sep-disj-map-def)
      (metis (mono-tags, opaque-lifting) Mapping.keys.abs-eq Mapping.lookup.abs-eq
disjoint-iff domIff map-add-dom-app-simps(3) mapping-eq1)
qed
end

```

instantiation *heap-ext* :: (*cancellative-sep-algebra*) *cancellative-sep-algebra*

begin

definition $0 :: 'a \text{ heap-scheme} \equiv (\mid \text{next-block} = 0, \text{heap-map} = \text{Mapping.empty}, \dots = 0 \mid)$

definition $(m1 :: 'a \text{ heap-scheme}) + (m2 :: 'a \text{ heap-scheme}) \equiv$
 $(\mid \text{next-block} = \text{next-block } m1 + \text{next-block } m2,$
 $\text{heap-map} = \text{Mapping } ((\text{Mapping.lookup } (\text{heap-map } m1)) ++$
 $(\text{Mapping.lookup } (\text{heap-map } m2))),$
 $\dots = \text{heap.more } m1 + \text{heap.more } m2 \mid)$

definition $(m1 :: 'a \text{ heap-scheme}) \#\# (m2 :: 'a \text{ heap-scheme}) \equiv$
 $\text{Mapping.keys } (\text{heap-map } m1) \cap \text{Mapping.keys } (\text{heap-map } m2) = \{\}$
 $\wedge \text{heap.more } m1 \#\# \text{heap.more } m2$

instance

proof

show $\bigwedge x :: 'a \text{ heap-scheme}. x \#\# 0$

by (simp add: plus-heap-ext-def sep-disj-heap-ext-def zero-heap-ext-def)

next

show $\bigwedge x y :: 'a \text{ heap-scheme}. x \#\# y \implies y \#\# x$

by (simp add: plus-heap-ext-def sep-disj-heap-ext-def zero-heap-ext-def inf-commute sep-disj-commute)

next

show $\bigwedge x :: 'a \text{ heap-scheme}. x + 0 = x$

by (simp add: plus-heap-ext-def sep-disj-heap-ext-def zero-heap-ext-def inf-commute sep-disj-commute Mapping.empty-def Mapping.lookup.abs-eq)

(simp add: Mapping.lookup.rep-eq rep-inverse)

next

show $\bigwedge x y :: 'a \text{ heap-scheme}. x \#\# y \implies x + y = y + x$

by (simp add: plus-heap-ext-def sep-disj-heap-ext-def zero-heap-ext-def)

(metis keys-dom-lookup map-add-comm sep-add-commute)


```

next
  show  $\bigwedge x y z :: 'a \text{ heap-scheme. } \llbracket x \#\# y; y \#\# z; x \#\# z \rrbracket \implies x + y + z = x + (y + z)$ 
  by (simp add: plus-heap-ext-def sep-disj-heap-ext-def zero-heap-ext-def Mapping.lookup.abs-eq sep-add-assoc)
next
  show  $\bigwedge x y z :: 'a \text{ heap-scheme. } \llbracket x \#\# y + z; y \#\# z \rrbracket \implies x \#\# y$ 
  by (simp add: plus-heap-ext-def sep-disj-heap-ext-def zero-heap-ext-def)
  (metis plus-map-def sep-disj-addD sep-disj-map-def)
next
  show  $\bigwedge x y z :: 'a \text{ heap-scheme. } \llbracket x \#\# y + z; y \#\# z \rrbracket \implies x + y \#\# z$ 
  by (simp add: plus-heap-ext-def sep-disj-heap-ext-def zero-heap-ext-def Mapping.lookup.abs-eq disjoint-iff keys-dom-lookup sep-disj-addI1)
next
  show  $\bigwedge x z y :: 'a \text{ heap-scheme. } \llbracket x + z = y + z; x \#\# z; y \#\# z \rrbracket \implies x = y$ 
  by (simp add: plus-heap-ext-def sep-disj-heap-ext-def zero-heap-ext-def)
  (metis heap.equality plus-map-def sep-add-cancel sep-disj-map-def)
qed
end

```

instantiation *mem-capability-ext* :: (*comp-countable*, *zero*) *zero*

begin

definition *0* :: ('a, 'b) *mem-capability-scheme* \equiv

```

  ( $\mid$  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 ..

end

subclass (**in** *comp-countable*) *zero* .

instantiation *capability-ext* :: (*zero*) *zero*

begin

definition *0* \equiv (\mid *tag* = *False*, ... = 0)

instance ..

end

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

definition *null-capability* :: *cap* (*NULL*)

where

$NULL \equiv 0$

context

notes *null-capability-def*[simp]

begin

lemma *null-capability-block-id*[simp]:

block-id NULL = 0

by (*simp add: zero-mem-capability-ext-def*)

lemma *null-capability-offset*[simp]:

offset NULL = 0

by (*simp add: zero-mem-capability-ext-def*)

lemma *null-capability-base*[simp]:

base NULL = 0

by (*simp add: zero-mem-capability-ext-def*)

lemma *null-capability-len*[simp]:

len NULL = 0

by (*simp add: zero-mem-capability-ext-def*)

lemma *null-capability-perm-load*[simp]:

perm-load NULL = False

by (*simp add: zero-mem-capability-ext-def*)

lemma *null-capability-perm-cap-load*[simp]:

perm-cap-load NULL = False

by (*simp add: zero-mem-capability-ext-def*)

lemma *null-capability-perm-store*[simp]:

perm-store NULL = False

by (*simp add: zero-mem-capability-ext-def*)

lemma *null-capability-perm-cap-store*[simp]:

perm-cap-store NULL = False

by (*simp add: zero-mem-capability-ext-def*)

lemma *null-capability-perm-cap-store-local*[simp]:

perm-cap-store-local NULL = False

by (*simp add: zero-mem-capability-ext-def*)

lemma *null-capability-tag*[simp]:

tag NULL = False

by (*simp add: zero-capability-ext-def zero-mem-capability-ext-def*)

end

Here, we define the initial heap.

definition *init-heap* :: heap
where
init-heap $\equiv 0$ (\mid next-block := 1 \mid)

abbreviation *cap-offset* :: nat \Rightarrow nat
where
cap-offset $p \equiv$ if $p \bmod |Cap|_\tau = 0$ then p else $p - p \bmod |Cap|_\tau$

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

definition *wellformed* :: (block, t) mapping \Rightarrow bool ($\mathcal{W}_f^C/(-)$)
where
 $\mathcal{W}_f^C(h) \equiv$
 $\forall b \text{ obj. Mapping.lookup } h \ b = \text{Some } (Map \ \text{obj})$
 $\longrightarrow \text{Set.filter } (\lambda x. x \bmod |Cap|_\tau \neq 0) (\text{Mapping.keys } (tags \ \text{obj})) = \{\}$

lemma *init-heap-empty*:
Mapping.keys (heap-map *init-heap*) = $\{\}$
unfolding *init-heap-def zero-heap-ext-def*
by *simp*

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

lemma *init-wellformed*:
 $\mathcal{W}_f^C(\text{heap-map } \textit{init-heap})$
unfolding *init-heap-def wellformed-def zero-heap-ext-def*
by *simp*

lemma *mapping-lookup-disj1*:
 $m1 \ \#\# \ m2 \Longrightarrow \text{Mapping.lookup } m1 \ n = \text{Some } x \Longrightarrow \text{Mapping.lookup } (m1 + m2) \ n = \text{Some } x$
by (*metis Mapping.keys.rep-eq Mapping.lookup.abs-eq Mapping.lookup.rep-eq disjoint-iff is-none-simps(2) keys-is-none-rep map-add-dom-app-simps(3) plus-map-def sep-disj-map-def*)

lemma *mapping-lookup-disj2*:
 $m1 \ \#\# \ m2 \Longrightarrow \text{Mapping.lookup } m2 \ n = \text{Some } x \Longrightarrow \text{Mapping.lookup } (m1 + m2) \ n = \text{Some } x$
by (*metis Mapping.keys.rep-eq Mapping.lookup.abs-eq Mapping.lookup.rep-eq disjoint-iff is-none-simps(2) keys-is-none-rep map-add-dom-app-simps(2) plus-map-def sep-disj-map-def*)

Below shows that well-formedness is composition-compatible

lemma *heap-map h1 ## heap-map h2 $\Longrightarrow \mathcal{W}_f^C(\text{heap-map } h1 + \text{heap-map } h2)$*
 $\Longrightarrow \mathcal{W}_f^C(\text{heap-map } h1) \wedge \mathcal{W}_f^C(\text{heap-map } h2)$
unfolding *wellformed-def*
by (*safe, erule-tac x=b in allE, erule-tac x=obj in allE*)
(fastforce intro: mapping-lookup-disj1 mapping-lookup-disj2)+

4 Helper functions and lemmas

primrec *is-memval-defined* :: (nat, memval) mapping ⇒ nat ⇒ nat ⇒ bool

where

is-memval-defined - - 0 = True

| *is-memval-defined* m off (Suc siz) = ((off ∈ Mapping.keys m) ∧ *is-memval-defined* m (Suc off) siz)

primrec *is-contiguous-bytes* :: (nat, memval) mapping ⇒ nat ⇒ nat ⇒ bool

where

is-contiguous-bytes - - 0 = True

| *is-contiguous-bytes* m off (Suc siz) = ((off ∈ Mapping.keys m)
 ∧ memval-is-byte (the (Mapping.lookup m off))
 ∧ *is-contiguous-bytes* m (Suc off) siz)

definition *get-cap* :: (nat, memval) mapping ⇒ nat ⇒ memcap

where

get-cap m off = of-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 safe

```

show is-contiguous-zeros m off siz  $\implies$  is-contiguous-zeros-prim m off siz
  unfolding is-contiguous-zeros-def
proof (induct siz arbitrary: off)
  case 0
  thus ?case by simp
next
  case (Suc siz)
  thus ?case
    by fastforce
qed
next
show is-contiguous-zeros-prim m off siz  $\implies$  is-contiguous-zeros m off siz
  unfolding is-contiguous-zeros-def
proof (induct siz arbitrary: off)
  case 0
  thus ?case
    by auto
next
  case (Suc siz)
  have alt: is-contiguous-zeros-prim m (Suc off) siz
    using Suc(2) is-contiguous-zeros-prim.simps(2) [where ?m=m and ?off=off
and ?siz=siz]
    by blast
  have add-simp: Suc off + siz = off + Suc siz
    by simp
  show ?case
    using Suc(1) [where ?off=Suc off, OF alt, simplified add-simp le-eq-less-or-eq
Suc-le-eq]
    Suc(2) Suc-le-eq le-eq-less-or-eq
    by auto
qed
qed

```

```

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

```

```

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

```

```

definition retrieve-tval :: object  $\Rightarrow$  nat  $\Rightarrow$  cctype  $\Rightarrow$  bool  $\Rightarrow$  block cval

```

where

retrieve-tval obj off typ pcl \equiv

if is-contiguous-bytes (content obj) off |typ| $_{\tau}$ then

(case typ of

Uint8 \Rightarrow *Uint8-v (decode-u8-list (retrieve-bytes (content obj) off |typ| $_{\tau}$))*

| Sint8 \Rightarrow *Sint8-v (decode-s8-list (retrieve-bytes (content obj) off |typ| $_{\tau}$))*

| Uint16 \Rightarrow *Uint16-v (cat-u16 (retrieve-bytes (content obj) off |typ| $_{\tau}$))*

| Sint16 \Rightarrow *Sint16-v (cat-s16 (retrieve-bytes (content obj) off |typ| $_{\tau}$))*

| Uint32 \Rightarrow *Uint32-v (cat-u32 (retrieve-bytes (content obj) off |typ| $_{\tau}$))*

| Sint32 \Rightarrow *Sint32-v (cat-s32 (retrieve-bytes (content obj) off |typ| $_{\tau}$))*

| Uint64 \Rightarrow *Uint64-v (cat-u64 (retrieve-bytes (content obj) off |typ| $_{\tau}$))*

| Sint64 \Rightarrow *Sint64-v (cat-s64 (retrieve-bytes (content obj) off |typ| $_{\tau}$))*

| Cap \Rightarrow *if is-contiguous-zeros (content obj) off |typ| $_{\tau}$ 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 \Rightarrow False | True \Rightarrow tv) in

let cv = mem-capability.extend cap (λ tag = t λ) in

let nth-frag = of-nth (the (Mapping.lookup (content obj) off)) in

(case typ of

Uint8 \Rightarrow *Cap-v-frag (mem-capability.extend cap (λ tag = False λ)) nth-frag*

| Sint8 \Rightarrow *Cap-v-frag (mem-capability.extend cap (λ tag = False λ)) nth-frag*

| Cap \Rightarrow *if is-contiguous-cap (content obj) cap off |typ| $_{\tau}$ then Cap-v cv else*

Undef

| - \Rightarrow Undef)

else Undef

primrec *store-bytes* :: (nat, memval) mapping \Rightarrow nat \Rightarrow 8 word list \Rightarrow (nat, memval) 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 \Rightarrow nat \Rightarrow cap \Rightarrow nat \Rightarrow (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 \Rightarrow nat \Rightarrow bool \Rightarrow (nat, bool) mapping

where

store-tag obj off tg \equiv Mapping.update off tg obj

definition *store-tval* :: object \Rightarrow nat \Rightarrow block cval \Rightarrow object

where

$store\text{-}tval\ obj\ off\ val \equiv$
 $case\ val\ of\ Uint8\text{-}v\ v \Rightarrow obj\ (\ content := store\text{-}bytes\ (content\ obj)\ off$
 $(encode\text{-}u8\text{-}list\ v),$
 $tags := store\text{-}tag\ (tags\ obj)\ (cap\text{-}offset\ off)\ False\ \rangle$
 $| Sint8\text{-}v\ v \Rightarrow obj\ (\ content := store\text{-}bytes\ (content\ obj)\ off$
 $(encode\text{-}s8\text{-}list\ v),$
 $tags := store\text{-}tag\ (tags\ obj)\ (cap\text{-}offset\ off)\ False\ \rangle$
 $| Uint16\text{-}v\ v \Rightarrow obj\ (\ content := store\text{-}bytes\ (content\ obj)\ off$
 $(u16\text{-}split\ v),$
 $tags := store\text{-}tag\ (tags\ obj)\ (cap\text{-}offset\ off)\ False\ \rangle$
 $| Sint16\text{-}v\ v \Rightarrow obj\ (\ content := store\text{-}bytes\ (content\ obj)\ off\ (s16\text{-}split$
 $v),$
 $tags := store\text{-}tag\ (tags\ obj)\ (cap\text{-}offset\ off)\ False\ \rangle$
 $| Uint32\text{-}v\ v \Rightarrow obj\ (\ content := store\text{-}bytes\ (content\ obj)\ off$
 $(flatten\text{-}u32\ v),$
 $tags := store\text{-}tag\ (tags\ obj)\ (cap\text{-}offset\ off)\ False\ \rangle$
 $| Sint32\text{-}v\ v \Rightarrow obj\ (\ content := store\text{-}bytes\ (content\ obj)\ off$
 $(flatten\text{-}s32\ v),$
 $tags := store\text{-}tag\ (tags\ obj)\ (cap\text{-}offset\ off)\ False\ \rangle$
 $| Uint64\text{-}v\ v \Rightarrow obj\ (\ content := store\text{-}bytes\ (content\ obj)\ off$
 $(flatten\text{-}u64\ v),$
 $tags := store\text{-}tag\ (tags\ obj)\ (cap\text{-}offset\ off)\ False\ \rangle$
 $| Sint64\text{-}v\ v \Rightarrow obj\ (\ content := store\text{-}bytes\ (content\ obj)\ off$
 $(flatten\text{-}s64\ v),$
 $tags := store\text{-}tag\ (tags\ obj)\ (cap\text{-}offset\ off)\ False\ \rangle$
 $| Cap\text{-}v\ c \Rightarrow obj\ (\ content := store\text{-}cap\ (content\ obj)\ off\ c\ |Cap|_{\tau},$
 $tags := store\text{-}tag\ (tags\ obj)\ (cap\text{-}offset\ off)\ (tag\ c)\ \rangle$
 $| Cap\text{-}v\text{-}frag\ c\ n \Rightarrow obj\ (\ content := Mapping.update\ off\ (ACap$
 $(mem\text{-}capability.truncate\ c)\ n)\ (content\ obj),$
 $tags := store\text{-}tag\ (tags\ obj)\ (cap\text{-}offset\ off)\ False\ \rangle$

lemma *stored-bytes-prev:*

assumes $x < off$

shows $Mapping.lookup\ (store\text{-}bytes\ obj\ off\ vs)\ x = Mapping.lookup\ obj\ x$

using *assms*

by $(induct\ vs\ arbitrary:\ obj\ off)\ fastforce+$

lemma *stored-tags-prev:*

assumes $x < off$

shows $Mapping.lookup\ (store\text{-}tag\ obj\ off\ vs)\ x = Mapping.lookup\ obj\ x$

using *assms*

by *force*

lemma *stored-cap-prev:*

assumes $x < off$

shows $Mapping.lookup\ (store\text{-}cap\ obj\ off\ cap\ siz)\ x = Mapping.lookup\ obj\ x$

using *assms*

by $(induct\ siz\ arbitrary:\ obj\ off)\ fastforce+$

lemma *stored-bytes-instant-correctness:*

Mapping.lookup (store-bytes obj off (v # vs)) off = Some (Byte v)

proof (*induct vs arbitrary: obj off*)

case *Nil*

thus *?case by force*

next

case (*Cons a vs*)

thus *?case using stored-bytes-prev Suc-eq-plus1 lessI store-bytes.simps(2)*

by *metis*

qed

lemma *stored-cap-instant-correctness:*

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

proof (*induct siz arbitrary: obj off*)

case *0*

thus *?case by force*

next

case (*Suc siz*)

thus *?case using stored-cap-prev Suc-eq-plus1 lessI store-cap.simps(2) lookup-update*

by *metis*

qed

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

by (*simp add: eval-nat-numeral*)

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

by (*simp add: eval-nat-numeral*)

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

by (*simp add: eval-nat-numeral*)

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

by (*simp add: eval-nat-numeral*)

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

by (*simp add: eval-nat-numeral*)

lemma *list-length-2-realise:*

length ls = 2 \implies \exists n0 n1. ls = [n0, n1]

by (*metis One-nat-def Suc-length-conv add-diff-cancel-right' len-gt-0 len-of-finite-2-def*)

list.size(4) list-exhaust-size-eq0 list-exhaust-size-gt0 one-add-one)

lemma *list-length-4-realise:*

length ls = 4 \implies \exists n0 n1 n2 n3. ls = [n0, n1, n2, n3]

by (*metis list-exhaust-size-eq0 list-exhaust-size-gt0 numeral-4-eq-4 size-Cons-lem-eq zero-less-Suc*)

lemma *list-length-8-realise*:

$length\ ls = 8 \implies \exists\ n0\ n1\ n2\ n3\ n4\ n5\ n6\ n7. ls = [n0, n1, n2, n3, n4, n5, n6, n7]$

using *list-exhaust-size-eq0 list-exhaust-size-gt0 numeral-8-eq-8 size-Cons-lem-eq zero-less-Suc*

by *smt*

lemma *u16-split-realise*:

$\exists\ b0\ b1. u16-split\ v = [b0, b1]$

using *list-length-2-realise[where ?ls=u16-split v, OF flatten-u16-length[where ?vs=v]]*

by *assumption*

lemma *s16-split-realise*:

$\exists\ b0\ b1. s16-split\ v = [b0, b1]$

using *list-length-2-realise[where ?ls=s16-split v, OF flatten-s16-length[where ?vs=v]]*

by *assumption*

lemma *u32-split-realise*:

$\exists\ b0\ b1\ b2\ b3. flatten-u32\ v = [b0, b1, b2, b3]$

using *list-length-4-realise[where ?ls=flatten-u32 v, OF flatten-u32-length[where ?vs=v]]*

by *assumption*

lemma *s32-split-realise*:

$\exists\ b0\ b1\ b2\ b3. flatten-s32\ v = [b0, b1, b2, b3]$

using *list-length-4-realise[where ?ls=flatten-s32 v, OF flatten-s32-length[where ?vs=v]]*

by *assumption*

lemma *u64-split-realise*:

$\exists\ b0\ b1\ b2\ b3\ b4\ b5\ b6\ b7. flatten-u64\ v = [b0, b1, b2, b3, b4, b5, b6, b7]$

using *list-length-8-realise[where ?ls=flatten-u64 v, OF flatten-u64-length[where ?vs=v]]*

by *assumption*

lemma *s64-split-realise*:

$\exists\ b0\ b1\ b2\ b3\ b4\ b5\ b6\ b7. flatten-s64\ v = [b0, b1, b2, b3, b4, b5, b6, b7]$

using *list-length-8-realise[where ?ls=flatten-s64 v, OF flatten-s64-length[where ?vs=v]]*

by *assumption*

lemma *store-bytes-u16*:

shows $off \in Mapping.keys\ (store-bytes\ m\ off\ (u16-split\ v))$

and $Suc\ off \in Mapping.keys\ (store-bytes\ m\ off\ (u16-split\ v))$

and $\exists\ b0. Mapping.lookup\ (store-bytes\ m\ off\ (u16-split\ v))\ off = Some\ (Byte\ b0)$

and $\exists\ b1. Mapping.lookup\ (store-bytes\ m\ off\ (u16-split\ v))\ (Suc\ off) = Some$

(*Byte b1*)
proof –
 show $off \in \text{Mapping.keys } (store\text{-}bytes\ m\ off\ (u16\text{-}split\ v))$
 by (*metis* (*no-types*, *opaque-lifting*) *domIff* *u16-split-realise* *handy-if-lemma*
keys-dom-lookup
stored-bytes-instant-correctness)
next
 show $Suc\ off \in \text{Mapping.keys } (store\text{-}bytes\ m\ off\ (u16\text{-}split\ v))$
 by (*metis* (*mono-tags*, *opaque-lifting*) *domIff* *u16-split-realise* *handy-if-lemma*
keys-dom-lookup
store-bytes.simps(2) *stored-bytes-instant-correctness*)
next
 show $\exists\ b0. \text{Mapping.lookup } (store\text{-}bytes\ m\ off\ (u16\text{-}split\ v))\ off = \text{Some } (Byte\ b0)$
 by (*metis* *u16-split-realise* *stored-bytes-instant-correctness*)
next
 show $\exists\ b1. \text{Mapping.lookup } (store\text{-}bytes\ m\ off\ (u16\text{-}split\ v))\ (Suc\ off) = \text{Some } (Byte\ b1)$
 by (*metis* *u16-split-realise* *store-bytes.simps(2)* *stored-bytes-instant-correctness*)
qed

lemma *store-bytes-s16*:

shows $off \in \text{Mapping.keys } (store\text{-}bytes\ m\ off\ (s16\text{-}split\ v))$
 and $Suc\ off \in \text{Mapping.keys } (store\text{-}bytes\ m\ off\ (s16\text{-}split\ v))$
 and $\exists\ b0. \text{Mapping.lookup } (store\text{-}bytes\ m\ off\ (s16\text{-}split\ v))\ off = \text{Some } (Byte\ b0)$
 and $\exists\ b1. \text{Mapping.lookup } (store\text{-}bytes\ m\ off\ (s16\text{-}split\ v))\ (Suc\ off) = \text{Some } (Byte\ b1)$
proof –
 show $off \in \text{Mapping.keys } (store\text{-}bytes\ m\ off\ (s16\text{-}split\ v))$
 by (*metis* (*no-types*, *opaque-lifting*) *domIff* *s16-split-realise* *handy-if-lemma*
keys-dom-lookup
stored-bytes-instant-correctness)
next
 show $Suc\ off \in \text{Mapping.keys } (store\text{-}bytes\ m\ off\ (s16\text{-}split\ v))$
 by (*metis* (*mono-tags*, *opaque-lifting*) *domIff* *s16-split-realise* *handy-if-lemma*
keys-dom-lookup
store-bytes.simps(2) *stored-bytes-instant-correctness*)
next
 show $\exists\ b0. \text{Mapping.lookup } (store\text{-}bytes\ m\ off\ (s16\text{-}split\ v))\ off = \text{Some } (Byte\ b0)$
 by (*metis* *s16-split-realise* *stored-bytes-instant-correctness*)
next
 show $\exists\ b1. \text{Mapping.lookup } (store\text{-}bytes\ m\ off\ (s16\text{-}split\ v))\ (Suc\ off) = \text{Some } (Byte\ b1)$
 by (*metis* *s16-split-realise* *store-bytes.simps(2)* *stored-bytes-instant-correctness*)
qed

lemma *store-bytes-u32*:

```

shows  $off \in Mapping.keys (store-bytes m off (flatten-u32 v))$ 
  and  $Suc\ off \in Mapping.keys (store-bytes m off (flatten-u32 v))$ 
  and  $Suc (Suc\ off) \in Mapping.keys (store-bytes m off (flatten-u32 v))$ 
  and  $Suc (Suc (Suc\ off)) \in Mapping.keys (store-bytes m off (flatten-u32 v))$ 
  and  $\exists b0. Mapping.lookup (store-bytes m off (flatten-u32 v))\ off = Some (Byte\ b0)$ 
  and  $\exists b1. Mapping.lookup (store-bytes m off (flatten-u32 v)) (Suc\ off) = Some (Byte\ b1)$ 
  and  $\exists b2. Mapping.lookup (store-bytes m off (flatten-u32 v)) (Suc (Suc\ off)) = Some (Byte\ b2)$ 
  and  $\exists b3. Mapping.lookup (store-bytes m off (flatten-u32 v)) (Suc (Suc (Suc\ off))) = Some (Byte\ b3)$ 
proof –
  show  $off \in Mapping.keys (store-bytes m off (flatten-u32 v))$ 
    by (metis (no-types, opaque-lifting) domIff handy-if-lemma keys-dom-lookup stored-bytes-instant-correctness u32-split-realise)
next
  show  $Suc\ off \in Mapping.keys (store-bytes m off (flatten-u32 v))$ 
    by (metis (mono-tags, opaque-lifting) domIff u32-split-realise handy-if-lemma keys-dom-lookup store-bytes.simps(2) stored-bytes-instant-correctness)
next
  show  $Suc (Suc\ off) \in Mapping.keys (store-bytes m off (flatten-u32 v))$ 
    by (metis (mono-tags, opaque-lifting) domIff u32-split-realise handy-if-lemma keys-dom-lookup store-bytes.simps(2) stored-bytes-instant-correctness)
next
  show  $Suc (Suc (Suc\ off)) \in Mapping.keys (store-bytes m off (flatten-u32 v))$ 
    by (metis (mono-tags, opaque-lifting) domIff u32-split-realise handy-if-lemma keys-dom-lookup store-bytes.simps(2) stored-bytes-instant-correctness)
next
  show  $\exists b0. Mapping.lookup (store-bytes m off (flatten-u32 v))\ off = Some (Byte\ b0)$ 
    by (metis u32-split-realise stored-bytes-instant-correctness)
next
  show  $\exists b1. Mapping.lookup (store-bytes m off (flatten-u32 v)) (Suc\ off) = Some (Byte\ b1)$ 
    by (metis u32-split-realise store-bytes.simps(2) stored-bytes-instant-correctness)
next
  show  $\exists b2. Mapping.lookup (store-bytes m off (flatten-u32 v)) (Suc (Suc\ off)) = Some (Byte\ b2)$ 
    by (metis u32-split-realise store-bytes.simps(2) stored-bytes-instant-correctness)
next
  show  $\exists b3. Mapping.lookup (store-bytes m off (flatten-u32 v)) (Suc (Suc (Suc\ off))) = Some (Byte\ b3)$ 
    by (metis u32-split-realise store-bytes.simps(2) stored-bytes-instant-correctness)
qed

```

```

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 -
    show off ∈ Mapping.keys (store-bytes m off (flatten-s32 v))
      by (metis (no-types, opaque-lifting) domIff handy-if-lemma keys-dom-lookup
stored-bytes-instant-correctness s32-split-realise)
    next
      show Suc off ∈ Mapping.keys (store-bytes m off (flatten-s32 v))
        by (metis (mono-tags, opaque-lifting) domIff s32-split-realise handy-if-lemma
keys-dom-lookup
store-bytes.simps(2) stored-bytes-instant-correctness)
    next
      show Suc (Suc off) ∈ Mapping.keys (store-bytes m off (flatten-s32 v))
        by (metis (mono-tags, opaque-lifting) domIff s32-split-realise handy-if-lemma
keys-dom-lookup
store-bytes.simps(2) stored-bytes-instant-correctness)
    next
      show Suc (Suc (Suc off)) ∈ Mapping.keys (store-bytes m off (flatten-s32 v))
        by (metis (mono-tags, opaque-lifting) domIff s32-split-realise handy-if-lemma
keys-dom-lookup
store-bytes.simps(2) stored-bytes-instant-correctness)
    next
      show ∃ b0. Mapping.lookup (store-bytes m off (flatten-s32 v)) off = Some (Byte
b0)
        by (metis s32-split-realise stored-bytes-instant-correctness)
    next
      show ∃ b1. Mapping.lookup (store-bytes m off (flatten-s32 v)) (Suc off) = Some
(Byte b1)
        by (metis s32-split-realise store-bytes.simps(2) stored-bytes-instant-correctness)
    next
      show ∃ b2. Mapping.lookup (store-bytes m off (flatten-s32 v)) (Suc (Suc off))
= Some (Byte b2)
        by (metis s32-split-realise store-bytes.simps(2) stored-bytes-instant-correctness)
    next
      show ∃ b3. Mapping.lookup (store-bytes m off (flatten-s32 v)) (Suc (Suc (Suc
off))) = Some (Byte b3)
        by (metis s32-split-realise store-bytes.simps(2) stored-bytes-instant-correctness)
  qed

```

lemma *store-bytes-u64*:

shows $off \in \text{Mapping.keys } (store\text{-bytes } m \text{ off } (flatten\text{-u64 } v))$
and $Suc \text{ off} \in \text{Mapping.keys } (store\text{-bytes } m \text{ off } (flatten\text{-u64 } v))$
and $Suc (Suc \text{ off}) \in \text{Mapping.keys } (store\text{-bytes } m \text{ off } (flatten\text{-u64 } v))$
and $Suc (Suc (Suc \text{ off})) \in \text{Mapping.keys } (store\text{-bytes } m \text{ off } (flatten\text{-u64 } v))$
and $Suc (Suc (Suc (Suc \text{ off}))) \in \text{Mapping.keys } (store\text{-bytes } m \text{ off } (flatten\text{-u64 } v))$
and $Suc (Suc (Suc (Suc (Suc \text{ off})))) \in \text{Mapping.keys } (store\text{-bytes } m \text{ off } (flatten\text{-u64 } v))$
and $Suc (Suc (Suc (Suc (Suc (Suc \text{ off})))))) \in \text{Mapping.keys } (store\text{-bytes } m \text{ off } (flatten\text{-u64 } v))$
and $Suc (Suc (Suc (Suc (Suc (Suc (Suc \text{ off})))))) \in \text{Mapping.keys } (store\text{-bytes } m \text{ off } (flatten\text{-u64 } v))$
and $\exists b0. \text{Mapping.lookup } (store\text{-bytes } m \text{ off } (flatten\text{-u64 } v)) \text{ off} = \text{Some } (Byte \text{ b0})$
and $\exists b1. \text{Mapping.lookup } (store\text{-bytes } m \text{ off } (flatten\text{-u64 } v)) (Suc \text{ off}) = \text{Some } (Byte \text{ b1})$
and $\exists b2. \text{Mapping.lookup } (store\text{-bytes } m \text{ off } (flatten\text{-u64 } v)) (Suc (Suc \text{ off})) = \text{Some } (Byte \text{ b2})$
and $\exists b3. \text{Mapping.lookup } (store\text{-bytes } m \text{ off } (flatten\text{-u64 } v)) (Suc (Suc (Suc \text{ off}))) = \text{Some } (Byte \text{ b3})$
and $\exists b0. \text{Mapping.lookup } (store\text{-bytes } m \text{ off } (flatten\text{-u64 } v)) (Suc (Suc (Suc (Suc \text{ off})))) = \text{Some } (Byte \text{ b0})$
and $\exists b1. \text{Mapping.lookup } (store\text{-bytes } m \text{ off } (flatten\text{-u64 } v)) (Suc (Suc (Suc (Suc (Suc \text{ off})))))) = \text{Some } (Byte \text{ b1})$
and $\exists b2. \text{Mapping.lookup } (store\text{-bytes } m \text{ off } (flatten\text{-u64 } v)) (Suc (Suc (Suc (Suc (Suc (Suc \text{ off})))))) = \text{Some } (Byte \text{ b2})$
and $\exists b3. \text{Mapping.lookup } (store\text{-bytes } m \text{ off } (flatten\text{-u64 } v)) (Suc (Suc (Suc (Suc (Suc (Suc (Suc \text{ off})))))) = \text{Some } (Byte \text{ b3})$

proof –

show $off \in \text{Mapping.keys } (store\text{-bytes } m \text{ off } (flatten\text{-u64 } v))$
by (*metis* (*no-types*, *opaque-lifting*) *domIff handy-if-lemma keys-dom-lookup stored-bytes-instant-correctness u64-split-realise*)

next

show $Suc \text{ off} \in \text{Mapping.keys } (store\text{-bytes } m \text{ off } (flatten\text{-u64 } v))$
by (*metis* (*mono-tags*, *opaque-lifting*) *domIff u64-split-realise handy-if-lemma keys-dom-lookup store-bytes.simps(2) stored-bytes-instant-correctness*)

next

show $Suc (Suc \text{ off}) \in \text{Mapping.keys } (store\text{-bytes } m \text{ off } (flatten\text{-u64 } v))$
by (*metis* (*mono-tags*, *opaque-lifting*) *domIff u64-split-realise handy-if-lemma keys-dom-lookup store-bytes.simps(2) stored-bytes-instant-correctness*)

next

show $Suc (Suc (Suc \text{ off})) \in \text{Mapping.keys } (store\text{-bytes } m \text{ off } (flatten\text{-u64 } v))$
by (*metis* (*mono-tags*, *opaque-lifting*) *domIff u64-split-realise handy-if-lemma keys-dom-lookup store-bytes.simps(2) stored-bytes-instant-correctness*)

```

next
  show  $Suc (Suc (Suc (Suc\ off))) \in Mapping.keys (store-bytes\ m\ off (flatten-u64\ v))$ 
  by (metis (mono-tags, opaque-lifting) domIff u64-split-realise handy-if-lemma keys-dom-lookup store-bytes.simps(2) stored-bytes-instant-correctness)
next
  show  $Suc (Suc (Suc (Suc (Suc\ off)))) \in Mapping.keys (store-bytes\ m\ off (flatten-u64\ v))$ 
  by (metis (mono-tags, opaque-lifting) domIff u64-split-realise handy-if-lemma keys-dom-lookup store-bytes.simps(2) stored-bytes-instant-correctness)
next
  show  $Suc (Suc (Suc (Suc (Suc (Suc\ off)))))) \in Mapping.keys (store-bytes\ m\ off (flatten-u64\ v))$ 
  by (metis (mono-tags, opaque-lifting) domIff u64-split-realise handy-if-lemma keys-dom-lookup store-bytes.simps(2) stored-bytes-instant-correctness)
next
  show  $Suc (Suc (Suc (Suc (Suc (Suc (Suc\ off)))))) \in Mapping.keys (store-bytes\ m\ off (flatten-u64\ v))$ 
  by (metis (mono-tags, opaque-lifting) domIff u64-split-realise handy-if-lemma keys-dom-lookup store-bytes.simps(2) stored-bytes-instant-correctness)
next
  show  $\exists b0. Mapping.lookup (store-bytes\ m\ off (flatten-u64\ v))\ off = Some (Byte\ b0)$ 
  by (metis u64-split-realise stored-bytes-instant-correctness)
next
  show  $\exists b1. Mapping.lookup (store-bytes\ m\ off (flatten-u64\ v)) (Suc\ off) = Some (Byte\ b1)$ 
  by (metis u64-split-realise store-bytes.simps(2) stored-bytes-instant-correctness)
next
  show  $\exists b2. Mapping.lookup (store-bytes\ m\ off (flatten-u64\ v)) (Suc (Suc\ off)) = Some (Byte\ b2)$ 
  by (metis u64-split-realise store-bytes.simps(2) stored-bytes-instant-correctness)
next
  show  $\exists b3. Mapping.lookup (store-bytes\ m\ off (flatten-u64\ v)) (Suc (Suc (Suc\ off))) = Some (Byte\ b3)$ 
  by (metis u64-split-realise store-bytes.simps(2) stored-bytes-instant-correctness)
next
  show  $\exists b0. Mapping.lookup (store-bytes\ m\ off (flatten-u64\ v)) (Suc (Suc (Suc (Suc\ off)))) = Some (Byte\ b0)$ 
  by (metis u64-split-realise store-bytes.simps(2) stored-bytes-instant-correctness)
next
  show  $\exists b1. Mapping.lookup (store-bytes\ m\ off (flatten-u64\ v)) (Suc (Suc (Suc (Suc (Suc\ off)))))) = Some (Byte\ b1)$ 
  by (metis u64-split-realise store-bytes.simps(2) stored-bytes-instant-correctness)
next

```


by (*metis* (*mono-tags*, *opaque-lifting*) *domIff* *s64-split-realise* *handy-if-lemma* *keys-dom-lookup* *store-bytes.simps(2)* *stored-bytes-instant-correctness*)
next
show *Suc* (*Suc* (*Suc* *off*)) \in *Mapping.keys* (*store-bytes* *m* *off* (*flatten-s64* *v*))
by (*metis* (*mono-tags*, *opaque-lifting*) *domIff* *s64-split-realise* *handy-if-lemma* *keys-dom-lookup* *store-bytes.simps(2)* *stored-bytes-instant-correctness*)
next
show *Suc* (*Suc* (*Suc* (*Suc* *off*))) \in *Mapping.keys* (*store-bytes* *m* *off* (*flatten-s64* *v*))
by (*metis* (*mono-tags*, *opaque-lifting*) *domIff* *s64-split-realise* *handy-if-lemma* *keys-dom-lookup* *store-bytes.simps(2)* *stored-bytes-instant-correctness*)
next
show *Suc* (*Suc* (*Suc* (*Suc* (*Suc* *off*)))) \in *Mapping.keys* (*store-bytes* *m* *off* (*flatten-s64* *v*))
by (*metis* (*mono-tags*, *opaque-lifting*) *domIff* *s64-split-realise* *handy-if-lemma* *keys-dom-lookup* *store-bytes.simps(2)* *stored-bytes-instant-correctness*)
next
show *Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* *off*)))))) \in *Mapping.keys* (*store-bytes* *m* *off* (*flatten-s64* *v*))
by (*metis* (*mono-tags*, *opaque-lifting*) *domIff* *s64-split-realise* *handy-if-lemma* *keys-dom-lookup* *store-bytes.simps(2)* *stored-bytes-instant-correctness*)
next
show \exists *b0*. *Mapping.lookup* (*store-bytes* *m* *off* (*flatten-s64* *v*)) *off* = *Some* (*Byte* *b0*)
by (*metis* *s64-split-realise* *stored-bytes-instant-correctness*)
next
show \exists *b1*. *Mapping.lookup* (*store-bytes* *m* *off* (*flatten-s64* *v*)) (*Suc* *off*) = *Some* (*Byte* *b1*)
by (*metis* *s64-split-realise* *store-bytes.simps(2)* *stored-bytes-instant-correctness*)
next
show \exists *b2*. *Mapping.lookup* (*store-bytes* *m* *off* (*flatten-s64* *v*)) (*Suc* (*Suc* *off*)) = *Some* (*Byte* *b2*)
by (*metis* *s64-split-realise* *store-bytes.simps(2)* *stored-bytes-instant-correctness*)
next
show \exists *b3*. *Mapping.lookup* (*store-bytes* *m* *off* (*flatten-s64* *v*)) (*Suc* (*Suc* (*Suc* *off*))) = *Some* (*Byte* *b3*)
by (*metis* *s64-split-realise* *store-bytes.simps(2)* *stored-bytes-instant-correctness*)
next

show $\exists b0. \text{Mapping.lookup (store-bytes m off (flatten-s64 v)) (Suc (Suc (Suc (Suc off))))} = \text{Some (Byte b0)}$
by (metis s64-split-realise store-bytes.simps(2) stored-bytes-instant-correctness)
next
show $\exists b1. \text{Mapping.lookup (store-bytes m off (flatten-s64 v)) (Suc (Suc (Suc (Suc (Suc off))))} = \text{Some (Byte b1)}$
by (metis s64-split-realise store-bytes.simps(2) stored-bytes-instant-correctness)
next
show $\exists b2. \text{Mapping.lookup (store-bytes m off (flatten-s64 v)) (Suc (Suc (Suc (Suc (Suc (Suc off))))} = \text{Some (Byte b2)}$
by (metis s64-split-realise store-bytes.simps(2) stored-bytes-instant-correctness)
next
show $\exists b3. \text{Mapping.lookup (store-bytes m off (flatten-s64 v)) (Suc (Suc (Suc (Suc (Suc (Suc (Suc off))))} = \text{Some (Byte b3)}$
by (metis s64-split-realise store-bytes.simps(2) stored-bytes-instant-correctness)
qed

corollary *u16-store-bytes-imp-is-contiguous-bytes:*
is-contiguous-bytes (store-bytes m off (u16-split v)) off 2
by (metis One-nat-def Suc-1 is-contiguous-bytes.simps(1) is-contiguous-bytes.simps(2) memval-memcap-not-byte option.sel store-bytes-u16)

corollary *s16-store-bytes-imp-is-contiguous-bytes:*
is-contiguous-bytes (store-bytes m off (s16-split v)) off 2
by (metis One-nat-def Suc-1 is-contiguous-bytes.simps(1) is-contiguous-bytes.simps(2) memval-memcap-not-byte option.sel store-bytes-s16)

corollary *u32-store-bytes-imp-is-contiguous-bytes:*
is-contiguous-bytes (store-bytes m off (flatten-u32 v)) off 4
by (simp add: numeral-4-eq-4, safe)
(simp add: store-bytes-u32, metis memval-memcap-not-byte option.sel store-bytes-u32)+

corollary *s32-store-bytes-imp-is-contiguous-bytes:*
is-contiguous-bytes (store-bytes m off (flatten-s32 v)) off 4
by (simp add: numeral-4-eq-4, safe)
(simp add: store-bytes-s32, metis memval-memcap-not-byte option.sel store-bytes-s32)+

corollary *u64-store-bytes-imp-is-contiguous-bytes:*
is-contiguous-bytes (store-bytes m off (flatten-u64 v)) off 8
by (simp add: numeral-8-eq-8, safe)
(simp add: store-bytes-u64, metis memval-memcap-not-byte option.sel store-bytes-u64)+

corollary *s64-store-bytes-imp-is-contiguous-bytes:*
is-contiguous-bytes (store-bytes m off (flatten-s64 v)) off 8
by (simp add: numeral-8-eq-8, safe)
(simp add: store-bytes-s64, metis memval-memcap-not-byte option.sel store-bytes-s64)+

lemma *stored-tval-contiguous-bytes*:

assumes $val \neq \text{Undef}$

and $\forall v. val \neq \text{Cap-}v$

and $\forall v n. val \neq \text{Cap-}v\text{-frag } v n$

shows *is-contiguous-bytes* (*content* (*store-tval obj off val*)) *off* |*memval-type val*| $_{\tau}$

unfolding *sizeof-def*

by (*simp add: assms store-tval-def memval-is-byte-def split: ccval.split*) (*presburger*

add: s16-store-bytes-imp-is-contiguous-bytes s32-store-bytes-imp-is-contiguous-bytes

s64-store-bytes-imp-is-contiguous-bytes u16-store-bytes-imp-is-contiguous-bytes u32-store-bytes-imp-is-contiguous-

u64-store-bytes-imp-is-contiguous-bytes)

lemma *suc-of-32*:

$32 = \text{Suc } 31$

by *simp*

lemma *store-cap-correct-dom*:

shows $off \in \text{Mapping.keys (store-cap } m \text{ off cap } 32)$

and $off + 1 \in \text{Mapping.keys (store-cap } m \text{ off cap } 32)$

and $off + 2 \in \text{Mapping.keys (store-cap } m \text{ off cap } 32)$

and $off + 3 \in \text{Mapping.keys (store-cap } m \text{ off cap } 32)$

and $off + 4 \in \text{Mapping.keys (store-cap } m \text{ off cap } 32)$

and $off + 5 \in \text{Mapping.keys (store-cap } m \text{ off cap } 32)$

and $off + 6 \in \text{Mapping.keys (store-cap } m \text{ off cap } 32)$

and $off + 7 \in \text{Mapping.keys (store-cap } m \text{ off cap } 32)$

and $off + 8 \in \text{Mapping.keys (store-cap } m \text{ off cap } 32)$

and $off + 9 \in \text{Mapping.keys (store-cap } m \text{ off cap } 32)$

and $off + 10 \in \text{Mapping.keys (store-cap } m \text{ off cap } 32)$

and $off + 11 \in \text{Mapping.keys (store-cap } m \text{ off cap } 32)$

and $off + 12 \in \text{Mapping.keys (store-cap } m \text{ off cap } 32)$

and $off + 13 \in \text{Mapping.keys (store-cap } m \text{ off cap } 32)$

and $off + 14 \in \text{Mapping.keys (store-cap } m \text{ off cap } 32)$

and $off + 15 \in \text{Mapping.keys (store-cap } m \text{ off cap } 32)$

and $off + 16 \in \text{Mapping.keys (store-cap } m \text{ off cap } 32)$

and $off + 17 \in \text{Mapping.keys (store-cap } m \text{ off cap } 32)$

and $off + 18 \in \text{Mapping.keys (store-cap } m \text{ off cap } 32)$

and $off + 19 \in \text{Mapping.keys (store-cap } m \text{ off cap } 32)$

and $off + 20 \in \text{Mapping.keys (store-cap } m \text{ off cap } 32)$

and $off + 21 \in \text{Mapping.keys (store-cap } m \text{ off cap } 32)$

and $off + 22 \in \text{Mapping.keys (store-cap } m \text{ off cap } 32)$

and $off + 23 \in \text{Mapping.keys (store-cap } m \text{ off cap } 32)$

and $off + 24 \in \text{Mapping.keys (store-cap } m \text{ off cap } 32)$

and $off + 25 \in \text{Mapping.keys (store-cap } m \text{ off cap } 32)$

and $off + 26 \in \text{Mapping.keys (store-cap } m \text{ off cap } 32)$

and $off + 27 \in \text{Mapping.keys (store-cap } m \text{ off cap } 32)$

and $off + 28 \in \text{Mapping.keys (store-cap } m \text{ off cap } 32)$

and $off + 29 \in \text{Mapping.keys (store-cap } m \text{ off cap } 32)$

and $off + 30 \in \text{Mapping.keys (store-cap } m \text{ off cap } 32)$

and $off + 31 \in \text{Mapping.keys (store-cap } m \text{ off cap } 32)$

proof – **qed** (*simp add: suc-of-32 domIff eval-nat-numeral(3) numeral-Bit0*)+

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 – **qed** (*simp add: stored-cap-instant-correctness suc-of-32 eval-nat-numeral(3) numeral-Bit0*)⁺

corollary *store-cap-imp-is-contiguous-cap*:
is-contiguous-cap (store-cap m off cap 32) (mem-capability.truncate cap) off 32
using *memval-byte-not-memcap*
by (*simp add: eval-nat-numeral(3) numeral-Bit0, blast*)

lemma *stored-tval-is-cap*:
assumes *val = Cap-v v*
shows *is-cap (content (store-tval obj off val)) off*
using *memval-byte-not-memcap sizeof-def store-cap-correct-val(1)*
by (*auto simp add: assms store-tval-def conjI sizeof-def store-cap-correct-dom(1) split: cval.split*)

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|_τ*
using *assms store-tval-def*
by (*simp add: sizeof-def store-cap-imp-is-contiguous-cap*)

lemma *decode-encoded-u16-in-mem*:
cat-u16 (retrieve-bytes (content (store-tval obj off (Uint16-v x))) off |Uint16|_τ)
= *x*

proof –
have *of-byte (the (Mapping.lookup (store-bytes (content obj) off (u16-split x)) off)) = u16-split x ! 0*
by (*smt (verit, best) Some-to-the length-nth-simps(3) memval.sel(1) stored-bytes-instant-correctness u16-split-realise*)
also have *of-byte (the (Mapping.lookup (store-bytes (content obj) off (u16-split*

$x)) (Suc\ off))) = (u16-split\ x) ! 1$
by (*metis One-nat-def length-nth-simps(3) memval.sel(1) nth-Cons-Suc option.sel store-bytes.simps(2) stored-bytes-instant-correctness u16-split-realise*)
ultimately show *?thesis*
by (*clarsimp simp add: sizeof-def store-tval-def eval-nat-numeral(3), simp add: numeral-Bit0*)
(metis cat-flatten-u16-eq length-nth-simps(3) nth-Cons-Suc u16-split-realise)
qed

lemma *decode-encoded-s16-in-mem:*

cat-s16 (retrieve-bytes (content (store-tval obj off (Sint16-v x))) off |Sint16|_τ) = x

proof –

have *of-byte (the (Mapping.lookup (store-bytes (content obj) off (s16-split x)) off)) = s16-split x ! 0*

by (*smt (verit, best) Some-to-the length-nth-simps(3) memval.sel(1) stored-bytes-instant-correctness s16-split-realise*)

also have *of-byte (the (Mapping.lookup (store-bytes (content obj) off (s16-split x)) (Suc off))) = (s16-split x) ! 1*

by (*metis One-nat-def length-nth-simps(3) memval.sel(1) nth-Cons-Suc option.sel store-bytes.simps(2) stored-bytes-instant-correctness s16-split-realise*)

ultimately show *?thesis*

by (*clarsimp simp add: sizeof-def store-tval-def eval-nat-numeral(3), simp add: numeral-Bit0*)

(metis cat-flatten-s16-eq length-nth-simps(3) nth-Cons-Suc s16-split-realise)

qed

lemma *decode-encoded-u32-in-mem:*

cat-u32 (retrieve-bytes (content (store-tval obj off (Uint32-v x))) off |Uint32|_τ) = x

proof –

have *fst: of-byte (the (Mapping.lookup (store-bytes (content obj) off (flatten-u32 x)) off)) = flatten-u32 x ! 0*

by (*metis length-nth-simps(3) memval.sel(1) option.sel stored-bytes-instant-correctness u32-split-realise*)

have *snd: of-byte (the (Mapping.lookup (store-bytes (content obj) off (flatten-u32 x)) (Suc off))) = flatten-u32 x ! 1*

by (*metis One-nat-def length-nth-simps(3) length-nth-simps(4) memval.sel(1) option.sel store-bytes.simps(2) stored-bytes-instant-correctness u32-split-realise*)

have *trd: of-byte (the (Mapping.lookup (store-bytes (content obj) off (flatten-u32 x)) (Suc (Suc off)))) = flatten-u32 x ! 2*

by (*metis One-nat-def Suc-1 length-nth-simps(3) length-nth-simps(4) memval.sel(1) option.sel store-bytes.simps(2) stored-bytes-instant-correctness u32-split-realise*)

have *fth: of-byte (the (Mapping.lookup (store-bytes (content obj) off (flatten-u32 x)) (Suc (Suc (Suc off))))) = flatten-u32 x ! 3*

by (*metis Some-to-the length-nth-simps(3) length-nth-simps(4) memval.sel(1) numeral-3-eq-3 store-bytes.simps(2) stored-bytes-instant-correctness u32-split-realise*)

show *?thesis*

by (*clarsimp simp add: sizeof-def store-tval-def eval-nat-numeral(3), simp add:*

numeral-*Bit0*)
 (smt (verit, del-insts) fst snd trd fth One-nat-def Suc-1 eval-nat-numeral(3)
 length-nth-simps(3) length-nth-simps(4) u32-split-realise word-rcat-rsplit)
qed

lemma *decode-encoded-s32-in-mem:*

cat-s32 (retrieve-bytes (content (store-tval obj off (Sint32-v x))) off |Sint32|_τ) =
 x

proof –

have fst: of-byte (the (Mapping.lookup (store-bytes (content obj) off (flatten-s32
 x)) off)) = flatten-s32 x ! 0

by (metis length-nth-simps(3) memval.sel(1) option.sel stored-bytes-instant-correctness
 s32-split-realise)

have snd: of-byte (the (Mapping.lookup (store-bytes (content obj) off (flatten-s32
 x)) (Suc off))) = flatten-s32 x ! 1

by (metis One-nat-def length-nth-simps(3) length-nth-simps(4) memval.sel(1)
 option.sel store-bytes.simps(2) stored-bytes-instant-correctness s32-split-realise)

have trd: of-byte (the (Mapping.lookup (store-bytes (content obj) off (flatten-s32
 x)) (Suc (Suc off)))) = flatten-s32 x ! 2

by (metis One-nat-def Suc-1 length-nth-simps(3) length-nth-simps(4) mem-
 val.sel(1) option.sel store-bytes.simps(2) stored-bytes-instant-correctness s32-split-realise)

have fth: of-byte (the (Mapping.lookup (store-bytes (content obj) off (flatten-s32
 x)) (Suc (Suc (Suc off)))))) = flatten-s32 x ! 3

by (metis Some-to-the length-nth-simps(3) length-nth-simps(4) memval.sel(1)
 numeral-3-eq-3 store-bytes.simps(2) stored-bytes-instant-correctness s32-split-realise)

show ?thesis

by (clarsimp simp add: sizeof-def store-tval-def eval-nat-numeral(3), simp add:
 numeral-*Bit0*)

(smt (verit, del-insts) fst snd trd fth One-nat-def Suc-1 eval-nat-numeral(3)
 length-nth-simps(3) length-nth-simps(4) s32-split-realise word-rcat-rsplit)

qed

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

using u64-split-realise[**where** ?v=vs]

byclarsimp (metis cat-flatten-u64-eq)

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

using s64-split-realise[**where** ?v=vs]

byclarsimp (metis word-rcat-rsplit)

lemma *decode-encoded-u64-in-mem:*

cat-u64 (retrieve-bytes (content (store-tval obj off (Uint64-v x))) off |Uint64|_τ)
 = x

proof –

have all-bytes: of-byte (the (Mapping.lookup (store-bytes (content obj) off (flatten-u64

$x)) \text{ off})) = \text{flatten-u64 } x ! 0 \wedge$
 $\text{of-byte (the (Mapping.lookup (store-bytes (content obj) off (flatten-u64 } x)) (Suc$
 $\text{off})) = \text{flatten-u64 } x ! 1 \wedge$
 $\text{of-byte (the (Mapping.lookup (store-bytes (content obj) off (flatten-u64 } x)) (Suc$
 $(Suc \text{ off})))) = \text{flatten-u64 } x ! 2 \wedge$
 $\text{of-byte (the (Mapping.lookup (store-bytes (content obj) off (flatten-u64 } x)) (Suc$
 $(Suc (Suc \text{ off})))) = \text{flatten-u64 } x ! 3 \wedge$
 $\text{of-byte (the (Mapping.lookup (store-bytes (content obj) off (flatten-u64 } x)) (Suc$
 $(Suc (Suc (Suc \text{ off})))) = \text{flatten-u64 } x ! 4 \wedge$
 $\text{of-byte (the (Mapping.lookup (store-bytes (content obj) off (flatten-u64 } x)) (Suc$
 $(Suc (Suc (Suc (Suc \text{ off})))) = \text{flatten-u64 } x ! 5 \wedge$
 $\text{of-byte (the (Mapping.lookup (store-bytes (content obj) off (flatten-u64 } x)) (Suc$
 $(Suc (Suc (Suc (Suc (Suc \text{ off})))) = \text{flatten-u64 } x ! 6 \wedge$
 $\text{of-byte (the (Mapping.lookup (store-bytes (content obj) off (flatten-u64 } x)) (Suc$
 $(Suc (Suc (Suc (Suc (Suc (Suc \text{ off})))) = \text{flatten-u64 } x ! 7$
by (smt (verit, best) length-nth-simps(3) length-nth-simps(4) memval.sel(1) op-
tion.sel One-nat-def numeral-2-eq-2 numeral-3-eq-3 numeral-4-eq-4 numeral-5-eq-5
numeral-6-eq-6 numeral-7-eq-7 store-bytes.simps(2) stored-bytes-instant-correctness
u64-split-realise)
show ?thesis
by (clarsimp simp add: sizeof-def store-tval-def eval-nat-numeral(3), simp add:
numeral-Bit0)
(presburger add: all-bytes cat-flatten-u64-contents-eq)
qed

lemma decode-encoded-s64-in-mem:

$\text{cat-s64 (retrieve-bytes (content (store-tval obj off (Sint64-v } x))) \text{ off } |Sint64|_{\tau}) =$
 x

proof –

have all-bytes: $\text{of-byte (the (Mapping.lookup (store-bytes (content obj) off (flatten-s64$
 $x)) \text{ off})) = \text{flatten-s64 } x ! 0 \wedge$
 $\text{of-byte (the (Mapping.lookup (store-bytes (content obj) off (flatten-s64 } x)) (Suc$
 $\text{off})) = \text{flatten-s64 } x ! 1 \wedge$
 $\text{of-byte (the (Mapping.lookup (store-bytes (content obj) off (flatten-s64 } x)) (Suc$
 $(Suc \text{ off})))) = \text{flatten-s64 } x ! 2 \wedge$
 $\text{of-byte (the (Mapping.lookup (store-bytes (content obj) off (flatten-s64 } x)) (Suc$
 $(Suc (Suc \text{ off})))) = \text{flatten-s64 } x ! 3 \wedge$
 $\text{of-byte (the (Mapping.lookup (store-bytes (content obj) off (flatten-s64 } x)) (Suc$
 $(Suc (Suc (Suc \text{ off})))) = \text{flatten-s64 } x ! 4 \wedge$
 $\text{of-byte (the (Mapping.lookup (store-bytes (content obj) off (flatten-s64 } x)) (Suc$
 $(Suc (Suc (Suc (Suc \text{ off})))) = \text{flatten-s64 } x ! 5 \wedge$
 $\text{of-byte (the (Mapping.lookup (store-bytes (content obj) off (flatten-s64 } x)) (Suc$
 $(Suc (Suc (Suc (Suc (Suc \text{ off})))) = \text{flatten-s64 } x ! 6 \wedge$
 $\text{of-byte (the (Mapping.lookup (store-bytes (content obj) off (flatten-s64 } x)) (Suc$
 $(Suc (Suc (Suc (Suc (Suc (Suc \text{ off})))) = \text{flatten-s64 } x ! 7$
by (smt (verit, best) length-nth-simps(3) length-nth-simps(4) memval.sel(1) op-
tion.sel One-nat-def numeral-2-eq-2 numeral-3-eq-3 numeral-4-eq-4 numeral-5-eq-5
numeral-6-eq-6 numeral-7-eq-7 store-bytes.simps(2) stored-bytes-instant-correctness
s64-split-realise)

```

show ?thesis
  by (clarsimp simp add: sizeof-def store-tval-def eval-nat-numeral(3), simp add:
numeral-Bit0)
    (presburger add: all-bytes cat-flatten-s64-contents-eq)
qed

lemma retrieve-stored-tval-cap:
  assumes val = Cap-v v
  shows retrieve-tval (store-tval obj off val) off (memval-type val) True = val
  apply (clarsimp simp add: assms)
  unfolding retrieve-tval-def
  apply (clarsimp simp add: stored-tval-contiguous-cap; safe)
    apply (metis is-contiguous-bytes.simps(2) less-numeral-extra(3))
not0-implies-Suc sizeof-nonzero)
  subgoal
    apply (subgoal-tac is-contiguous-cap (content (store-tval obj off (Cap-v v)))
(get-cap (content (store-tval obj off (Cap-v v))) off) off |Cap| $\tau$ )
      apply clarsimp
      apply (simp only: store-tval-def get-cap-def sizeof-def)
      apply clarsimp
      apply (subst suc-of-32)
      apply (simp only: stored-cap-instant-correctness)
      apply simp
      apply (simp add: mem-capability.extend-def mem-capability.truncate-def)
      apply (metis ctype.simps(81) get-cap-def is-contiguous-cap.simps(2) mem-
val-size-cap sizeof-def stored-tval-contiguous-cap suc-of-32)
    done
  subgoal
    using stored-tval-is-cap by auto
  subgoal
    using stored-tval-is-cap by auto
  subgoal
    using stored-tval-is-cap by auto
  subgoal
    using stored-tval-is-cap by auto
  subgoal
    using stored-tval-is-cap by auto
  subgoal
    apply (metis is-contiguous-bytes.simps(2) less-numeral-extra(3) not0-implies-Suc
sizeof-nonzero)
  done
  subgoal
    apply (subgoal-tac is-contiguous-cap (content (store-tval obj off (Cap-v v)))
(get-cap (content (store-tval obj off (Cap-v v))) off) off |Cap| $\tau$ )
      apply clarsimp
      apply (simp only: store-tval-def get-cap-def sizeof-def)
      apply clarsimp
      apply (subst suc-of-32)
      apply (simp only: stored-cap-instant-correctness)
      apply simp
      apply (simp add: mem-capability.extend-def mem-capability.truncate-def)

```



```

    apply (metis cctype.simps(81) get-cap-def is-contiguous-cap.simps(2) mem-
val-size-cap sizeof-def stored-tval-contiguous-cap suc-of-32)
  done
  subgoal
    using stored-tval-is-cap by auto
  subgoal
    using stored-tval-is-cap by auto
  subgoal
    apply (metis cctype.simps(81) is-contiguous-bytes.simps(2) sizeof-def suc-of-32)
  done
  subgoal
    apply (subgoal-tac is-contiguous-cap (content (store-tval obj off (Cap-v v)))
(get-cap (content (store-tval obj off (Cap-v v))) off) off |Cap| $\tau$ ))
    apply clarsimp
    apply (simp only: store-tval-def get-cap-def sizeof-def)
    apply clarsimp
    apply (metis is-contiguous-zeros-def le-eq-less-or-eq less-add-same-cancel1 mem-
val-memcap-not-byte option.sel suc-of-32 zero-less-Suc)
    apply (metis cctype.simps(81) get-cap-def is-contiguous-cap.simps(2) mem-
val-size-cap sizeof-def stored-tval-contiguous-cap suc-of-32)
  done
  subgoal
    using stored-tval-is-cap by auto
  subgoal
    using stored-tval-is-cap by auto
  subgoal
    using stored-tval-is-cap by auto
  subgoal
    using stored-tval-is-cap by auto
  subgoal
    using stored-tval-is-cap by auto
  subgoal
    apply (metis gr-implies-not-zero is-contiguous-bytes.simps(2) old.nat.exhaust
sizeof-nonzero)
  done
  subgoal
    apply (subgoal-tac is-contiguous-cap (content (store-tval obj off (Cap-v v)))
(get-cap (content (store-tval obj off (Cap-v v))) off) off |Cap| $\tau$ ))
    apply clarsimp
    apply (simp only: store-tval-def get-cap-def sizeof-def)
    apply clarsimp
    apply (subst suc-of-32)
    apply (simp only: stored-cap-instant-correctness)
    apply simp
    apply (simp add: mem-capability.extend-def mem-capability.truncate-def)
    apply (metis cctype.simps(81) get-cap-def is-contiguous-cap.simps(2) mem-
val-size-cap sizeof-def stored-tval-contiguous-cap suc-of-32)
  done
  subgoal
    using stored-tval-is-cap by auto
  subgoal

```

```

    using stored-tval-is-cap by auto
  done

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 )))
  apply (clarsimp simp add: assms)
  unfolding retrieve-tval-def
  apply (clarsimp simp add: stored-tval-contiguous-cap; safe)
  subgoal
    apply (metis is-contiguous-bytes.simps(2) less-numeral-extra(3) not0-implies-Suc
sizeof-nonzero)
  done
  subgoal
    apply (subgoal-tac is-contiguous-cap (content (store-tval obj off (Cap-v v)))
(get-cap (content (store-tval obj off (Cap-v v))) off) off |Cap| $\tau$ )
    apply clarsimp
    apply (simp only: store-tval-def get-cap-def sizeof-def)
    apply clarsimp
    apply (metis is-contiguous-zeros-def le-eq-less-or-eq less-add-same-cancel1 mem-
val-memcap-not-byte option.sel suc-of-32 zero-less-Suc)
    apply (metis ctype.simps(81) get-cap-def is-contiguous-cap.simps(2) mem-
val-size-cap sizeof-def stored-tval-contiguous-cap suc-of-32)
  done
  subgoal
    using stored-tval-is-cap by auto
  subgoal
    using stored-tval-is-cap by auto
  subgoal
    using stored-tval-is-cap by auto
  subgoal
    using stored-tval-is-cap by auto
  subgoal
    apply (metis is-contiguous-bytes.simps(2) less-numeral-extra(3) not0-implies-Suc
sizeof-nonzero)
  done
  subgoal
    apply (subgoal-tac is-contiguous-cap (content (store-tval obj off (Cap-v v)))
(get-cap (content (store-tval obj off (Cap-v v))) off) off |Cap| $\tau$ )
    apply clarsimp
    apply (simp only: store-tval-def get-cap-def sizeof-def)
    apply clarsimp
    apply (subst suc-of-32)
    apply (simp only: stored-cap-instant-correctness)
    apply simp
    apply (simp add: mem-capability.extend-def mem-capability.truncate-def)
    apply (metis ctype.simps(81) get-cap-def is-contiguous-cap.simps(2) mem-
val-size-cap sizeof-def stored-tval-contiguous-cap suc-of-32)

```

```

done
subgoal
  using stored-tval-is-cap by auto
subgoal
  using stored-tval-is-cap by auto
done

lemma retrieve-stored-tval-u8:
  assumes val = Uint8-v v
  shows retrieve-tval (store-tval obj off val) off (memval-type val) b = val
  unfolding retrieve-tval-def
proof (clarsimp simp add: assms stored-tval-contiguous-bytes; safe)
  show  $\llbracket \text{off} \in \text{Mapping.keys (content (store-tval obj off (Uint8-v v))); memval-is-cap (the (Mapping.lookup (content (store-tval obj off (Uint8-v v))) \text{off}))}; \text{is-contiguous-bytes (content (store-tval obj off (Uint8-v v))) off | Uint8|}_\tau \rrbracket$ 
     $\implies \text{decode-u8-list (retrieve-bytes (content (store-tval obj off (Uint8-v v))) off | Uint8|}_\tau) = v$ 
    by (simp add: sizeof-def)
next
  show  $\llbracket \text{off} \in \text{Mapping.keys (content (store-tval obj off (Uint8-v v))); memval-is-cap (the (Mapping.lookup (content (store-tval obj off (Uint8-v v))) \text{off}))}; \text{is-contiguous-bytes (content (store-tval obj off (Uint8-v v))) off | Uint8|}_\tau \rrbracket$ 
     $\implies \text{is-contiguous-bytes (content (store-tval obj off (Uint8-v v))) off | Uint8|}_\tau$ 
    by (metis One-nat-def ccval.distinct(15) ccval.distinct(17) ccval.distinct(19) is-contiguous-bytes.simps(2) memval-size-u8 stored-tval-contiguous-bytes)
next
  show  $\llbracket \text{off} \notin \text{Mapping.keys (content (store-tval obj off (Uint8-v v))); is-contiguous-bytes (content (store-tval obj off (Uint8-v v))) off | Uint8|}_\tau \rrbracket$ 
     $\implies \text{decode-u8-list (retrieve-bytes (content (store-tval obj off (Uint8-v v))) off | Uint8|}_\tau) = v$ 
    by (simp add: sizeof-def)
next
  show  $\text{off} \notin \text{Mapping.keys (content (store-tval obj off (Uint8-v v)))} \implies \text{is-contiguous-bytes (content (store-tval obj off (Uint8-v v))) off | Uint8|}_\tau$ 
    by (metis ctype.simps(73) ccval.distinct(15) ccval.distinct(17) ccval.distinct(19) memval-size-u8 sizeof-def stored-tval-contiguous-bytes)
next
  show  $\llbracket \text{memval-is-byte (the (Mapping.lookup (content (store-tval obj off (Uint8-v v))) \text{off})); is-contiguous-bytes (content (store-tval obj off (Uint8-v v))) off | Uint8|}_\tau \rrbracket$ 
     $\implies \text{decode-u8-list (retrieve-bytes (content (store-tval obj off (Uint8-v v))) off | Uint8|}_\tau) = v$ 
    by (clarsimp simp add: sizeof-def store-tval-def)
next
  show  $\text{memval-is-byte (the (Mapping.lookup (content (store-tval obj off (Uint8-v v))) \text{off}))} \implies \text{is-contiguous-bytes (content (store-tval obj off (Uint8-v v))) off | Uint8|}_\tau$ 
    by (metis ctype.simps(73) ccval.distinct(15) ccval.distinct(17) ccval.distinct(19) memval-size-u8 sizeof-def stored-tval-contiguous-bytes)
qed

```

lemma *retrieve-stored-tval-s8*:

assumes $val = \text{Sint8-v } v$
shows $\text{retrieve-tval (store-tval obj off val) off (memval-type val) } b = val$
unfolding *retrieve-tval-def*
proof (*clarsimp simp add: assms stored-tval-contiguous-bytes; safe*)
show $\llbracket \text{off} \in \text{Mapping.keys (content (store-tval obj off (Sint8-v v)))}; \text{memval-is-cap (the (Mapping.lookup (content (store-tval obj off (Sint8-v v))) off))}; \text{is-contiguous-bytes (content (store-tval obj off (Sint8-v v))) off } \llbracket \text{Sint8} \llbracket_{\tau} \rrbracket \rrbracket$
 $\implies \text{decode-u8-list (retrieve-bytes (content (store-tval obj off (Sint8-v v))) off } \llbracket \text{Sint8} \llbracket_{\tau} \rrbracket = \text{SCAST}(8 \text{ signed} \rightarrow 8) v$
by (*simp add: sizeof-def*)
next
show $\llbracket \text{off} \in \text{Mapping.keys (content (store-tval obj off (Sint8-v v)))}; \text{memval-is-cap (the (Mapping.lookup (content (store-tval obj off (Sint8-v v))) off))} \rrbracket$
 $\implies \text{is-contiguous-bytes (content (store-tval obj off (Sint8-v v))) off } \llbracket \text{Sint8} \llbracket_{\tau} \rrbracket$
by (*metis One-nat-def ccval.distinct(33) ccval.distinct(35) ccval.distinct(37) is-contiguous-bytes.simps(2) memval-size-s8 stored-tval-contiguous-bytes*)
next
show $\llbracket \text{off} \notin \text{Mapping.keys (content (store-tval obj off (Sint8-v v)))}; \text{is-contiguous-bytes (content (store-tval obj off (Sint8-v v))) off } \llbracket \text{Sint8} \llbracket_{\tau} \rrbracket \rrbracket$
 $\implies \text{decode-u8-list (retrieve-bytes (content (store-tval obj off (Sint8-v v))) off } \llbracket \text{Sint8} \llbracket_{\tau} \rrbracket = \text{SCAST}(8 \text{ signed} \rightarrow 8) v$
by (*simp add: sizeof-def*)
next
show $\text{off} \notin \text{Mapping.keys (content (store-tval obj off (Sint8-v v)))} \implies \text{is-contiguous-bytes (content (store-tval obj off (Sint8-v v))) off } \llbracket \text{Sint8} \llbracket_{\tau} \rrbracket$
by (*metis cctype.simps(74) ccval.distinct(33) ccval.distinct(35) ccval.distinct(37) memval-size-s8 sizeof-def stored-tval-contiguous-bytes*)
next
show $\llbracket \text{memval-is-byte (the (Mapping.lookup (content (store-tval obj off (Sint8-v v))) off))}; \text{is-contiguous-bytes (content (store-tval obj off (Sint8-v v))) off } \llbracket \text{Sint8} \llbracket_{\tau} \rrbracket \rrbracket$
 $\implies \text{decode-u8-list (retrieve-bytes (content (store-tval obj off (Sint8-v v))) off } \llbracket \text{Sint8} \llbracket_{\tau} \rrbracket = \text{SCAST}(8 \text{ signed} \rightarrow 8) v$
by (*clarsimp simp add: sizeof-def store-tval-def*)
next
show $\text{memval-is-byte (the (Mapping.lookup (content (store-tval obj off (Sint8-v v))) off))} \implies \text{is-contiguous-bytes (content (store-tval obj off (Sint8-v v))) off } \llbracket \text{Sint8} \llbracket_{\tau} \rrbracket$
by (*metis cctype.simps(74) ccval.distinct(33) ccval.distinct(35) ccval.distinct(37) memval-size-s8 sizeof-def stored-tval-contiguous-bytes*)
qed

lemma *retrieve-stored-tval-u16*:

assumes $val = \text{Uint16-v } v$
shows $\text{retrieve-tval (store-tval obj off val) off (memval-type val) } b = val$
unfolding *retrieve-tval-def*
proof (*clarsimp simp add: assms stored-tval-contiguous-bytes; safe*)
show $\llbracket \text{off} \in \text{Mapping.keys (content (store-tval obj off (Uint16-v v)))}; \text{memval-is-cap (the (Mapping.lookup (content (store-tval obj off (Uint16-v v))) off))}; \text{is-contiguous-bytes (content (store-tval obj off (Uint16-v v))) off } \llbracket \text{Uint16} \llbracket_{\tau} \rrbracket \rrbracket$

$\implies \text{cat-u16 (retrieve-bytes (content (store-tval obj off (Uint16-v v))) off |Uint16|_\tau)}$
 $= v$
by (*presburger add: decode-encoded-u16-in-mem*)
next
show $\llbracket \text{off} \in \text{Mapping.keys (content (store-tval obj off (Uint16-v v))); memval-is-cap (the (Mapping.lookup (content (store-tval obj off (Uint16-v v))) off))} \rrbracket$
 $\implies \text{is-contiguous-bytes (content (store-tval obj off (Uint16-v v))) off |Uint16|_\tau}$
by (*metis ccval.distinct(49) ccval.distinct(51) ccval.distinct(53) is-contiguous-bytes.simps(2) memval-size-u16 numeral-2-eq-2 stored-tval-contiguous-bytes*)
next
show $\llbracket \text{off} \notin \text{Mapping.keys (content (store-tval obj off (Uint16-v v))); is-contiguous-bytes (content (store-tval obj off (Uint16-v v))) off |Uint16|_\tau} \rrbracket$
 $\implies \text{cat-u16 (retrieve-bytes (content (store-tval obj off (Uint16-v v))) off |Uint16|_\tau)}$
 $= v$
by (*presburger add: decode-encoded-u16-in-mem*)
next
show $\text{off} \notin \text{Mapping.keys (content (store-tval obj off (Uint16-v v)))} \implies \text{is-contiguous-bytes (content (store-tval obj off (Uint16-v v))) off |Uint16|_\tau}$
by (*metis Suc-1 ccval.distinct(49) ccval.distinct(51) ccval.distinct(53) is-contiguous-bytes.simps(2) memval-size-u16 stored-tval-contiguous-bytes*)
next
show $\llbracket \text{memval-is-byte (the (Mapping.lookup (content (store-tval obj off (Uint16-v v))) off)); is-contiguous-bytes (content (store-tval obj off (Uint16-v v))) off |Uint16|_\tau} \rrbracket$
 $\implies \text{cat-u16 (retrieve-bytes (content (store-tval obj off (Uint16-v v))) off |Uint16|_\tau)}$
 $= v$
by (*presburger add: decode-encoded-u16-in-mem*)
next
show $\text{memval-is-byte (the (Mapping.lookup (content (store-tval obj off (Uint16-v v))) off))} \implies \text{is-contiguous-bytes (content (store-tval obj off (Uint16-v v))) off |Uint16|_\tau}$
by (*metis ctype.simps(75) ccval.distinct(49) ccval.distinct(51) ccval.distinct(53) memval-size-u16 sizeof-def stored-tval-contiguous-bytes*)
qed

lemma *retrieve-stored-tval-s16*:

assumes $\text{val} = \text{Sint16-v } v$
shows $\text{retrieve-tval (store-tval obj off val) off (memval-type val) b} = \text{val}$
unfolding *retrieve-tval-def*
proof (*clarsimp simp add: asms stored-tval-contiguous-bytes; safe*)
show $\llbracket \text{off} \in \text{Mapping.keys (content (store-tval obj off (Sint16-v v))); memval-is-cap (the (Mapping.lookup (content (store-tval obj off (Sint16-v v))) off)); is-contiguous-bytes (content (store-tval obj off (Sint16-v v))) off |Sint16|_\tau} \rrbracket$
 $\implies \text{cat-s16 (retrieve-bytes (content (store-tval obj off (Sint16-v v))) off |Sint16|_\tau)}$
 $= v$
by (*presburger add: decode-encoded-s16-in-mem*)
next
show $\llbracket \text{off} \in \text{Mapping.keys (content (store-tval obj off (Sint16-v v))); memval-is-cap (the (Mapping.lookup (content (store-tval obj off (Sint16-v v))) off))} \rrbracket$
 $\implies \text{is-contiguous-bytes (content (store-tval obj off (Sint16-v v))) off |Sint16|_\tau}$

by (*metis* *ccval.distinct*(63) *ccval.distinct*(65) *ccval.distinct*(67) *is-contiguous-bytes.simps*(2) *memval-size-s16 numeral-2-eq-2 stored-tval-contiguous-bytes*)
next
show $\llbracket \text{off} \notin \text{Mapping.keys} (\text{content} (\text{store-tval obj off} (Sint16-v v))); \text{is-contiguous-bytes} (\text{content} (\text{store-tval obj off} (Sint16-v v))) \text{off} \lfloor Sint16 \rfloor_{\tau} \rrbracket$
 $\implies \text{cat-s16} (\text{retrieve-bytes} (\text{content} (\text{store-tval obj off} (Sint16-v v))) \text{off} \lfloor Sint16 \rfloor_{\tau})$
 $= v$
by (*presburger add: decode-encoded-s16-in-mem*)
next
show $\text{off} \notin \text{Mapping.keys} (\text{content} (\text{store-tval obj off} (Sint16-v v))) \implies \text{is-contiguous-bytes} (\text{content} (\text{store-tval obj off} (Sint16-v v))) \text{off} \lfloor Sint16 \rfloor_{\tau}$
by (*metis* *Suc-1* *ccval.distinct*(63) *ccval.distinct*(65) *ccval.distinct*(67) *is-contiguous-bytes.simps*(2) *memval-size-s16 stored-tval-contiguous-bytes*)
next
show $\llbracket \text{memval-is-byte} (\text{the} (\text{Mapping.lookup} (\text{content} (\text{store-tval obj off} (Sint16-v v))) \text{off})); \text{is-contiguous-bytes} (\text{content} (\text{store-tval obj off} (Sint16-v v))) \text{off} \lfloor Sint16 \rfloor_{\tau} \rrbracket$
 $\implies \text{cat-s16} (\text{retrieve-bytes} (\text{content} (\text{store-tval obj off} (Sint16-v v))) \text{off} \lfloor Sint16 \rfloor_{\tau})$
 $= v$
by (*presburger add: decode-encoded-s16-in-mem*)
next
show $\text{memval-is-byte} (\text{the} (\text{Mapping.lookup} (\text{content} (\text{store-tval obj off} (Sint16-v v))) \text{off})) \implies \text{is-contiguous-bytes} (\text{content} (\text{store-tval obj off} (Sint16-v v))) \text{off} \lfloor Sint16 \rfloor_{\tau}$
by (*metis* *cctype.simps*(76) *ccval.distinct*(63) *ccval.distinct*(65) *ccval.distinct*(67) *memval-size-s16 sizeof-def stored-tval-contiguous-bytes*)
qed

lemma *retrieve-stored-tval-u32*:

assumes $\text{val} = \text{Uint32-v } v$
shows $\text{retrieve-tval} (\text{store-tval obj off val}) \text{off} (\text{memval-type val}) b = \text{val}$
unfolding *retrieve-tval-def*
proof (*clarsimp simp add: assms stored-tval-contiguous-bytes; safe*)
show $\llbracket \text{off} \in \text{Mapping.keys} (\text{content} (\text{store-tval obj off} (\text{Uint32-v } v))); \text{memval-is-cap} (\text{the} (\text{Mapping.lookup} (\text{content} (\text{store-tval obj off} (\text{Uint32-v } v))) \text{off})); \text{is-contiguous-bytes} (\text{content} (\text{store-tval obj off} (\text{Uint32-v } v))) \text{off} \lfloor \text{Uint32} \rfloor_{\tau} \rrbracket$
 $\implies \text{cat-u32} (\text{retrieve-bytes} (\text{content} (\text{store-tval obj off} (\text{Uint32-v } v))) \text{off} \lfloor \text{Uint32} \rfloor_{\tau})$
 $= v$
by (*presburger add: decode-encoded-u32-in-mem*)
next
show $\llbracket \text{off} \in \text{Mapping.keys} (\text{content} (\text{store-tval obj off} (\text{Uint32-v } v))); \text{memval-is-cap} (\text{the} (\text{Mapping.lookup} (\text{content} (\text{store-tval obj off} (\text{Uint32-v } v))) \text{off})) \rrbracket$
 $\implies \text{is-contiguous-bytes} (\text{content} (\text{store-tval obj off} (\text{Uint32-v } v))) \text{off} \lfloor \text{Uint32} \rfloor_{\tau}$
by (*metis* *ccval.distinct*(75) *ccval.distinct*(77) *ccval.distinct*(79) *is-contiguous-bytes.simps*(2) *memval-size-u32 numeral-4-eq-4 stored-tval-contiguous-bytes*)
next
show $\llbracket \text{off} \notin \text{Mapping.keys} (\text{content} (\text{store-tval obj off} (\text{Uint32-v } v))); \text{is-contiguous-bytes} (\text{content} (\text{store-tval obj off} (\text{Uint32-v } v))) \text{off} \lfloor \text{Uint32} \rfloor_{\tau} \rrbracket$
 $\implies \text{cat-u32} (\text{retrieve-bytes} (\text{content} (\text{store-tval obj off} (\text{Uint32-v } v))) \text{off} \lfloor \text{Uint32} \rfloor_{\tau})$
 $= v$

by (*presburger add: decode-encoded-u32-in-mem*)
next
show $\text{off} \notin \text{Mapping.keys} (\text{content} (\text{store-tval obj off} (U\text{int32-v } v))) \implies \text{is-contiguous-bytes} (\text{content} (\text{store-tval obj off} (U\text{int32-v } v))) \text{ off } |U\text{int32}|_\tau$
by (*metis ccval.distinct(77) ccval.distinct(79) is-cap.elims(2) is-contiguous-bytes.simps(2) memval-size-u32 numeral-4-eq-4 stored-tval-contiguous-bytes stored-tval-is-cap*)
next
show $\llbracket \text{memval-is-byte} (\text{the} (\text{Mapping.lookup} (\text{content} (\text{store-tval obj off} (U\text{int32-v } v))) \text{ off})); \text{is-contiguous-bytes} (\text{content} (\text{store-tval obj off} (U\text{int32-v } v))) \text{ off } |U\text{int32}|_\tau \rrbracket \implies \text{cat-u32} (\text{retrieve-bytes} (\text{content} (\text{store-tval obj off} (U\text{int32-v } v))) \text{ off } |U\text{int32}|_\tau) = v$
by (*presburger add: decode-encoded-u32-in-mem*)
next
show $\text{memval-is-byte} (\text{the} (\text{Mapping.lookup} (\text{content} (\text{store-tval obj off} (U\text{int32-v } v))) \text{ off})) \implies \text{is-contiguous-bytes} (\text{content} (\text{store-tval obj off} (U\text{int32-v } v))) \text{ off } |U\text{int32}|_\tau$
by (*metis cctype.simps(77) ccval.distinct(75) ccval.distinct(77) ccval.distinct(79) memval-size-u32 sizeof-def stored-tval-contiguous-bytes*)
qed

lemma *retrieve-stored-tval-s32*:
assumes $val = S\text{int32-v } v$
shows $\text{retrieve-tval} (\text{store-tval obj off } val) \text{ off} (\text{memval-type } val) b = val$
unfolding *retrieve-tval-def*
proof (*clarsimp simp add: assms stored-tval-contiguous-bytes; safe*)
show $\llbracket \text{off} \in \text{Mapping.keys} (\text{content} (\text{store-tval obj off} (S\text{int32-v } v))); \text{memval-is-cap} (\text{the} (\text{Mapping.lookup} (\text{content} (\text{store-tval obj off} (S\text{int32-v } v))) \text{ off})); \text{is-contiguous-bytes} (\text{content} (\text{store-tval obj off} (S\text{int32-v } v))) \text{ off } |S\text{int32}|_\tau \rrbracket \implies \text{cat-s32} (\text{retrieve-bytes} (\text{content} (\text{store-tval obj off} (S\text{int32-v } v))) \text{ off } |S\text{int32}|_\tau) = v$
by (*presburger add: decode-encoded-s32-in-mem*)
next
show $\llbracket \text{off} \in \text{Mapping.keys} (\text{content} (\text{store-tval obj off} (S\text{int32-v } v))); \text{memval-is-cap} (\text{the} (\text{Mapping.lookup} (\text{content} (\text{store-tval obj off} (S\text{int32-v } v))) \text{ off})) \rrbracket \implies \text{is-contiguous-bytes} (\text{content} (\text{store-tval obj off} (S\text{int32-v } v))) \text{ off } |S\text{int32}|_\tau$
by (*metis cctype.simps(78) ccval.distinct(85) ccval.distinct(87) ccval.distinct(89) flatten-s32-length memval-size-s32-eq-word-split-len sizeof-def stored-tval-contiguous-bytes*)
next
show $\llbracket \text{off} \notin \text{Mapping.keys} (\text{content} (\text{store-tval obj off} (S\text{int32-v } v))); \text{is-contiguous-bytes} (\text{content} (\text{store-tval obj off} (S\text{int32-v } v))) \text{ off } |S\text{int32}|_\tau \rrbracket \implies \text{cat-s32} (\text{retrieve-bytes} (\text{content} (\text{store-tval obj off} (S\text{int32-v } v))) \text{ off } |S\text{int32}|_\tau) = v$
by (*presburger add: decode-encoded-s32-in-mem*)
next
show $\text{off} \notin \text{Mapping.keys} (\text{content} (\text{store-tval obj off} (S\text{int32-v } v))) \implies \text{is-contiguous-bytes} (\text{content} (\text{store-tval obj off} (S\text{int32-v } v))) \text{ off } |S\text{int32}|_\tau$
by (*metis ccval.distinct(85) ccval.distinct(87) ccval.distinct(89) is-contiguous-bytes.simps(2) less-numeral-extra(3) not0-implies-Suc sizeof-nonzero stored-tval-contiguous-bytes*)
next

show $\llbracket \text{memval-is-byte (the (Mapping.lookup (content (store-tval obj off (Sint32-v v))) off)); is-contiguous-bytes (content (store-tval obj off (Sint32-v v))) off |Sint32|_\tau} \rrbracket$
 $\implies \text{cat-s32 (retrieve-bytes (content (store-tval obj off (Sint32-v v))) off |Sint32|_\tau)}$
 $= v$
by (*presburger add: decode-encoded-s32-in-mem*)
next
show $\text{memval-is-byte (the (Mapping.lookup (content (store-tval obj off (Sint32-v v))) off))} \implies \text{is-contiguous-bytes (content (store-tval obj off (Sint32-v v))) off |Sint32|_\tau}$
by (*metis ctype.simps(78) ccval.distinct(85) ccval.distinct(87) ccval.simps(100) flatten-s32-length memval-size-s32-eq-word-split-len sizeof-def stored-tval-contiguous-bytes*)
qed

lemma *retrieve-stored-tval-u64*:

assumes $\text{val} = \text{Uint64-v } v$
shows $\text{retrieve-tval (store-tval obj off val) off (memval-type val) } b = \text{val}$
unfolding *retrieve-tval-def*
proof (*clarsimp simp add: assms stored-tval-contiguous-bytes; safe*)
show $\llbracket \text{off} \in \text{Mapping.keys (content (store-tval obj off (Uint64-v v))); memval-is-cap (the (Mapping.lookup (content (store-tval obj off (Uint64-v v))) off)); is-contiguous-bytes (content (store-tval obj off (Uint64-v v))) off |Uint64|_\tau} \rrbracket$
 $\implies \text{cat-u64 (retrieve-bytes (content (store-tval obj off (Uint64-v v))) off |Uint64|_\tau)}$
 $= v$
by (*presburger add: decode-encoded-u64-in-mem*)
next
show $\llbracket \text{off} \in \text{Mapping.keys (content (store-tval obj off (Uint64-v v))); memval-is-cap (the (Mapping.lookup (content (store-tval obj off (Uint64-v v))) off))} \rrbracket$
 $\implies \text{is-contiguous-bytes (content (store-tval obj off (Uint64-v v))) off |Uint64|_\tau}$
by (*metis ctype.simps(79) ccval.distinct(93) ccval.distinct(95) ccval.distinct(97) memval-size-u64 sizeof-def stored-tval-contiguous-bytes*)
next
show $\llbracket \text{off} \notin \text{Mapping.keys (content (store-tval obj off (Uint64-v v))); is-contiguous-bytes (content (store-tval obj off (Uint64-v v))) off |Uint64|_\tau} \rrbracket$
 $\implies \text{cat-u64 (retrieve-bytes (content (store-tval obj off (Uint64-v v))) off |Uint64|_\tau)}$
 $= v$
by (*presburger add: decode-encoded-u64-in-mem*)
next
show $\text{off} \notin \text{Mapping.keys (content (store-tval obj off (Uint64-v v)))} \implies \text{is-contiguous-bytes (content (store-tval obj off (Uint64-v v))) off |Uint64|_\tau}$
by (*metis ctype.simps(79) ccval.distinct(95) ccval.distinct(97) is-cap.elims(1) memval-size-u64 sizeof-def stored-tval-contiguous-bytes stored-tval-is-cap*)
next
show $\llbracket \text{memval-is-byte (the (Mapping.lookup (content (store-tval obj off (Uint64-v v))) off)); is-contiguous-bytes (content (store-tval obj off (Uint64-v v))) off |Uint64|_\tau} \rrbracket$
 $\implies \text{cat-u64 (retrieve-bytes (content (store-tval obj off (Uint64-v v))) off |Uint64|_\tau)}$
 $= v$
by (*presburger add: decode-encoded-u64-in-mem*)
next
show $\text{memval-is-byte (the (Mapping.lookup (content (store-tval obj off (Uint64-v v))) off))}$

$v))) \text{ off})) \implies \text{is-contiguous-bytes (content (store-tval obj off (Uint64-v v))) off |Uint64|}_\tau$
by (*metis ctype.simps(79) ccval.distinct(93) ccval.distinct(95) ccval.distinct(97) memval-size-u64 sizeof-def stored-tval-contiguous-bytes*)
qed

lemma *retrieve-stored-tval-s64*:

assumes $\text{val} = \text{Sint64-v } v$

shows $\text{retrieve-tval (store-tval obj off val) off (memval-type val) b} = \text{val}$

unfolding *retrieve-tval-def*

proof (*clarsimp simp add: assms stored-tval-contiguous-bytes; safe*)

show $\llbracket \text{off} \in \text{Mapping.keys (content (store-tval obj off (Sint64-v v))); memval-is-cap (the (Mapping.lookup (content (store-tval obj off (Sint64-v v))) off))} \rrbracket$

$\implies \text{is-contiguous-bytes (content (store-tval obj off (Sint64-v v))) off |Sint64|}_\tau$

$\implies \text{cat-s64 (retrieve-bytes (content (store-tval obj off (Sint64-v v))) off |Sint64|}_\tau)$

$= v$

by (*presburger add: decode-encoded-s64-in-mem*)

next

show $\llbracket \text{off} \in \text{Mapping.keys (content (store-tval obj off (Sint64-v v))); memval-is-cap (the (Mapping.lookup (content (store-tval obj off (Sint64-v v))) off))} \rrbracket$

$\implies \text{is-contiguous-bytes (content (store-tval obj off (Sint64-v v))) off |Sint64|}_\tau$

by (*metis ctype.simps(80) ccval.distinct(101) ccval.distinct(103) ccval.distinct(99) memval-size-s64 sizeof-def stored-tval-contiguous-bytes*)

next

show $\llbracket \text{off} \notin \text{Mapping.keys (content (store-tval obj off (Sint64-v v))); is-contiguous-bytes (content (store-tval obj off (Sint64-v v))) off |Sint64|}_\tau \rrbracket$

$\implies \text{cat-s64 (retrieve-bytes (content (store-tval obj off (Sint64-v v))) off |Sint64|}_\tau)$

$= v$

by (*presburger add: decode-encoded-s64-in-mem*)

next

show $\text{off} \notin \text{Mapping.keys (content (store-tval obj off (Sint64-v v)))} \implies \text{is-contiguous-bytes (content (store-tval obj off (Sint64-v v))) off |Sint64|}_\tau$

by (*metis bot-nat-0.not-eq-extremum ccval.distinct(101) ccval.distinct(103) ccval.distinct(99) is-contiguous-bytes.simps(2) list-decode.cases sizeof-nonzero stored-tval-contiguous-bytes*)

next

show $\llbracket \text{memval-is-byte (the (Mapping.lookup (content (store-tval obj off (Sint64-v v))) off)); is-contiguous-bytes (content (store-tval obj off (Sint64-v v))) off |Sint64|}_\tau \rrbracket$

$\implies \text{cat-s64 (retrieve-bytes (content (store-tval obj off (Sint64-v v))) off |Sint64|}_\tau)$

$= v$

by (*presburger add: decode-encoded-s64-in-mem*)

next

show $\text{memval-is-byte (the (Mapping.lookup (content (store-tval obj off (Sint64-v v))) off))} \implies \text{is-contiguous-bytes (content (store-tval obj off (Sint64-v v))) off |Sint64|}_\tau$

by (*metis ctype.simps(80) ccval.distinct(101) ccval.distinct(103) ccval.distinct(99) memval-size-s64 sizeof-def stored-tval-contiguous-bytes*)

qed

lemma *memcap-truncate-extend-equiv*:

mem-capability.extend (mem-capability.truncate c) (| tag = tag c |) = c
by (*simp add: mem-capability.extend-def mem-capability.truncate-def*)

corollary *Acap-truncate-extend-equiv*:

mem-capability.extend (of-cap (ACap (mem-capability.truncate c) n)) (| tag = tag c |) = c
by *clarsimp (blast intro: memcap-truncate-extend-equiv)*

lemma *memcap-truncate-extend-gen*:

mem-capability.extend (mem-capability.truncate c) (| tag = b |) = c (| tag := b |)
by (*simp add: mem-capability.extend-def mem-capability.truncate-def*)

corollary *Acap-truncate-extend-gen*:

mem-capability.extend (of-cap (ACap (mem-capability.truncate c) n)) (| tag = b |) = c (| tag := b |)
by *clarsimp (blast intro: memcap-truncate-extend-gen)*

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
by (*clarsimp simp add: assms retrieve-tval-def store-tval-def sizeof-def get-cap-def*
memcap-truncate-extend-gen memval-is-byte-def split: bool.split)

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 $\forall v. val \neq Cap-v v$
and $\forall v n. val \neq Cap-v-frag v n$
shows *retrieve-tval (store-tval obj off val) off (memval-type val) b = val*
using *retrieve-stored-tval-prim[where ?obj=obj and ?off=off and ?val=val*
**and ?b=b]
by (*clarsimp simp add: assms split: cval.split*)**

lemma *retrieve-stored-tval-with-perm-cap-load*:

assumes *val ≠ Undef*
and $\forall v n. val \neq Cap-v-frag v n$
shows *retrieve-tval (store-tval obj off val) off (memval-type val) True = val*
using *retrieve-stored-tval-prim[where ?obj=obj and ?off=off and ?val=val*
**and ?b=True]
retrieve-stored-tval-cap[where ?obj=obj and ?off=off and ?val=val]
by (*clarsimp simp add: assms split: cval.split*)**

lemma *store-bytes-domain-1*:

assumes $x + \text{length } vs \leq n$
shows $\text{Mapping.lookup } (\text{store-bytes } m \ n \ vs) \ x = \text{Mapping.lookup } m \ x$
using *assms*
by (*induct vs arbitrary: x m n*) *simp-all*

lemma *store-bytes-domain-2*:
assumes $n + \text{length } vs \leq x$
shows $\text{Mapping.lookup } (\text{store-bytes } m \ n \ vs) \ x = \text{Mapping.lookup } m \ x$
using *assms*
by (*induct vs arbitrary: x m n*) *simp-all*

lemma *store-bytes-keys-1*:
 $\text{Set.filter } (\lambda x. x + \text{length } vs \leq n) (\text{Mapping.keys } m) =$
 $\text{Set.filter } (\lambda x. x + \text{length } vs \leq n) (\text{Mapping.keys } (\text{store-bytes } m \ n \ vs))$
by (*induct vs arbitrary: m n*)
(simp, smt (verit, best) Collect-cong Set.filter-def keys-is-none-rep store-bytes-domain-1)

lemma *store-bytes-keys-2*:
 $\text{Set.filter } (\lambda x. n + \text{length } vs \leq x) (\text{Mapping.keys } m) =$
 $\text{Set.filter } (\lambda x. n + \text{length } vs \leq x) (\text{Mapping.keys } (\text{store-bytes } m \ n \ vs))$
by (*induct vs arbitrary: m n*)
(simp, smt (verit, best) Collect-cong Set.filter-def keys-is-none-rep store-bytes-domain-2)

lemma *store-cap-domain-1*:
assumes $x + n \leq p$
shows $\text{Mapping.lookup } (\text{store-cap } m \ p \ c \ n) \ x = \text{Mapping.lookup } m \ x$
using *assms*
by (*induct n arbitrary: x m p*) *simp-all*

lemma *store-cap-domain-2*:
assumes $p + n \leq x$
shows $\text{Mapping.lookup } (\text{store-cap } m \ p \ c \ n) \ x = \text{Mapping.lookup } m \ x$
using *assms*
by (*induct n arbitrary: x m p*) *simp-all*

lemma *store-cap-keys-1*:
 $\text{Set.filter } (\lambda x. x + n \leq p) (\text{Mapping.keys } m) =$
 $\text{Set.filter } (\lambda x. x + n \leq p) (\text{Mapping.keys } (\text{store-cap } m \ p \ c \ n))$
by (*induct n arbitrary: m p*)
(force, smt (verit, best) Collect-cong Set.filter-def keys-is-none-rep store-cap-domain-1)

lemma *store-cap-keys-2*:
 $\text{Set.filter } (\lambda x. p + n \leq x) (\text{Mapping.keys } m) =$
 $\text{Set.filter } (\lambda x. p + n \leq x) (\text{Mapping.keys } (\text{store-cap } m \ p \ c \ n))$
by (*induct n arbitrary: m p*)
(force, smt (verit, best) Collect-cong Set.filter-def keys-is-none-rep store-cap-domain-2)

lemma *store-tags-domain-1*:
assumes $x < n$

shows $Mapping.lookup (store-tag m n b) x = Mapping.lookup m x$
using *assms* **by** *auto*

lemma *store-tags-domain-2*:

assumes $n < x$

shows $Mapping.lookup (store-tag m n b) x = Mapping.lookup m x$
using *assms* **by** *auto*

lemma *store-tags-keys-1*:

$Set.filter (\lambda x. x < n) (Mapping.keys m) =$

$Set.filter (\lambda x. x < n) (Mapping.keys (store-tag m n b))$

by *fastforce*

lemma *store-tags-keys-2*:

$Set.filter (\lambda x. n < x) (Mapping.keys m) =$

$Set.filter (\lambda x. n < x) (Mapping.keys (store-tag m n b))$

by *fastforce*

lemma *cap-offset-aligned*:

$(cap-offset n) \bmod |Cap|_\tau = 0$

unfolding *sizeof-def*

by *force*

lemma *store-tags-offset*:

assumes $Set.filter (\lambda x. x \bmod |Cap|_\tau \neq 0) (Mapping.keys m) = \{\}$

shows $Set.filter (\lambda x. x \bmod |Cap|_\tau \neq 0) (Mapping.keys (store-tag m (cap-offset n) b)) = \{\}$

using *assms*

unfolding *sizeof-def*

by *force*

lemma *store-tval-disjoint-bounds*:

assumes $store-tval\ obj\ off\ val = obj'$

and $val \neq Undef$

shows $bounds\ obj = bounds\ obj'$

using *assms*

unfolding *store-tval-def*

by (*clarsimp split: cval.split-asm*)

lemma *store-tval-disjoint-1-content*:

assumes $store-tval\ obj\ off\ val = obj'$

and $val \neq Undef$

and $off' < off$

shows $Mapping.lookup (content\ obj)\ off' = Mapping.lookup (content\ obj')\ off'$

using *assms*

by (*clarsimp simp add: store-tval-def split: cval.split-asm*)

(*presburger add: stored-bytes-prev stored-cap-prev*)**+**

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
using assms

proof (induct n arbitrary: obj obj' off val off')
  case 0
  then show ?case
    by force
next
  case (Suc n)
  then show ?case
    by (metis (no-types, lifting) Suc-eq-plus1 add.assoc le-eq-less-or-eq less-add-same-cancel1
order-le-less-trans plus-1-eq-Suc retrieve-bytes.simps(2) store-tval-disjoint-1-content
zero-less-Suc)
qed

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
  using assms
proof (induct n arbitrary: obj obj' off val off')
  case 0
  then show ?case
    by force
next
  case (Suc n)
  then show ?case
    by (smt (verit, best) add.assoc add.commute domIff is-contiguous-bytes.simps(2)
keys-dom-lookup le-eq-less-or-eq less-add-same-cancel1 order-le-less-trans plus-1-eq-Suc
store-tval-disjoint-1-content zero-less-Suc)
qed

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
  using assms
proof (induct n arbitrary: obj obj' off val off')
  case 0
  then show ?case
    by force
next

```

```

case (Suc n)
then show ?case
  by (smt (verit, best) add.assoc add.commute domIff is-contiguous-cap.simps(2)
keys-dom-lookup le-eq-less-or-eq less-add-same-cancel1 order-le-less-trans plus-1-eq-Suc
store-tval-disjoint-1-content zero-less-Suc)
qed

```

```

lemma store-tval-disjoint-1-tags:
assumes store-tval obj off val = obj'
and val ≠ Undef
and off' + |Cap|τ ≤ off
shows Mapping.lookup (tags obj) off' = Mapping.lookup (tags obj') off'
using assms
by (clarsimp simp add: store-tval-def split: ccval.split-asm)
(metis Mapping.lookup-update-neg add.commute add-cancel-right-right bot-nat-0.extremum-strict
diff-diff-cancel le-add1 le-add-diff-inverse less-diff-conv2 mod-less-divisor mod-less-eq-dividend
sizeof-nonzero)+

```

```

lemma store-tval-disjoint-2-content:
assumes store-tval obj off val = obj'
and val ≠ Undef
and off + |memval-type val|τ ≤ off'
shows Mapping.lookup (content obj) off' = Mapping.lookup (content obj') off'
using assms
proof (clarsimp simp add: store-tval-def split: ccval.split-asm)
show  $\bigwedge x. \llbracket \text{off} + |\text{Cap}|_{\tau} \leq \text{off}'; \text{val} = \text{Cap-v } x; \text{obj}' = \text{obj}(\text{content} := \text{store-cap}$ 
(content obj) off x |Cap|τ, tags := store-tag (tags obj) (cap-offset off) (tag x)) \rrbracket
 $\implies \text{Mapping.lookup (content obj) off}' = \text{Mapping.lookup (store-cap (content$ 
obj) off x |Cap|τ) off}'
by (presburger add: store-cap-domain-2)
qed (force simp add: sizeof-def | metis assms(3) store-bytes-domain-2 flatten-types-length
memval-size-types)+

```

```

lemma store-tval-disjoint-2-content-bytes:
assumes store-tval obj off val = obj'
and val ≠ Undef
and off + |memval-type val|τ ≤ off'
shows retrieve-bytes (content obj) off' n = retrieve-bytes (content obj') off' n
using assms
proof (induct n arbitrary: obj obj' off off' val)
case 0
then show ?case
  by force
next
case (Suc n)
then show ?case
  by (metis less-Suc-eq-le less-or-eq-imp-le retrieve-bytes.simps(2) store-tval-disjoint-2-content)

```

qed

lemma *store-tval-disjoint-2-content-contiguous-bytes*:

assumes *store-tval obj off val = obj'*
and *val ≠ Undef*
and *off + |memval-type val|_τ ≤ off'*
shows *is-contiguous-bytes (content obj) off' n = is-contiguous-bytes (content obj') off' n*
using *assms*
proof (*induct n arbitrary: obj obj' off val off'*)
case *0*
then show *?case*
by force
next
case (*Suc n*)
then show *?case*
by (*smt (verit, best) domIff is-contiguous-bytes.simps(2) keys-dom-lookup le-eq-less-or-eq lessI order-le-less-trans store-tval-disjoint-2-content*)
qed

lemma *store-tval-disjoint-2-content-contiguous-caps*:

assumes *store-tval obj off val = obj'*
and *val ≠ Undef*
and *off + |memval-type val|_τ ≤ off'*
shows *is-contiguous-cap (content obj) cap off' n = is-contiguous-cap (content obj') cap off' n*
using *assms*
proof (*induct n arbitrary: obj obj' off val off'*)
case *0*
then show *?case*
by force
next
case (*Suc n*)
then show *?case*
by (*smt (verit, best) domIff is-contiguous-cap.simps(2) keys-dom-lookup le-eq-less-or-eq lessI order-le-less-trans store-tval-disjoint-2-content*)
qed

lemma *store-tval-disjoint-2-tags*:

assumes *store-tval obj off val = obj'*
and *val ≠ Undef*
and *off + |memval-type val|_τ ≤ off'*
shows *Mapping.lookup (tags obj) off' = Mapping.lookup (tags obj') off'*
using *assms*
by (*clarsimp simp add: store-tval-def split: ccval.split-asm*)
(force simp add: assms(3) sizeof-def)+

lemma *zero-imp-bytes*:

is-contiguous-zeros obj off n ⇒ ¬ is-contiguous-bytes obj off n ⇒ False

```

proof (simp add: is-contiguous-zeros-code, induct n arbitrary: obj off)
  case 0
  then show ?case
    by simp
next
  case (Suc n)
  then show ?case
    using keys-dom-lookup memval-memcap-not-byte
    by fastforce
qed

```

```

lemma retrieve-stored-tval-disjoint-1:
  assumes store-tval obj off val = obj'
    and val ≠ Undef
    and off' + |t|τ ≤ off
  shows retrieve-tval obj off' t b = retrieve-tval obj' off' t b
  using assms
  apply (clarsimp simp add: retrieve-tval-def)
  apply (rule conjI, rule impI, rule conjI, rule impI, rule conjI, rule impI, rule
  conjI, rule impI, rule conjI, rule impI, rule conjI, rule impI, rule conjI, rule impI)
  subgoal
    by (simp split: cctype.split, safe; (force simp add: numeral-2-eq-2 numeral-4-eq-4
  numeral-8-eq-8 sizeof-def)+)
  subgoal
    apply (simp split: cctype.split, safe; (force simp add: sizeof-def numeral-2-eq-2
  numeral-4-eq-4 numeral-8-eq-8)+)
    apply (metis is-contiguous-bytes.simps(2) less-imp-Suc-add sizeof-nonzero)
    done
  subgoal
    apply (intro strip conjI)
    apply (simp split: cctype.split, safe; (force simp add: numeral-2-eq-2 nu-
  meral-4-eq-4 numeral-8-eq-8 sizeof-def)+)
    apply (simp split: cctype.split, metis is-contiguous-bytes.simps(1) is-contiguous-bytes.simps(2)
  old.nat.exhaust)
    done
  subgoal
    using zero-imp-bytes by presburger
  subgoal
    by (smt (z3) store-tval-disjoint-1-content-bytes zero-imp-bytes)
  subgoal
    by (simp add: is-contiguous-zeros-def, metis (no-types, lifting) order-less-le-trans
  store-tval-disjoint-1-content)
  subgoal
    unfolding is-contiguous-zeros-def
    by (smt (verit) cctype.exhaust cctype.simps(73) cctype.simps(74) cctype.simps(75)
  cctype.simps(76) cctype.simps(77) cctype.simps(78) cctype.simps(79) cctype.simps(80)
  get-cap-def keys-is-none-rep less-add-same-cancel1 order-less-le-trans sizeof-nonzero
  store-tval-disjoint-1-content store-tval-disjoint-1-content-bytes store-tval-disjoint-1-content-contiguous-bytes
  store-tval-disjoint-1-content-contiguous-caps store-tval-disjoint-1-tags)

```



```

apply (rule impI, rule conjI, rule impI, rule conjI, rule impI, rule conjI, rule
impI)
  apply (smt (z3) store-tval-disjoint-1-content-bytes zero-imp-bytes)
  apply (smt (verit, best) store-tval-disjoint-1-content-bytes zero-imp-bytes)
  apply (simp add: is-contiguous-zeros-def, smt (z3) le-add-diff-inverse2 store-tval-disjoint-1-content
trans-less-add2)
  apply (rule impI, rule conjI, rule impI)
  apply (simp add: is-contiguous-zeros-def)
  apply (smt (z3) le-add-diff-inverse2 store-tval-disjoint-1-content trans-less-add2)
  subgoal
    apply (rule impI, rule conjI, rule impI, rule conjI, rule impI, rule conjI, rule
impI, rule conjI, rule impI)
    apply (metis is-contiguous-bytes.simps(2) less-nat-zero-code old.nat.exhaust
sizeof-nonzero)
    using store-tval-disjoint-1-content-contiguous-bytes
    apply blast
    apply (metis is-contiguous-bytes.simps(2) less-numeral-extra(3) not0-implies-Suc
sizeof-nonzero)
    apply (rule impI, rule conjI, rule impI, rule conjI, rule impI)
    using store-tval-disjoint-1-content-contiguous-bytes
    apply blast
    apply (simp add: Let-def split: ctype.split; clarsimp, smt (verit, best)
add.commute add-diff-cancel-right' add-le-cancel-left bot-nat-0.not-eq-extremum get-cap-def
le-add-diff-inverse2 le-eq-less-or-eq order-less-le-trans sizeof-nonzero store-tval-disjoint-1-content
store-tval-disjoint-1-content-contiguous-caps store-tval-disjoint-1-tags zero-less-diff)
    apply (metis add.right-neutral add-less-cancel-left domIff keys-dom-lookup
le-add-diff-inverse2 sizeof-nonzero store-tval-disjoint-1-content trans-less-add2)
    apply (smt (z3) keys-is-none-rep less-add-same-cancel1 order-less-le-trans
sizeof-nonzero store-tval-disjoint-1-content store-tval-disjoint-1-content-bytes store-tval-disjoint-1-content-conti)
  done
done

```

```

lemma retrieve-stored-tval-disjoint-2:
  assumes store-tval obj off val = obj'
  and val ≠ Undef
  and off + |memval-type val|τ ≤ off'
  and t = Cap ⇒ off' mod |Cap|τ = 0
  shows retrieve-tval obj off' t b = retrieve-tval obj' off' t b
  using assms(1) assms(2) assms(3)
  apply (clarsimp simp add: retrieve-tval-def)
  apply (rule conjI, rule impI, rule conjI, rule impI, rule conjI, rule impI, rule
conjI, rule impI, rule conjI, rule impI, rule conjI, rule impI, rule conjI, rule impI)
  subgoal
    by (simp split: ctype.split, safe; (force simp add: numeral-2-eq-2 numeral-4-eq-4
numeral-8-eq-8 sizeof-def)+)
  subgoal
    apply (simp split: ctype.split, safe; (force simp add: sizeof-def numeral-2-eq-2
numeral-4-eq-4 numeral-8-eq-8)+)
    apply (metis is-contiguous-bytes.simps(2) less-imp-Suc-add sizeof-nonzero)

```

```

done
subgoal
  apply (rule impI, rule conjI, rule impI)
  apply (simp split: cctype.split, safe; (force simp add: numeral-2-eq-2 numeral-4-eq-4 numeral-8-eq-8 sizeof-def)+)
  apply (simp split: cctype.split, metis is-contiguous-bytes.simps(1) is-contiguous-bytes.simps(2) old.nat.exhaust)
done
subgoal
  using zero-imp-bytes by presburger
subgoal
  by (smt (z3) memval-type.elims store-tval-disjoint-2-content-bytes zero-imp-bytes)
subgoal
  by (smt (z3) is-contiguous-zeros-def le-trans memval-type.elims store-tval-disjoint-2-content)
subgoal
  unfolding is-contiguous-zeros-def
  by (smt (verit) get-cap-def keys-is-none-rep le-trans memval-type.elims store-tval-disjoint-2-content store-tval-disjoint-2-content-bytes store-tval-disjoint-2-content-contiguous-bytes store-tval-disjoint-2-content-contiguous-tags)
subgoal
  apply (rule impI, rule conjI, rule impI, rule conjI, rule impI, rule conjI, rule impI)
  apply (metis (no-types, opaque-lifting) is-contiguous-bytes.simps(2) less-numeral-extra(3) not0-implies-Suc sizeof-nonzero zero-imp-bytes)
  apply (smt (z3) assms(3) store-tval-disjoint-2-content-bytes zero-imp-bytes)
  apply (simp add: is-contiguous-zeros-def, metis assms(3) order-trans store-tval-disjoint-2-content)
  unfolding is-contiguous-zeros-def
  by (smt (verit) assms(3) assms(4) cctype.exhaust cctype.simps(73) cctype.simps(74) cctype.simps(75) cctype.simps(76) cctype.simps(77) cctype.simps(78) cctype.simps(79) cctype.simps(80) get-cap-def keys-is-none-rep mod-0-imp-dvd mod-greater-zero-iff-not-dvd store-tval-disjoint-2-content store-tval-disjoint-2-content-bytes store-tval-disjoint-2-content-contiguous-bytes)
done

```

lemma *type-uniq*:

```

assumes  $\exists x n. ret = Cap\text{-}v\text{-}frag\ x\ n$ 
shows  $ret \neq Uint8\text{-}v\ v1 \wedge ret \neq Sint8\text{-}v\ v2 \wedge ret \neq Uint16\text{-}v\ v3 \wedge ret \neq Sint16\text{-}v\ v4$ 
 $ret \neq Uint32\text{-}v\ v5 \wedge ret \neq Sint32\text{-}v\ v6 \wedge ret \neq Uint64\text{-}v\ v7 \wedge ret \neq Sint64\text{-}v\ v8$ 
 $ret \neq Cap\text{-}v\ v9$ 
using assms
by blast+

```

5 Memory Actions / Operations

definition *alloc* :: $heap \Rightarrow bool \Rightarrow nat \Rightarrow (heap \times cap)$ *result*
where
alloc *h c s* \equiv
 $let\ cap = (\!| block\text{-}id = (next\text{-}block\ h),$
 $offset = 0,$

```

    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 cval ⇒ 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 subsection.

5.1.1 Correctness Properties

lemma *alloc-always-success*:
 $\exists! \text{res. } \text{alloc } h \ c \ s = \text{Success } \text{res}$
 by (*simp add: alloc-def*)

schematic-goal *alloc-updated-heap-and-cap*:
 $\text{alloc } h \ c \ s = \text{Success } (?h', ?cap)$
 by (*fastforce simp add: alloc-def*)

lemma *alloc-never-fails*:
 $\text{alloc } h \ c \ s = \text{Error } e \implies \text{False}$
 by (*simp add: alloc-def*)

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

lemma *alloc-no-null-ret*:
assumes $\text{alloc } h \ c \ s = \text{Success } (h', \text{cap})$
shows $\text{cap} \neq \text{NULL}$
proof –
have *perm-load cap*
 using *assms alloc-def*
 by force
moreover have $\neg \text{perm-load } \text{NULL}$
 unfolding *null-capability-def zero-capability-ext-def zero-mem-capability-ext-def*
 by force
ultimately show *?thesis*
 by blast
qed

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)
using *assms alloc-def*
by *auto*

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)
by (*simp add: free-def*)

lemma *free-false-tag*:
assumes *c ≠ NULL*
and *tag c = False*
shows *free h c = Error (C2Err (TagViolation))*
by (*presburger add: assms free-def*)

lemma *free-global-cap*:
assumes *c ≠ NULL*
and *tag c = True*
and *perm-global c = True*
shows *free h c = Error (LogicErr (Unhandled 0))*
by (*presburger add: assms free-def*)

lemma *free-nonexistent-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))*
using *assms free-def*
by *auto*

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))*
using *free-def assms*
by *force*


```

    and  $c \neq \text{NULL} \implies \text{Mapping.lookup } (\text{heap-map } h') (\text{block-id } c) = \text{Some Freed}$ 
    and  $c \neq \text{NULL} \implies \text{cap} = c \ \& \ \text{tag} := \text{False} \ \&$ 
    and  $c = \text{NULL} \implies (h, c) = (h', \text{cap})$ 
proof –
  assume  $c \neq \text{NULL}$ 
  thus  $\text{tag } c = \text{True}$ 
    using assms unfolding free-def
    by (meson result.simps(4))
next
  assume  $c \neq \text{NULL}$ 
  thus  $\text{perm-global } c = \text{False}$ 
    using assms unfolding free-def
    by (meson result.simps(4))
next
  assume  $c \neq \text{NULL}$ 
  thus  $\text{offset } c = 0$ 
    using assms unfolding free-def
    by (smt (verit, ccfv-SIG) not-None-eq option.simps(4) option.simps(5) result.distinct(1) t.exhaust t.simps(4) t.simps(5))
next
  assume  $c \neq \text{NULL}$ 
  thus  $\exists m. \text{Mapping.lookup } (\text{heap-map } h) (\text{block-id } c) = \text{Some } (\text{Map } m) \wedge$ 
     $\text{bounds } m = (\text{base } c, \text{base } c + \text{len } c)$ 
    using assms unfolding free-def
    by (metis assms free-double-free free-incorrect-bounds free-incorrect-cap-offset free-nonexistent-obj not-Some-eq result.distinct(1) t.exhaust)
next
  assume  $c \neq \text{NULL}$ 
  hence  $h' = h \ \& \ \text{heap-map} := \text{Mapping.update } (\text{block-id } c) \text{ Freed } (\text{heap-map } h) \ \&$ 
    using assms unfolding free-def
    by (smt (verit, ccfv-SIG) free-nonexistent-obj not-Some-eq option.simps(4) option.simps(5) prod.inject result.distinct(1) result.exhaust result.inject(1) t.exhaust t.simps(4) t.simps(5))
  thus  $\text{Mapping.lookup } (\text{heap-map } h') (\text{block-id } c) = \text{Some Freed}$ 
    by fastforce
next
  assume  $c \neq \text{NULL}$ 
  thus  $\text{cap} = c \ \& \ \text{tag} := \text{False} \ \&$ 
    using assms unfolding free-def
    by (smt (verit, ccfv-SIG) not-Some-eq option.simps(4) option.simps(5) prod.inject result.distinct(1) result.inject(1) t.exhaust t.simps(4) t.simps(5))
next
  assume  $c = \text{NULL}$ 
  thus  $(h, c) = (h', \text{cap})$ 
    using free-null assms
    by force
qed

```

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 ≠ NULL*

shows *free h' cap = Error (C2Err TagViolation)*

proof –

have *cap = c (| tag := False) ⇒ tag cap = False*

by *fastforce*

thus *?thesis*

using *assms free-cond(6)[where ?h=h and ?c=c and ?h'=h' and ?cap=cap]*

free-false-tag[where ?c=cap and ?h=h'] free-cond(7)[where ?h=h and ?c=c and ?h'=h' and ?cap=cap]

by *blast*

qed

lemma *free-next-block*:

assumes *free h cap = Success (h', cap')*

shows *next-block h = next-block h'*

proof –

consider *(null) cap = NULL | (non-null) cap ≠ NULL* **by** *blast*

then show *?thesis*

proof (*cases*)

case *null*

then show *?thesis*

using *free-null assms null*

by *simp*

next

case *non-null*

then show *?thesis*

using *assms free-cond-non-null[OF assms non-null]*

unfolding *free-def*

by (*auto split: option.split-asm t.split-asm*)

qed

qed

lemma *load-null-error*:

load h NULL t = Error (C2Err TagViolation)

unfolding *load-def*

by *simp*

lemma *load-false-tag*:

assumes *tag c = False*

shows *load h c t = Error (C2Err TagViolation)*

unfolding *load-def*

using *assms*

by *presburger*

lemma *load-false-perm-load*:

```

assumes tag c = True
  and perm-load c = False
shows load h c t = Error (C2Err PermitLoadViolation)
unfolding load-def
using assms
by presburger

```

```

lemma load-bound-over:
  assumes tag c = True
    and perm-load c = True
    and offset c + |t|τ > base c + len c
  shows load h c t = Error (C2Err LengthViolation)
  unfolding load-def
  using assms
  by presburger

```

```

lemma load-bound-under:
  assumes tag c = True
    and perm-load c = True
    and offset c + |t|τ ≤ base c + len c
    and offset c < base c
  shows load h c t = Error (C2Err LengthViolation)
  unfolding load-def
  using assms
  by presburger

```

```

lemma load-misaligned:
  assumes tag c = True
    and perm-load c = True
    and offset c + |t|τ ≤ base c + len c
    and offset c ≥ base c
    and offset c mod |t|τ ≠ 0
  shows load h c t = Error (C2Err BadAddressViolation)
  unfolding load-def
  using assms
  by force

```

```

lemma load-nonexistent-obj:
  assumes tag c = True
    and perm-load c = True
    and offset c + |t|τ ≤ base c + len c
    and offset c ≥ base c
    and offset c mod |t|τ = 0
    and Mapping.lookup (heap-map h) (block-id c) = None
  shows load h c t = Error (LogicErr MissingResource)
  unfolding load-def
  using assms
  by auto

```

lemma *load-load-after-free*:
assumes $tag\ c = True$
and $perm-load\ c = True$
and $offset\ c + |t|_\tau \leq base\ c + len\ c$
and $offset\ c \geq base\ c$
and $offset\ c \bmod |t|_\tau = 0$
and $Mapping.lookup\ (heap-map\ h)\ (block-id\ c) = Some\ Freed$
shows $load\ h\ c\ t = Error\ (LogicErr\ UseAfterFree)$
unfolding *load-def*
using *assms*
by *fastforce*

lemma *load-cap-on-heap-bounds-fail-1*:
assumes $tag\ c = True$
and $perm-load\ c = True$
and $offset\ c + |t|_\tau \leq base\ c + len\ c$
and $offset\ c \geq base\ c$
and $offset\ c \bmod |t|_\tau = 0$
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 $\neg is-contiguous-zeros\ (content\ m)\ (nat\ (offset\ c))\ |t|_\tau$
and $offset\ c < fst\ (bounds\ m)$
shows $load\ h\ c\ t = Error\ (LogicErr\ BufferOverrun)$
unfolding *load-def retrieve-tval-def*
using *assms*
by *fastforce*

lemma *load-cap-on-heap-bounds-fail-2*:
assumes $tag\ c = True$
and $perm-load\ c = True$
and $offset\ c + |t|_\tau \leq base\ c + len\ c$
and $offset\ c \geq base\ c$
and $offset\ c \bmod |t|_\tau = 0$
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 $\neg is-contiguous-zeros\ (content\ m)\ (nat\ (offset\ c))\ |t|_\tau$
and $offset\ c + |t|_\tau > snd\ (bounds\ m)$
shows $load\ h\ c\ t = Error\ (LogicErr\ BufferOverrun)$
unfolding *load-def retrieve-tval-def*
using *assms*
by *fastforce*

lemma *load-cap-on-membytes-fail*:
assumes $tag\ c = True$
and $perm-load\ c = True$
and $offset\ c + |t|_\tau \leq base\ c + len\ c$
and $offset\ c \geq base\ c$

and $\text{offset } c \bmod |t|_\tau = 0$
and $\text{Mapping.lookup } (\text{heap-map } h) (\text{block-id } c) = \text{Some } (\text{Map } m)$
and $\text{is-contiguous-bytes } (\text{content } m) (\text{nat } (\text{offset } c)) |t|_\tau$
and $t = \text{Cap}$
and $\neg \text{is-contiguous-zeros } (\text{content } m) (\text{nat } (\text{offset } c)) |t|_\tau$
and $\text{offset } c \geq \text{fst } (\text{bounds } m)$
and $\text{offset } c + |t|_\tau \leq \text{snd } (\text{bounds } m)$
shows $\text{load } h \ c \ t = \text{Success } \text{Undef}$
unfolding $\text{load-def retrieve-tval-def}$
using assms
by fastforce

lemma $\text{load-null-cap-on-membytes}$:

assumes $\text{tag } c = \text{True}$
and $\text{perm-load } c = \text{True}$
and $\text{offset } c + |t|_\tau \leq \text{base } c + \text{len } c$
and $\text{offset } c \geq \text{base } c$
and $\text{offset } c \bmod |t|_\tau = 0$
and $\text{Mapping.lookup } (\text{heap-map } h) (\text{block-id } c) = \text{Some } (\text{Map } m)$
and $\text{is-contiguous-bytes } (\text{content } m) (\text{nat } (\text{offset } c)) |t|_\tau$
and $t = \text{Cap}$
and $\text{offset } c \geq \text{fst } (\text{bounds } m)$
and $\text{offset } c + |t|_\tau \leq \text{snd } (\text{bounds } m)$
and $\text{is-contiguous-zeros } (\text{content } m) (\text{nat } (\text{offset } c)) |t|_\tau$
shows $\text{load } h \ c \ t = \text{Success } (\text{Cap-v } \text{NULL})$
unfolding $\text{load-def retrieve-tval-def}$
using assms
by fastforce

lemma $\text{load-u8-on-membytes}$:

assumes $\text{tag } c = \text{True}$
and $\text{perm-load } c = \text{True}$
and $\text{offset } c + |t|_\tau \leq \text{base } c + \text{len } c$
and $\text{offset } c \geq \text{base } c$
and $\text{offset } c \bmod |t|_\tau = 0$
and $\text{Mapping.lookup } (\text{heap-map } h) (\text{block-id } c) = \text{Some } (\text{Map } m)$
and $\text{offset } c \geq \text{fst } (\text{bounds } m)$
and $\text{offset } c + |t|_\tau \leq \text{snd } (\text{bounds } m)$
and $\text{is-contiguous-bytes } (\text{content } m) (\text{nat } (\text{offset } c)) |t|_\tau$
and $t = \text{Uint8}$
shows $\text{load } h \ c \ t = \text{Success } (\text{Uint8-v } (\text{decode-u8-list } (\text{retrieve-bytes } (\text{content } m) (\text{nat } (\text{offset } c)) |t|_\tau)))$
unfolding $\text{load-def retrieve-tval-def}$
using assms
by fastforce

lemma $\text{load-s8-on-membytes}$:

assumes $\text{tag } c = \text{True}$
and $\text{perm-load } c = \text{True}$

```

and offset  $c + |t|_\tau \leq \text{base } c + \text{len } c$ 
and offset  $c \geq \text{base } c$ 
and offset  $c \bmod |t|_\tau = 0$ 
and Mapping.lookup (heap-map  $h$ ) (block-id  $c$ ) = Some (Map  $m$ )
and offset  $c \geq \text{fst}$  (bounds  $m$ )
and offset  $c + |t|_\tau \leq \text{snd}$  (bounds  $m$ )
and is-contiguous-bytes (content  $m$ ) (nat (offset  $c$ ))  $|t|_\tau$ 
and  $t = \text{Sint8}$ 
shows load  $h$   $c$   $t = \text{Success}$  (Sint8-v (decode-s8-list (retrieve-bytes (content  $m$ )
(nat (offset  $c$ ))  $|t|_\tau$ )))
unfolding load-def retrieve-tval-def
using assms
by fastforce

```

lemma *load-u16-on-membytes*:

```

assumes tag  $c = \text{True}$ 
and perm-load  $c = \text{True}$ 
and offset  $c + |t|_\tau \leq \text{base } c + \text{len } c$ 
and offset  $c \geq \text{base } c$ 
and offset  $c \bmod |t|_\tau = 0$ 
and Mapping.lookup (heap-map  $h$ ) (block-id  $c$ ) = Some (Map  $m$ )
and offset  $c \geq \text{fst}$  (bounds  $m$ )
and offset  $c + |t|_\tau \leq \text{snd}$  (bounds  $m$ )
and is-contiguous-bytes (content  $m$ ) (nat (offset  $c$ ))  $|t|_\tau$ 
and  $t = \text{Uint16}$ 
shows load  $h$   $c$   $t = \text{Success}$  (Uint16-v (cat-u16 (retrieve-bytes (content  $m$ ) (nat
(offset  $c$ ))  $|t|_\tau$ )))
unfolding load-def retrieve-tval-def
using assms
by fastforce

```

lemma *load-s16-on-membytes*:

```

assumes tag  $c = \text{True}$ 
and perm-load  $c = \text{True}$ 
and offset  $c + |t|_\tau \leq \text{base } c + \text{len } c$ 
and offset  $c \geq \text{base } c$ 
and offset  $c \bmod |t|_\tau = 0$ 
and Mapping.lookup (heap-map  $h$ ) (block-id  $c$ ) = Some (Map  $m$ )
and offset  $c \geq \text{fst}$  (bounds  $m$ )
and offset  $c + |t|_\tau \leq \text{snd}$  (bounds  $m$ )
and is-contiguous-bytes (content  $m$ ) (nat (offset  $c$ ))  $|t|_\tau$ 
and  $t = \text{Sint16}$ 
shows load  $h$   $c$   $t = \text{Success}$  (Sint16-v (cat-s16 (retrieve-bytes (content  $m$ ) (nat
(offset  $c$ ))  $|t|_\tau$ )))
unfolding load-def retrieve-tval-def
using assms
by fastforce

```

lemma *load-u32-on-membytes*:

```

assumes tag c = True
  and perm-load c = True
  and offset c + |t|τ ≤ base c + len c
  and offset c ≥ base c
  and offset c mod |t|τ = 0
  and Mapping.lookup (heap-map h) (block-id c) = Some (Map m)
  and offset c ≥ fst (bounds m)
  and offset c + |t|τ ≤ snd (bounds m)
  and is-contiguous-bytes (content m) (nat (offset c)) |t|τ
  and t = Uint32
shows load h c t = Success (Uint32-v (cat-u32 (retrieve-bytes (content m) (nat
(offset c)) |t|τ)))
  unfolding load-def retrieve-tval-def
  using assms
  by fastforce

```

lemma load-s32-on-membytes:

```

assumes tag c = True
  and perm-load c = True
  and offset c + |t|τ ≤ base c + len c
  and offset c ≥ base c
  and offset c mod |t|τ = 0
  and Mapping.lookup (heap-map h) (block-id c) = Some (Map m)
  and offset c ≥ fst (bounds m)
  and offset c + |t|τ ≤ snd (bounds m)
  and is-contiguous-bytes (content m) (nat (offset c)) |t|τ
  and t = Sint32
shows load h c t = Success (Sint32-v (cat-s32 (retrieve-bytes (content m) (nat
(offset c)) |t|τ)))
  unfolding load-def retrieve-tval-def
  using assms
  by fastforce

```

lemma load-u64-on-membytes:

```

assumes tag c = True
  and perm-load c = True
  and offset c + |t|τ ≤ base c + len c
  and offset c ≥ base c
  and offset c mod |t|τ = 0
  and Mapping.lookup (heap-map h) (block-id c) = Some (Map m)
  and offset c ≥ fst (bounds m)
  and offset c + |t|τ ≤ snd (bounds m)
  and is-contiguous-bytes (content m) (nat (offset c)) |t|τ
  and t = Uint64
shows load h c t = Success (Uint64-v (cat-u64 (retrieve-bytes (content m) (nat
(offset c)) |t|τ)))
  unfolding load-def retrieve-tval-def
  using assms
  by fastforce

```

lemma *load-s64-on-membytes*:

assumes $tag\ c = True$
and $perm-load\ c = True$
and $offset\ c + |t|_\tau \leq base\ c + len\ c$
and $offset\ c \geq base\ c$
and $offset\ c \bmod |t|_\tau = 0$
and $Mapping.lookup\ (heap-map\ h)\ (block-id\ c) = Some\ (Map\ m)$
and $offset\ c \geq fst\ (bounds\ m)$
and $offset\ c + |t|_\tau \leq snd\ (bounds\ m)$
and $is-contiguous-bytes\ (content\ m)\ (nat\ (offset\ c))\ |t|_\tau$
and $t = Sint64$
shows $load\ h\ c\ t = Success\ (Sint64-v\ (cat-s64\ (retrieve-bytes\ (content\ m)\ (nat\ (offset\ c))\ |t|_\tau)))$
unfolding *load-def retrieve-tval-def*
using *assms*
by *fastforce*

lemma *load-not-cap-in-mem*:

assumes $tag\ c = True$
and $perm-load\ c = True$
and $offset\ c + |t|_\tau \leq base\ c + len\ c$
and $offset\ c \geq base\ c$
and $offset\ c \bmod |t|_\tau = 0$
and $Mapping.lookup\ (heap-map\ h)\ (block-id\ c) = Some\ (Map\ m)$
and $offset\ c \geq fst\ (bounds\ m)$
and $offset\ c + |t|_\tau \leq snd\ (bounds\ m)$
and $\neg is-contiguous-bytes\ (content\ m)\ (nat\ (offset\ c))\ |t|_\tau$
and $\neg is-cap\ (content\ m)\ (nat\ (offset\ c))$
shows $load\ h\ c\ t = Success\ Undef$
unfolding *load-def retrieve-tval-def*
using *assms*
by *fastforce*

lemma *load-not-contiguous-cap-in-mem*:

assumes $tag\ c = True$
and $perm-load\ c = True$
and $offset\ c + |t|_\tau \leq base\ c + len\ c$
and $offset\ c \geq base\ c$
and $offset\ c \bmod |t|_\tau = 0$
and $Mapping.lookup\ (heap-map\ h)\ (block-id\ c) = Some\ (Map\ m)$
and $offset\ c \geq fst\ (bounds\ m)$
and $offset\ c + |t|_\tau \leq snd\ (bounds\ m)$
and $\neg is-contiguous-bytes\ (content\ m)\ (nat\ (offset\ c))\ |t|_\tau$
and $is-cap\ (content\ m)\ (nat\ (offset\ c))$
and $mc = get-cap\ (content\ m)\ (nat\ (offset\ c))$
and $\neg is-contiguous-cap\ (content\ m)\ mc\ (nat\ (offset\ c))\ |t|_\tau$
and $t \neq Uint8$
and $t \neq Sint8$

shows $load\ h\ c\ t = Success\ Undef$
unfolding $load-def\ retrieve-tval-def\ Let-def$
using $assms$
by $(clarsimp\ split: cctype.split)$

lemma $load-cap-frag-u8$:

assumes $tag\ c = True$
and $perm-load\ c = True$
and $offset\ c + |t|_\tau \leq base\ c + len\ c$
and $offset\ c \geq base\ c$
and $offset\ c\ mod\ |t|_\tau = 0$
and $Mapping.lookup\ (heap-map\ h)\ (block-id\ c) = Some\ (Map\ m)$
and $offset\ c \geq fst\ (bounds\ m)$
and $offset\ c + |t|_\tau \leq snd\ (bounds\ m)$
and $\neg\ is-contiguous-bytes\ (content\ m)\ (nat\ (offset\ c))\ |t|_\tau$
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 $tg = (case\ perm-cap-load\ c\ of\ False \Rightarrow False\ |\ True \Rightarrow tagval)$
and $nth-frag = of-nth\ (the\ (Mapping.lookup\ (content\ m)\ (nat\ (offset\ c))))$
shows $load\ h\ c\ t = Success\ (Cap-v-frag\ (mem-capability.extend\ mc\ (!\ tag = False$
 $!))\ nth-frag)$
unfolding $load-def\ retrieve-tval-def\ Let-def$
using $assms$
by $(clarsimp\ simp\ add: sizeof-def\ split: cctype.split)$

lemma $load-cap-frag-s8$:

assumes $tag\ c = True$
and $perm-load\ c = True$
and $offset\ c + |t|_\tau \leq base\ c + len\ c$
and $offset\ c \geq base\ c$
and $offset\ c\ mod\ |t|_\tau = 0$
and $Mapping.lookup\ (heap-map\ h)\ (block-id\ c) = Some\ (Map\ m)$
and $offset\ c \geq fst\ (bounds\ m)$
and $offset\ c + |t|_\tau \leq snd\ (bounds\ m)$
and $\neg\ is-contiguous-bytes\ (content\ m)\ (nat\ (offset\ c))\ |t|_\tau$
and $is-cap\ (content\ m)\ (nat\ (offset\ c))$
and $mc = get-cap\ (content\ m)\ (nat\ (offset\ c))$
and $\neg\ is-contiguous-cap\ (content\ m)\ mc\ (nat\ (offset\ c))\ |t|_\tau$
and $t = Sint8$
and $tagval = the\ (Mapping.lookup\ (tags\ m)\ (cap-offset\ (nat\ (offset\ c))))$
and $tg = (case\ perm-cap-load\ c\ of\ False \Rightarrow False\ |\ True \Rightarrow tagval)$
and $nth-frag = of-nth\ (the\ (Mapping.lookup\ (content\ m)\ (nat\ (offset\ c))))$
shows $load\ h\ c\ t = Success\ (Cap-v-frag\ (mem-capability.extend\ mc\ (!\ tag = False$
 $!))\ nth-frag)$
unfolding $load-def\ retrieve-tval-def\ Let-def$
using $assms$

by (clarsimp simp add: sizeof-def split: ctype.split)

lemma load-bytes-on-capbytes-fail:

assumes tag c = True
and perm-load c = True
and offset c + |t|_τ ≤ base c + len c
and offset c ≥ base c
and offset c mod |t|_τ = 0
and Mapping.lookup (heap-map h) (block-id c) = Some (Map m)
and offset c ≥ fst (bounds m)
and offset c + |t|_τ ≤ snd (bounds m)
and ¬ is-contiguous-bytes (content m) (nat (offset c)) |t|_τ
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|_τ
and t ≠ Cap
and t ≠ Uint8
and t ≠ Sint8
shows load h c t = Success Undef
unfolding load-def retrieve-tval-def Let-def
using assms
by (clarsimp split: ctype.split)

lemma load-cap-on-capbytes:

assumes tag c = True
and perm-load c = True
and offset c + |t|_τ ≤ base c + len c
and offset c ≥ base c
and offset c mod |t|_τ = 0
and Mapping.lookup (heap-map h) (block-id c) = Some (Map m)
and offset c ≥ fst (bounds m)
and offset c + |t|_τ ≤ snd (bounds m)
and ¬ is-contiguous-bytes (content m) (nat (offset c)) |t|_τ
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|_τ
and t = Cap
and tagval = the (Mapping.lookup (tags m) (nat (offset c)))
and tg = (case perm-cap-load c of False ⇒ False | True ⇒ tagval)
shows load h c t = Success (Cap-v (mem-capability.extend mc (| tag = tg)))
unfolding load-def retrieve-tval-def
using assms
by (clarsimp split: ctype.split)
(smt (verit) assms(5) nat-int nat-less-le nat-mod-distrib of-nat-0-le-iff semiring-1-class.of-nat-0)

lemma load-cond-hard-cap:

assumes load h c t = Success ret
shows tag c = True

```

    and perm-load c = True
    and offset c + |t|τ ≤ base c + len c
    and offset c ≥ base c
    and offset c mod |t|τ = 0
  proof -
    show tag c = True
      using assms result.distinct(1)
      unfolding load-def
      by metis
  next
    show perm-load c = True
      using assms result.distinct(1)
      unfolding load-def
      by metis
  next
    show offset c + |t|τ ≤ base c + len c
      using assms result.distinct(1) linorder-not-le
      unfolding load-def
      by metis
  next
    show offset c ≥ base c
      using assms result.distinct(1) linorder-not-le
      unfolding load-def
      by metis
  next
    show offset c mod |t|τ = 0
      using assms result.distinct(1)
      unfolding load-def
      by metis
qed

```

lemma *load-cond-bytes*:

```

  assumes load h c t = Success ret
  and ret ≠ Undef
  and ∀ x. ret ≠ Cap-v x
  and ∀ x n . ret ≠ Cap-v-frag x n
  shows ∃ m. Mapping.lookup (heap-map h) (block-id c) = Some (Map m)
    ∧ offset c ≥ fst (bounds m)
    ∧ offset c + |t|τ ≤ snd (bounds m)
    ∧ is-contiguous-bytes (content m) (nat (offset c)) |t|τ
  proof (cases ret)
  case (Cap-v x9)
  thus ?thesis
    using assms(3)
    by blast
  next
  case (Cap-v-frag x101 x102)
  thus ?thesis
    using assms(4)

```

```

    by blast
next
case Undef
thus ?thesis
  using assms(2)
  by simp
qed (insert assms(1) load-cond-hard-cap[where ?h=h and ?c=c and ?t=t and
?ret=ret], clarsimp, unfold load-def retrieve-tval-def, clarsimp split: option.split-asm
t.split-asm, smt (z3) assms(2) assms(3) assms(4) cctype.exhaust cctype.simps(73)
cctype.simps(74) cctype.simps(75) cctype.simps(76) cctype.simps(77) cctype.simps(78)
cctype.simps(79) cctype.simps(80) cctype.simps(81) result.distinct(1) result.inject(1))+

```

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-map h) (block-id c) = Some (Map m)  $\wedge$ 
  offset c  $\geq$  fst (bounds m)  $\wedge$ 
  offset c + |t| $_{\tau}$   $\leq$  snd (bounds m)  $\wedge$ 
  (is-contiguous-bytes (content m) (nat (offset c)) |t| $_{\tau}$   $\longrightarrow$ 
  is-contiguous-zeros (content m) (nat (offset c)) |t| $_{\tau}$   $\wedge$ 
  ret = Cap-v NULL)  $\wedge$ 
  ( $\neg$  is-contiguous-bytes (content m) (nat (offset c)) |t| $_{\tau}$   $\longrightarrow$ 
  is-cap (content m) (nat (offset c))  $\wedge$ 
  mc = get-cap (content m) (nat (offset c))  $\wedge$ 
  is-contiguous-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-cap-load c of False  $\Rightarrow$  False | True  $\Rightarrow$  tagval))
  using assms(2)
proof (cases ret)
case (Cap-v ca)
show ?thesis
  using assms load-cond-hard-cap[where ?h=h and ?c=c and ?t=t and ?ret=ret]
  by (clarsimp, simp only: load-def retrieve-tval-def Let-def, clarsimp split: option.split-asm,
  clarsimp split: t.split-asm, rename-tac x nth map, subgoal-tac int
  (fst (bounds map))  $\leq$  int |t| $_{\tau}$  * nth  $\wedge$  int |t| $_{\tau}$  * nth + int |t| $_{\tau}$   $\leq$  int (snd (bounds
  map)), clarsimp split: cctype.split-asm, safe; force?)
  (metis ccval.distinct(105) ccval.distinct(107) ccval.inject(9) is-cap.elims(2)
  linorder-not-le result.distinct(1))+
qed blast+

```

lemma *load-cond-cap-frag*:

```

assumes load h c t = Success ret
and  $\exists x\ n. ret = Cap\text{-}v\text{-}frag\ x\ n$ 
shows  $\exists m\ mc\ tagval\ tg\ nth\ \text{-}frag.$ 
  Mapping.lookup (heap-map h) (block-id c) = Some (Map m)  $\wedge$ 
  offset c  $\geq$  fst (bounds m)  $\wedge$ 

```

```

offset c + |t|τ ≤ snd (bounds m) ∧
(is-contiguous-bytes (content m) (nat (offset c)) |t|τ →
 is-contiguous-zeros (content m) (nat (offset c)) |t|τ ∧
 ret = Cap-v NULL) ∧
(¬ is-contiguous-bytes (content m) (nat (offset c)) |t|τ →
 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)))))
using assms(2)
proof (cases ret)
  case (Cap-v-frag x101 x102)
  show ?thesis
    using assms load-cond-hard-cap[where ?h=h and ?c=c and ?t=t and ?ret=ret]
    by (clarsimp, simp only: load-def retrieve-tval-def Let-def, clarsimp split: option.split-asm, clarsimp split: t.split-asm if-split-asm cctype.split-asm)
qed (simp add: type-uniq assms(2))+

```

```

lemma store-null-error:
  store h NULL v = Error (C2Err TagViolation)
  unfolding store-def
  by simp

```

```

lemma store-false-tag:
  assumes tag c = False
  shows store h c v = Error (C2Err TagViolation)
  unfolding store-def
  using assms
  by presburger

```

```

lemma store-false-perm-store:
  assumes tag c = True
  and perm-store c = False
  shows store h c v = Error (C2Err PermitStoreViolation)
  unfolding store-def
  using assms
  by presburger

```

```

lemma store-cap-false-perm-cap-store:
  assumes tag c = True
  and perm-store c = True
  and perm-cap-store c = False
  and ∃ cv. v = Cap-v cv ∧ tag cv = True
  shows store h c v = Error (C2Err PermitStoreCapViolation)
  unfolding store-def
  using assms

```

by force

lemma *store-cap-false-perm-cap-store-local*:

assumes $\text{tag } c = \text{True}$
and $\text{perm-store } c = \text{True}$
and $\text{perm-cap-store } c = \text{True}$
and $\text{perm-cap-store-local } c = \text{False}$
and $\exists cv. v = \text{Cap-}v \text{ } cv \wedge \text{tag } cv = \text{True} \wedge \text{perm-global } cv = \text{False}$
shows $\text{store } h \ c \ v = \text{Error } (\text{C2Err PermitStoreLocalCapViolation})$
unfolding *store-def*
using *assms*
by force

lemma *store-bound-over*:

assumes $\text{tag } c = \text{True}$
and $\text{perm-store } c = \text{True}$
and $\bigwedge cv. \llbracket v = \text{Cap-}v \text{ } cv; \text{tag } cv \rrbracket \implies \text{perm-cap-store } c \wedge (\text{perm-cap-store-local } c \vee \text{perm-global } cv)$
and $\text{offset } c + |\text{memval-type } v|_{\tau} > \text{base } c + \text{len } c$
shows $\text{store } h \ c \ v = \text{Error } (\text{C2Err LengthViolation})$
unfolding *store-def*
using *assms*
by (*clarsimp split: ccval.split*)

lemma *store-bound-under*:

assumes $\text{tag } c = \text{True}$
and $\text{perm-store } c = \text{True}$
and $\bigwedge cv. \llbracket v = \text{Cap-}v \text{ } cv; \text{tag } cv \rrbracket \implies \text{perm-cap-store } c \wedge (\text{perm-cap-store-local } c \vee \text{perm-global } cv)$
and $\text{offset } c + |\text{memval-type } v|_{\tau} \leq \text{base } c + \text{len } c$
and $\text{offset } c < \text{base } c$
shows $\text{store } h \ c \ v = \text{Error } (\text{C2Err LengthViolation})$
unfolding *store-def*
using *assms*
by (*clarsimp split: ccval.split*)

lemma *store-misaligned*:

assumes $\text{tag } c = \text{True}$
and $\text{perm-store } c = \text{True}$
and $\bigwedge cv. \llbracket v = \text{Cap-}v \text{ } cv; \text{tag } cv \rrbracket \implies \text{perm-cap-store } c \wedge (\text{perm-cap-store-local } c \vee \text{perm-global } cv)$
and $\text{offset } c + |\text{memval-type } v|_{\tau} \leq \text{base } c + \text{len } c$
and $\text{offset } c \geq \text{base } c$
and $\text{offset } c \bmod |\text{memval-type } v|_{\tau} \neq 0$
shows $\text{store } h \ c \ v = \text{Error } (\text{C2Err BadAddressViolation})$
unfolding *store-def*
using *assms*
by (*clarsimp split: ccval.split*)

lemma *store-undef-val*:
assumes $tag\ c = True$
and $perm-store\ c = True$
and $\bigwedge cv. \llbracket v = Cap-v\ cv; tag\ cv \rrbracket \implies perm-cap-store\ c \wedge (perm-cap-store-local\ c \vee perm-global\ cv)$
and $offset\ c + |memval-type\ v|_{\tau} \leq base\ c + len\ c$
and $offset\ c \geq base\ c$
and $offset\ c\ mod\ |memval-type\ v|_{\tau} = 0$
and $v = Undef$
shows $store\ h\ c\ v = Error\ (LogicErr\ (Unhandled\ 0))$
unfolding *store-def*
using *assms*
by *auto*

lemma *store-nonexistent-obj*:
assumes $tag\ c = True$
and $perm-store\ c = True$
and $\bigwedge cv. \llbracket v = Cap-v\ cv; tag\ cv \rrbracket \implies perm-cap-store\ c \wedge (perm-cap-store-local\ c \vee perm-global\ cv)$
and $offset\ c + |memval-type\ v|_{\tau} \leq base\ c + len\ c$
and $offset\ c \geq base\ c$
and $offset\ c\ mod\ |memval-type\ v|_{\tau} = 0$
and $v \neq Undef$
and $Mapping.lookup\ (heap-map\ h)\ (block-id\ c) = None$
shows $store\ h\ c\ v = Error\ (LogicErr\ MissingResource)$
unfolding *store-def*
using *assms*
by $(clarsimp\ split: ccval.split)$

lemma *store-store-after-free*:
assumes $tag\ c = True$
and $perm-store\ c = True$
and $\bigwedge cv. \llbracket v = Cap-v\ cv; tag\ cv \rrbracket \implies perm-cap-store\ c \wedge (perm-cap-store-local\ c \vee perm-global\ cv)$
and $offset\ c + |memval-type\ v|_{\tau} \leq base\ c + len\ c$
and $offset\ c \geq base\ c$
and $offset\ c\ mod\ |memval-type\ v|_{\tau} = 0$
and $v \neq Undef$
and $Mapping.lookup\ (heap-map\ h)\ (block-id\ c) = Some\ Freed$
shows $store\ h\ c\ v = Error\ (LogicErr\ UseAfterFree)$
unfolding *store-def*
using *assms*
by $(clarsimp\ split: ccval.split)$

lemma *store-bound-violated-1*:
assumes $tag\ c = True$
and $perm-store\ c = True$
and $\bigwedge cv. \llbracket v = Cap-v\ cv; tag\ cv \rrbracket \implies perm-cap-store\ c \wedge (perm-cap-store-local\ c \vee perm-global\ cv)$

and $\text{offset } c + |\text{memval-type } v|_\tau \leq \text{base } c + \text{len } c$
and $\text{offset } c \geq \text{base } c$
and $\text{offset } c \bmod |\text{memval-type } v|_\tau = 0$
and $v \neq \text{Undef}$
and $\text{Mapping.lookup } (\text{heap-map } h) (\text{block-id } c) = \text{Some } (\text{Map } m)$
and $\text{offset } c < \text{fst } (\text{bounds } m)$
shows $\text{store } h \ c \ v = \text{Error } (\text{LogicErr BufferOverrun})$
unfolding *store-def* **using** *assms*
by (*clarsimp split: ccval.split*)

lemma *store-bound-violated-2:*

assumes $\text{tag } c = \text{True}$
and $\text{perm-store } c = \text{True}$
and $\bigwedge cv. \llbracket v = \text{Cap-}v \ cv; \text{tag } cv \rrbracket \implies \text{perm-cap-store } c \wedge (\text{perm-cap-store-local } c \vee \text{perm-global } cv)$
and $\text{offset } c + |\text{memval-type } v|_\tau \leq \text{base } c + \text{len } c$
and $\text{offset } c \geq \text{base } c$
and $\text{offset } c \bmod |\text{memval-type } v|_\tau = 0$
and $v \neq \text{Undef}$
and $\text{Mapping.lookup } (\text{heap-map } h) (\text{block-id } c) = \text{Some } (\text{Map } m)$
and $\text{offset } c + |\text{memval-type } v|_\tau > \text{snd } (\text{bounds } m)$
shows $\text{store } h \ c \ v = \text{Error } (\text{LogicErr BufferOverrun})$
unfolding *store-def* **using** *assms*
by (*clarsimp split: ccval.split*)

lemma *store-success:*

assumes $\text{tag } c = \text{True}$
and $\text{perm-store } c = \text{True}$
and $\bigwedge cv. \llbracket v = \text{Cap-}v \ cv; \text{tag } cv \rrbracket \implies \text{perm-cap-store } c \wedge (\text{perm-cap-store-local } c \vee \text{perm-global } cv)$
and $\text{offset } c + |\text{memval-type } v|_\tau \leq \text{base } c + \text{len } c$
and $\text{offset } c \geq \text{base } c$
and $\text{offset } c \bmod |\text{memval-type } v|_\tau = 0$
and $v \neq \text{Undef}$
and $\text{Mapping.lookup } (\text{heap-map } h) (\text{block-id } c) = \text{Some } (\text{Map } m)$
and $\text{offset } c \geq \text{fst } (\text{bounds } m)$
and $\text{offset } c + |\text{memval-type } v|_\tau \leq \text{snd } (\text{bounds } m)$
shows $\exists \text{ret. store } h \ c \ v = \text{Success } \text{ret} \wedge$
 $\text{next-block } \text{ret} = \text{next-block } h \wedge$
 $\text{heap-map } \text{ret} = \text{Mapping.update } (\text{block-id } c) (\text{Map } (\text{store-tval } m \ (\text{nat } (\text{offset } c)) \ v)) (\text{heap-map } h)$
unfolding *store-def*
using *assms*
by (*clarsimp split: ccval.split*)

lemma *store-cond-hard-cap:*

assumes $\text{store } h \ c \ v = \text{Success } \text{ret}$
shows $\text{tag } c = \text{True}$
and $\text{perm-store } c = \text{True}$

```

and  $\bigwedge cv. \llbracket v = \text{Cap-}v\ cv; \text{tag } cv \rrbracket \implies \text{perm-cap-store } c \wedge (\text{perm-cap-store-local } c \vee \text{perm-global } cv)$ 
and  $\text{offset } c + |\text{memval-type } v|_\tau \leq \text{base } c + \text{len } c$ 
and  $\text{offset } c \geq \text{base } c$ 
and  $\text{offset } c \bmod |\text{memval-type } v|_\tau = 0$ 
proof –
  show  $\text{tag } c = \text{True}$ 
    using assms unfolding store-def
    by (meson result.simps(4))
  next
    show  $\text{perm-store } c = \text{True}$ 
      using assms unfolding store-def
      by (meson result.simps(4))
  next
    show  $\bigwedge cv. \llbracket v = \text{Cap-}v\ cv; \text{tag } cv \rrbracket \implies \text{perm-cap-store } c \wedge (\text{perm-cap-store-local } c \vee \text{perm-global } cv)$ 
      using assms unfolding store-def
      by (metis (no-types, lifting) assms result.simps(4) store-cap-false-perm-cap-store store-cap-false-perm-cap-store-local)
  next
    show  $\text{offset } c + |\text{memval-type } v|_\tau \leq \text{base } c + \text{len } c$ 
      using assms unfolding store-def
      by (meson linorder-not-le result.simps(4))
  next
    show  $\text{offset } c \geq \text{base } c$ 
      using assms unfolding store-def
      by (meson linorder-not-le result.simps(4))
  next
    show  $\text{offset } c \bmod |\text{memval-type } v|_\tau = 0$ 
      using assms unfolding store-def
      by (meson linorder-not-le result.simps(4))
qed

```

lemma *store-cond-bytes-bounds*:

```

assumes  $\text{store } h\ c\ \text{val} = \text{Success } h'$ 
and  $\forall x. \text{val} \neq \text{Cap-}v\ x$ 
shows  $\exists m. \text{Mapping.lookup } (\text{heap-map } h) (\text{block-id } c) = \text{Some } (\text{Map } m)$ 
   $\wedge \text{offset } c \geq \text{fst } (\text{bounds } m)$ 
   $\wedge \text{offset } c + |\text{memval-type } \text{val}|_\tau \leq \text{snd } (\text{bounds } m)$ 
using store-cond-hard-cap[where ?h=h and ?c=c and ?v=val and ?ret=h', OF assms(1)] assms
unfolding store-def
by (simp split: ccval.split-asm; simp split: option.split-asm t.split-asm)
  (metis nle-le order.strict-iff-not result.distinct(1))+

```

lemma *store-cond-bytes*:

```

assumes  $\text{store } h\ c\ \text{val} = \text{Success } h'$ 
and  $\forall x. \text{val} \neq \text{Cap-}v\ x$ 
shows  $\exists m. \text{Mapping.lookup } (\text{heap-map } h') (\text{block-id } c) = \text{Some } (\text{Map } m)$ 

```


$\wedge \text{offset } c \geq \text{fst } (\text{bounds } m)$
 $\wedge \text{offset } c + |\text{memval-type } \text{val}|_{\tau} \leq \text{snd } (\text{bounds } m)$
using *store-cond-hard-cap*[**where** $?h=h$ **and** $?c=c$ **and** $?v=\text{val}$ **and** $?ret=h'$, *OF*
assms(1)] *assms*
unfolding *store-def*
by (*simp split: ccval.split-asm; simp split: option.split-asm t.split-asm*)
(*auto split: if-split-asm simp add: store-tval-def*)

lemma *store-cond-cap-bounds*:

assumes *store* h c $\text{val} = \text{Success } h'$
and $\text{val} = \text{Cap-v } x$
shows $\exists m. \text{Mapping.lookup } (\text{heap-map } h) (\text{block-id } c) = \text{Some } (\text{Map } m)$
 $\wedge \text{offset } c \geq \text{fst } (\text{bounds } m)$
 $\wedge \text{offset } c + |\text{memval-type } \text{val}|_{\tau} \leq \text{snd } (\text{bounds } m)$

proof –

have *cap-perms*: $\neg(\neg \text{perm-cap-store } c \wedge \text{tag } x) \wedge \neg(\neg \text{perm-cap-store-local } c \wedge$
 $\text{tag } x \wedge \neg \text{perm-global } x)$
using *store-cond-hard-cap(1)*[**where** $?h=h$ **and** $?c=c$ **and** $?v=\text{val}$ **and** $?ret=h'$,
OF *assms(1)*]
store-cond-hard-cap(2)[**where** $?h=h$ **and** $?c=c$ **and** $?v=\text{val}$ **and** $?ret=h'$, *OF*
assms(1)]
store-cond-hard-cap(3)[**where** $?h=h$ **and** $?c=c$ **and** $?v=\text{val}$ **and** $?ret=h'$ **and**
 $?cv=x$, *OF* *assms(1)*]
store-cond-hard-cap(4)[**where** $?h=h$ **and** $?c=c$ **and** $?v=\text{val}$ **and** $?ret=h'$, *OF*
assms(1)]
store-cond-hard-cap(5)[**where** $?h=h$ **and** $?c=c$ **and** $?v=\text{val}$ **and** $?ret=h'$, *OF*
assms(1)]
store-cond-hard-cap(6)[**where** $?h=h$ **and** $?c=c$ **and** $?v=\text{val}$ **and** $?ret=h'$, *OF*
assms(1)]
assms
unfolding *store-def*
by *blast*
thus *?thesis*
using *store-cond-hard-cap(1)*[**where** $?h=h$ **and** $?c=c$ **and** $?v=\text{val}$ **and** $?ret=h'$,
OF *assms(1)*]
store-cond-hard-cap(2)[**where** $?h=h$ **and** $?c=c$ **and** $?v=\text{val}$ **and** $?ret=h'$, *OF*
assms(1)]
store-cond-hard-cap(3)[**where** $?h=h$ **and** $?c=c$ **and** $?v=\text{val}$ **and** $?ret=h'$ **and**
 $?cv=x$, *OF* *assms(1)*]
store-cond-hard-cap(4)[**where** $?h=h$ **and** $?c=c$ **and** $?v=\text{val}$ **and** $?ret=h'$, *OF*
assms(1)]
store-cond-hard-cap(5)[**where** $?h=h$ **and** $?c=c$ **and** $?v=\text{val}$ **and** $?ret=h'$, *OF*
assms(1)]
store-cond-hard-cap(6)[**where** $?h=h$ **and** $?c=c$ **and** $?v=\text{val}$ **and** $?ret=h'$, *OF*
assms(1)]
assms
by (*simp split: ccval.split option.split-asm t.split-asm add: store-def cap-perms*)
(*metis linorder-not-le result.distinct(1)*)

qed

lemma *store-cond-cap*:

assumes *store h c val = Success h'*

and *val = Cap-v v*

shows $\exists m. \text{Mapping.lookup (heap-map h')} (\text{block-id } c) = \text{Some (Map } m)$
 $\wedge \text{offset } c \geq \text{fst (bounds } m)$
 $\wedge \text{offset } c + |\text{memval-type val}|_\tau \leq \text{snd (bounds } m)$

proof –

have *cap-perms*: $\neg(\neg \text{perm-cap-store } c \wedge \text{tag } v) \wedge \neg(\neg \text{perm-cap-store-local } c \wedge \text{tag } v \wedge \neg \text{perm-global } v)$

using *store-cond-hard-cap(1)* [**where** *?h=h* **and** *?c=c* **and** *?v=val* **and** *?ret=h'*, *OF assms(1)*]

store-cond-hard-cap(2) [**where** *?h=h* **and** *?c=c* **and** *?v=val* **and** *?ret=h'*, *OF assms(1)*]

store-cond-hard-cap(3) [**where** *?h=h* **and** *?c=c* **and** *?v=val* **and** *?ret=h'* **and** *?cv=v*, *OF assms(1)*]

store-cond-hard-cap(4) [**where** *?h=h* **and** *?c=c* **and** *?v=val* **and** *?ret=h'*, *OF assms(1)*]

store-cond-hard-cap(5) [**where** *?h=h* **and** *?c=c* **and** *?v=val* **and** *?ret=h'*, *OF assms(1)*]

store-cond-hard-cap(6) [**where** *?h=h* **and** *?c=c* **and** *?v=val* **and** *?ret=h'*, *OF assms(1)*]

assms

by *blast*

show *?thesis*

using *store-cond-hard-cap(1)* [**where** *?h=h* **and** *?c=c* **and** *?v=val* **and** *?ret=h'*, *OF assms(1)*]

store-cond-hard-cap(2) [**where** *?h=h* **and** *?c=c* **and** *?v=val* **and** *?ret=h'*, *OF assms(1)*]

store-cond-hard-cap(3) [**where** *?h=h* **and** *?c=c* **and** *?v=val* **and** *?ret=h'* **and** *?cv=v*, *OF assms(1)*]

store-cond-hard-cap(4) [**where** *?h=h* **and** *?c=c* **and** *?v=val* **and** *?ret=h'*, *OF assms(1)*]

store-cond-hard-cap(5) [**where** *?h=h* **and** *?c=c* **and** *?v=val* **and** *?ret=h'*, *OF assms(1)*]

store-cond-hard-cap(6) [**where** *?h=h* **and** *?c=c* **and** *?v=val* **and** *?ret=h'*, *OF assms(1)*]

assms

by (*clarsimp split: ccval.split option.split-asm t.split-asm simp add: store-def cap-perms*)

(*smt (verit) Mapping.lookup-update assms(1) result.distinct(1) result.sel(1)*)

store-bound-violated-2 store-cond-hard-cap(3) store-def store-success store-tval-disjoint-bounds)

qed

lemma *store-cond*:

assumes *store h c val = Success h'*

shows $\exists m. \text{Mapping.lookup (heap-map h')} (\text{block-id } c) = \text{Some (Map } m)$

$\wedge \text{offset } c \geq \text{fst (bounds } m)$

$\wedge \text{offset } c + |\text{memval-type val}|_\tau \leq \text{snd (bounds } m)$

```

using store-cond-bytes[OF assms(1)] store-cond-cap[OF assms(1)]
by blast

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'
  using assms store-cond-hard-cap[OF assms(1)]
proof (cases v)
  case (Uint8-v x1)
  thus ?thesis
    using assms store-cond-hard-cap[OF assms(1)]
    unfolding store-def
    by (clarsimp split: if-split-asm) (blast intro: store-tval-disjoint-bounds)
next
  case (Sint8-v x2)
  thus ?thesis
    using assms store-cond-hard-cap[OF assms(1)]
    unfolding store-def
    by (clarsimp split: if-split-asm) (blast intro: store-tval-disjoint-bounds)
next
  case (Uint16-v x3)
  thus ?thesis
    using assms store-cond-hard-cap[OF assms(1)]
    unfolding store-def
    by (clarsimp split: if-split-asm) (blast intro: store-tval-disjoint-bounds)
next
  case (Sint16-v x4)
  thus ?thesis
    using assms store-cond-hard-cap[OF assms(1)]
    unfolding store-def
    by (clarsimp split: if-split-asm) (blast intro: store-tval-disjoint-bounds)
next
  case (Uint32-v x5)
  thus ?thesis
    using assms store-cond-hard-cap[OF assms(1)]
    unfolding store-def
    by (clarsimp split: if-split-asm) (blast intro: store-tval-disjoint-bounds)
next
  case (Sint32-v x6)
  thus ?thesis
    using assms store-cond-hard-cap[OF assms(1)]
    unfolding store-def
    by (clarsimp split: if-split-asm) (blast intro: store-tval-disjoint-bounds)
next
  case (Uint64-v x7)
  thus ?thesis
    using assms store-cond-hard-cap[OF assms(1)]

```

```

    unfolding store-def
    by (clarsimp split: if-split-asm) (blast intro: store-tval-disjoint-bounds)
next
case (Sint64-v x8)
  thus ?thesis
  using assms store-cond-hard-cap[OF assms(1)]
  unfolding store-def
  by (clarsimp split: if-split-asm) (blast intro: store-tval-disjoint-bounds)
next
case (Cap-v x9)
  moreover hence  $\neg(\neg \text{perm-cap-store } c \wedge \text{tag } x9) \wedge \neg(\neg \text{perm-cap-store-local } c$ 
 $\wedge \text{tag } x9 \wedge \neg \text{perm-global } x9)$ 
  using assms store-cond-hard-cap[OF assms(1)]
  unfolding store-def
  by blast
  ultimately show ?thesis
  using assms store-cond-hard-cap[OF assms(1)]
  unfolding store-def
  by (simp split: if-split-asm add: store-tval-def; auto)
next
case (Cap-v-frag x101 x102)
  thus ?thesis
  using assms store-cond-hard-cap[OF assms(1)]
  unfolding store-def
  by (clarsimp split: if-split-asm) (blast intro: store-tval-disjoint-bounds)
next
case Undef
  thus ?thesis
  using assms store-cond-hard-cap[OF assms(1)]
  unfolding store-def
  by force
qed

```

lemma *store-cond-cap-frag*:

```

  assumes store h c val = Success h'
  and val = Cap-v-frag v n
  shows  $\exists m. \text{Mapping.lookup } (\text{heap-map } h') (\text{block-id } c) = \text{Some } (\text{Map } m)$ 
  using store-cond-hard-cap[where ?h=h and ?c=c and ?v=val and ?ret=h', OF
  assms(1)] assms
  unfolding store-def
  by (simp split: ccval.split-asm; simp split: option.split-asm t.split-asm)
  (metis Mapping.lookup-update heap.select-convs(2) heap.surjective heap.update-convs(2)
  result.distinct(1) result.sel(1))

```

lemma *store-undef-false*:

```

  assumes store h c Undef = Success ret
  shows False
  using store-cond-hard-cap[where ?h=h and ?c=c and ?v=Undef and ?ret=ret,
  OF assms] assms

```

```

unfolding store-def
by simp

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 –
  have tag cap = True
    using assms alloc-def
    by auto
  moreover have perm-load cap = True
    using assms alloc-def
    by force
  moreover have base cap = 0
    using assms alloc-def
    by fastforce
  moreover have len cap = s
    using assms alloc-def
    by auto
  ultimately show ?thesis
    using assms load-def by auto
qed

```

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 falsified.

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 –

let $?m = (\text{bounds} = (0, s), \text{content} = \text{Mapping.empty}, \text{tags} = \text{Mapping.empty})$

have *tag cap = True*

using *assms(1) alloc-def*

by *fastforce*

moreover have *perm-load cap = True*

using *assms(1) alloc-def*

by *fastforce*

moreover have $\text{offset cap} + |t|_\tau \leq \text{base cap} + \text{len cap}$

using *assms alloc-def*

by *fastforce*

moreover have $\text{offset cap} \geq \text{base cap}$

using *assms alloc-def*

by *fastforce*

moreover have $\text{offset cap} \bmod |t|_\tau = 0$

using *assms alloc-def*

by *fastforce*

moreover have $\text{Mapping.lookup} (\text{heap-map } h') (\text{block-id cap}) = \text{Some} (\text{Map } ?m)$

using *assms alloc-def*

by *fastforce*

moreover have $\text{offset cap} \geq \text{fst} (\text{bounds } ?m)$

using *assms alloc-def*

by *fastforce*

moreover have $\text{offset cap} + |t|_\tau \leq \text{snd} (\text{bounds } ?m)$

using *assms alloc-def*

by *fastforce*

moreover have $\neg \text{is-contiguous-bytes} (\text{content } ?m) (\text{nat} (\text{offset cap})) |t|_\tau$

proof –

have $\exists n. |t|_\tau = \text{Suc } n$

using *not0-implies-Suc sizeof-nonzero*

by *force*
 thus *?thesis*
 using *assms alloc-def*
 by *fastforce*
 qed
 moreover have $\neg \text{is-cap } (\text{content } ?m) (\text{nat } (\text{offset } \text{cap}))$
 by *simp*
 ultimately show *?thesis*
 using *load-not-cap-in-mem*
 by *presburger*
 qed

theorem *load-after-alloc-2*:
 assumes *alloc h c s = Success (h', cap)*
 and $|t|_{\tau} \leq s$
 and *block-id cap \neq block-id cap'*
 shows *load h' cap' t = load h cap' t*
 using *assms unfolding alloc-def load-def*
 by *force*

theorem *load-after-free-1*:
 assumes *free h c = Success (h', cap)*
 shows *load h cap t = Error (C2Err TagViolation)*

proof –
 consider $(\text{null}) \ c = \text{NULL} \mid (\text{non-null}) \ c \neq \text{NULL}$ by *blast*
 then show *?thesis*
proof (*cases*)
 case *null*
 moreover hence $c = \text{cap}$
 using *assms free-null*
 by *force*
 ultimately show *?thesis*
 using *load-null-error assms*
 by *blast*
 next
 case *non-null*
 hence $\text{cap} = c \ (\ \text{tag} := \text{False} \)$
 using *assms free-cond(6)*[**where** *?h=h and ?c=c and ?h'=h' and ?cap=cap*]

 by *presburger*
 moreover hence $\text{tag } \text{cap} = \text{False}$
 using *assms*
 by *force*
 ultimately show *?thesis* using *load-false-tag*
 by *blast*
 qed
 qed

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
using assms free-cond[OF assms(1)]
unfolding free-def load-def
by fastforce

```

```

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
  using assms store-cond-hard-cap[OF assms(1)]
  unfolding store-def load-def
  by (clarsimp split: ccval.split-asm option.split-asm t.split-asm if-split-asm)

```

```

theorem load-after-store-disjoint-2:
  assumes store h c v = Success h'
    and offset c' + |t|τ ≤ offset c
  shows load h' c' t = load h c' t
  using assms store-cond[OF assms(1)] store-cond-hard-cap[OF assms(1)]
  apply (clarsimp simp add: store-def load-def split: if-split-asm option.split t.split
option.split-asm t.split-asm)
  apply (intro conjI impI)
  apply (smt (z3) lookup-update' option.discI)
  apply (rule allI, rule conjI, rule impI)
  apply (simp add: lookup-update')
  apply (rule allI, rule conjI, rule impI)
  apply (smt (z3) Mapping.lookup-update Mapping.lookup-update-neq option.distinct(1)
option.inject store-tval-disjoint-bounds t.distinct(1) t.inject)
  apply (rule conjI, rule impI)
  apply (smt (z3) Mapping.lookup-update Mapping.lookup-update-neq option.distinct(1)
option.sel store-tval-disjoint-bounds t.distinct(1) t.sel)
  apply (intro impI conjI allI)
  apply (metis (no-types, lifting) Mapping.lookup-update-neq option.distinct(1))
  apply (metis (no-types, lifting) lookup-update' option.inject t.discI)
  apply (smt (z3) Mapping.lookup-update Mapping.lookup-update-neq option.inject
store-tval-disjoint-bounds t.inject)
  apply (metis (no-types, lifting) lookup-update' option.sel store-tval-disjoint-bounds
t.sel)
  using retrieve-stored-tval-disjoint-1
  apply (smt (z3) int-nat-eq lookup-update' of-nat-0-le-iff of-nat-add of-nat-le-iff
option.sel t.sel)
done

```

```

theorem load-after-store-disjoint-3:
  assumes store h c v = Success h'
    and offset c + |memval-type v|τ ≤ offset c'
  shows load h' c' t = load h c' t
  using assms store-cond[OF assms(1)] store-cond-hard-cap[OF assms(1)]

```



```

apply (clarsimp simp add: store-def load-def split: if-split-asm option.split t.split
option.split-asm t.split-asm)
apply (rule conjI)
apply (smt (z3) lookup-update' option.discI)
apply (rule allI, rule conjI)
apply (simp add: lookup-update')
apply (rule allI, rule conjI)
apply (smt (z3) Mapping.lookup-update Mapping.lookup-update-neq option.distinct(1)
option.inject store-tval-disjoint-bounds t.distinct(1) t.inject)
apply (rule conjI)
apply (smt (z3) Mapping.lookup-update Mapping.lookup-update-neq option.distinct(1)
option.sel store-tval-disjoint-bounds t.distinct(1) t.sel)
apply (intro impI conjI allI)
apply (metis (no-types, lifting) Mapping.lookup-update-neq option.distinct(1))
apply (metis (no-types, lifting) lookup-update' option.inject t.discI)
apply (smt (z3) Mapping.lookup-update Mapping.lookup-update-neq option.inject
store-tval-disjoint-bounds t.inject)
apply (metis (no-types, lifting) lookup-update' option.sel store-tval-disjoint-bounds
t.sel)
using retrieve-stored-tval-disjoint-2
apply (smt (z3) int-nat-eq lookup-update' memval-type.simps nat-int nat-mod-distrib
of-nat-add of-nat-le-0-iff of-nat-le-iff option.sel t.sel)
done

```

theorem load-after-store-prim:

```

assumes store h c val = Success h'
and  $\forall v. val \neq \text{Cap-}v\ v$ 
and  $\forall v\ n. val \neq \text{Cap-}v\text{-frag}\ v\ n$ 
and perm-load c = True
shows load h' c (memval-type val) = Success val
using assms(1) store-cond-hard-cap[where ?h=h and ?c=c and ?v=val and
?ret=h', OF assms(1)]
store-cond-bytes[OF assms(1) assms(2)] retrieve-stored-tval-any-perm[OF - -
assms(3)]
by (clarsimp simp add: store-def load-def split: if-split-asm option.split-asm
ccval.split)
(safe; clarsimp simp add: assms)

```

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 |))
using store-cond-hard-cap(1)[where ?h=h and ?c=c and ?v=Cap-v v and ?ret=h',
OF assms(1)]
store-cond-hard-cap(2)[where ?h=h and ?c=c and ?v=Cap-v v and ?ret=h',
OF assms(1)]
store-cond-hard-cap(3)[where ?h=h and ?c=c and ?v=Cap-v v and ?ret=h'
and ?cv=v, OF assms(1) refl]

```

```

    store-cond-hard-cap(4)[where ?h=h and ?c=c and ?v=Cap-v v and ?ret=h',
OF assms(1)]
    store-cond-hard-cap(5)[where ?h=h and ?c=c and ?v=Cap-v v and ?ret=h',
OF assms(1)]
    store-cond-hard-cap(6)[where ?h=h and ?c=c and ?v=Cap-v v and ?ret=h',
OF assms(1)]
  assms
  retrieve-stored-tval-cap[where ?val=Cap-v v and ?v=v, OF refl]
  retrieve-stored-tval-cap-no-perm-cap-load[where ?val=Cap-v v and ?v=v, OF
refl]
  apply (clarsimp, simp split: ccval.split; safe; clarsimp)
  apply (simp only: load-def; clarsimp split: option.split)
  apply (simp add: sizeof-def split: t.split, safe)
  subgoal
  by (blast dest: store-cond-cap)
  subgoal
  by (metis option.sel store-cond-cap t.distinct(1))
  subgoal
  apply (simp add: sizeof-def store-def)
  apply (subgoal-tac  $\neg(\neg \text{perm-cap-store } c \wedge \text{tag } v) \wedge \neg(\neg \text{perm-cap-store-local } c$ 
 $\wedge \text{tag } v \wedge \neg \text{perm-global } v)$ ; presburger?)
  apply (clarsimp split: if-split-asm)
  apply (simp split: option.split-asm t.split-asm if-split-asm)
  apply clarsimp
  apply (smt (verit, best)  $\langle \text{offset } c + \text{int } | \text{memval-type } (Cap-v \ v) |_{\tau} \leq \text{int } (base \ c +$ 
 $len \ c) \rangle \langle \text{offset } c \bmod \text{int } | \text{memval-type } (Cap-v \ v) |_{\tau} = 0 \rangle$  assms(1) ccval.distinct(107)
lookup-update' option.sel result.inject(1) result.simps(4) store-bound-violated-2 store-cond-cap
store-cond-hard-cap(3) store-success t.sel)
  done
  subgoal
  apply (simp add: sizeof-def store-def)
  apply (subgoal-tac  $\neg(\neg \text{perm-cap-store } c \wedge \text{tag } v) \wedge \neg(\neg \text{perm-cap-store-local } c$ 
 $\wedge \text{tag } v \wedge \neg \text{perm-global } v)$ ; presburger?)
  apply (clarsimp split: if-split-asm)
  apply (simp split: option.split-asm t.split-asm if-split-asm)
  apply (clarsimp simp add: sizeof-def)
  apply (smt (verit, ccfv-SIG) Mapping.lookup-update assms(1) heap.select-convs(2)
heap.surjective heap.update-convs(2) store-bounds-preserved)
  done
  subgoal
  apply (simp add: store-def)
  apply (subgoal-tac  $\neg(\neg \text{perm-cap-store } c \wedge \text{tag } v) \wedge \neg(\neg \text{perm-cap-store-local } c$ 
 $\wedge \text{tag } v \wedge \neg \text{perm-global } v)$ ; presburger?)
  apply (clarsimp split: if-split-asm option.split-asm t.split-asm)
  apply (cases perm-cap-load c; clarsimp)
  done
  done

```

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)
using assms(1) store-cond-hard-cap[where ?h=h and ?c=c and ?v=Cap-v-frag
c' n and ?ret=h', OF assms(1)]
      retrieve-stored-tval-cap-frag[where ?val=Cap-v-frag c' n and ?c=c' and
?n=n and ?off=nat (offset c) and ?b=(perm-cap-load c), OF refl, simplified]
unfolding store-def
apply (simp split: option.split-asm t.split-asm option.split t.split add: load-def
assms(2), safe; simp split: if-split-asm)
using assms(1) store-cond-cap-frag apply blast
      apply (metis assms(1) option.sel store-cond-cap-frag t.distinct(1))
      apply (metis assms(1) store-bounds-preserved)
      apply (metis assms(1) store-bounds-preserved)
apply (metis Mapping.lookup-update heap.select-convs(2) heap.surjective heap.update-convs(2)
option.sel t.sel)
done

```

5.1.3 Miscellaneous Laws

lemma free-after-alloc:

```

assumes alloc h c s = Success (h', cap)
shows  $\exists!$  ret. free h' cap = Success ret
using alloc-def assms free-non-null-correct alloc-no-null-ret
by force

```

lemma store-after-alloc:

```

assumes alloc h True s = Success (h', cap)
and memval-type v| $\tau$   $\leq$  s
and v  $\neq$  Undef
shows  $\exists$  h''. store h' cap v = Success h''
proof -
let ?m = (bounds = (0, s), content = Mapping.empty, tags = Mapping.empty)
have tag cap = True
      using assms(1) alloc-def
      by fastforce
moreover have perm-store cap = True
      using assms(1) alloc-def
      by fastforce
moreover have  $\bigwedge$  cv. [v = Cap-v cv; tag cv]  $\implies$  perm-cap-store cap  $\wedge$  (perm-cap-store-local
cap  $\vee$  perm-global cv)
proof -
      have  $\neg$  (case v of Cap-v cv  $\implies$   $\neg$  perm-cap-store cap  $\wedge$  tag cv | -  $\implies$  False)
        using assms unfolding alloc-def
        by (simp split: cval.split, force)
      moreover have  $\neg$  (case v of Cap-v cv  $\implies$   $\neg$  perm-cap-store-local cap  $\wedge$  tag cv
 $\wedge$   $\neg$  perm-global cv | -  $\implies$  False)
        using assms unfolding alloc-def

```

by (*simp split: ccval.split, force*)
ultimately show $\bigwedge cv. \llbracket v = \text{Cap-}v \text{ cv}; \text{tag } cv \rrbracket \implies \text{perm-cap-store } cap \wedge$
 (*perm-cap-store-local cap \vee perm-global cv*)
 by *force*
qed
moreover have $\text{offset } cap + |\text{memval-type } v|_\tau \leq \text{base } cap + \text{len } cap$
 using *assms alloc-def*
 by *fastforce*
moreover have $\text{offset } cap \geq \text{base } cap$
 using *assms alloc-def*
 by *fastforce*
moreover have $\text{offset } cap \bmod |\text{memval-type } v|_\tau = 0$
 using *assms alloc-def*
 by *fastforce*
moreover have $\text{Mapping.lookup } (\text{heap-map } h') (\text{block-id } cap) = \text{Some } (\text{Map } ?m)$
 using *assms alloc-def*
 by *fastforce*
moreover have $\text{offset } cap \geq \text{fst } (\text{bounds } ?m)$
 using *assms alloc-def*
 by *fastforce*
moreover have $\text{offset } cap + |\text{memval-type } v|_\tau \leq \text{snd } (\text{bounds } ?m)$
 using *assms alloc-def*
 by *fastforce*
ultimately show *?thesis*
 using *store-success* [where *?c=cap and ?v=v and ?m=(bounds = (0, s), content = Mapping.empty, tags = Mapping.empty)* and *?h=h'*] *assms(3)*
 by *simp (blast)*
qed

lemma *store-after-alloc-gen:*

assumes *alloc h True s = Success (h', cap)*
 and $|\text{memval-type } v|_\tau \leq s$
 and $v \neq \text{Undef}$
 and $n \bmod |\text{memval-type } v|_\tau = 0$
 and $\text{offset } cap + n + |\text{memval-type } v|_\tau \leq \text{base } cap + \text{len } cap$
shows $\exists h''. \text{store } h' (cap \ () \ \text{offset} := \text{offset } cap + n \ ()) v = \text{Success } h''$
proof –
let *?m = (bounds = (0, s), content = Mapping.empty, tags = Mapping.empty)*
have *tag cap = True*
 using *assms(1) alloc-def*
 by *fastforce*
moreover have $\text{perm-store } (cap \ () \ \text{offset} := \text{offset } cap + n \ ()) = \text{True}$
 using *assms(1) alloc-def*
 by *fastforce*
moreover have $\bigwedge cv. \llbracket v = \text{Cap-}v \text{ cv}; \text{tag } cv \rrbracket \implies \text{perm-cap-store } (cap \ () \ \text{offset} := \text{offset } cap + n \ ()) \wedge$
 (*perm-cap-store-local (cap () offset := offset cap + n ()) \vee perm-global cv*)
proof –
have $\neg (\text{case } v \text{ of } \text{Cap-}v \text{ cv} \Rightarrow \neg \text{perm-cap-store } (cap \ () \ \text{offset} := \text{offset } cap +$

```

n )) ∧ tag cv | - ⇒ False)
  using assms unfolding alloc-def
  by (simp split: ccval.split, force)
  moreover have ¬ (case v of Cap-v cv ⇒ ¬ perm-cap-store-local (cap () offset
:= offset cap + n )) ∧ tag cv ∧ ¬ perm-global cv | - ⇒ False)
  using assms unfolding alloc-def
  by (simp split: ccval.split, force)
  ultimately show ∧ cv. [v = Cap-v cv; tag cv] ⇒ perm-cap-store (cap () offset
:= offset cap + n )) ∧ (perm-cap-store-local (cap () offset := offset cap + n )) ∨
perm-global cv)
  by force
qed
  moreover have offset (cap () offset := offset cap + n )) + |memval-type v|τ ≤
base (cap () offset := offset cap + n )) + len (cap () offset := offset cap + n ))
  using assms alloc-def
  by fastforce
  moreover have offset (cap () offset := offset cap + n )) ≥ base (cap () offset :=
offset cap + n ))
  using assms alloc-def
  by fastforce
  moreover have offset (cap () offset := offset cap + n )) mod |memval-type v|τ
= 0
  using assms alloc-def
  by fastforce
  moreover have Mapping.lookup (heap-map h') (block-id (cap () offset := offset
cap + n )) = Some (Map ?m)
  using assms alloc-def
  by fastforce
  moreover have offset (cap () offset := offset cap + n )) ≥ fst (bounds ?m)
  using assms alloc-def
  by fastforce
  moreover have offset (cap () offset := offset cap + n )) + |memval-type v|τ ≤
snd (bounds ?m)
  using assms alloc-def
  by fastforce
  ultimately show ?thesis
  using store-success[where ?c=(cap () offset := offset cap + n )) and ?v=v
and ?m=(bounds = (0, s), content = Mapping.empty, tags = Mapping.empty)
and ?h=h'] assms(3)
  by simp (blast)
qed

```

5.2 Well-formedness of actions

lemma alloc-wellformed:

```

assumes  $\mathcal{W}_f(\text{heap-map } h)$ 
  and alloc h True s = Success (h', cap)
shows  $\mathcal{W}_f(\text{heap-map } h')$ 
using assms

```

by (*simp add: alloc-def wellformed-def, safe, erule-tac x=b in allE, erule-tac x=obj in allE, simp*)
(smt (verit, best) Mapping.keys-empty Mapping.lookup-update Mapping.lookup-update-neq Set.filter-def bot-nat-0.not-eq-extremum empty-iff mem-Collect-eq object.select-convs(3) option.sel t.sel zero-less-diff)

lemma *free-wellformed:*

assumes $\mathcal{W}_f(\text{heap-map } h)$
and *free* h $cap = \text{Success } (h', cap')$
shows $\mathcal{W}_f(\text{heap-map } h')$

proof –

consider $(null) \text{ cap} = \text{NULL} \mid (non-null) \text{ cap} \neq \text{NULL}$ **by** *blast*

then show *?thesis*

proof (*cases*)

case *null*

show *?thesis*

using *free-null assms null*

by *simp*

next

case *non-null*

show *?thesis*

by (*smt (z3) Mapping.lookup-update-neq assms(1) assms(2) free-cond-non-null(3)*)

free-cond-non-null(4) free-cond-non-null(5) free-def free-non-null-correct fst-conv heap.select-convs(2) heap.surjective heap.update-convs(2) option.sel result.sel(1) t.discI wellformed-def)

qed

qed

lemma *load-wellformed:*

assumes $\mathcal{W}_f(\text{heap-map } h)$
and *load* h c $t = \text{Success } v$
shows $\mathcal{W}_f(\text{heap-map } h)$
using *assms(1)*
by *assumption*

lemma *store-wellformed:*

assumes $\mathcal{W}_f(\text{heap-map } h)$
and *store* h c $v = \text{Success } h'$
shows $\mathcal{W}_f(\text{heap-map } h')$

using *store-cond-hard-cap(1)[where ?h=h and ?c=c and ?v=v and ?ret=h', OF assms(2)]*

store-cond-hard-cap(2)[where ?h=h and ?c=c and ?v=v and ?ret=h', OF assms(2)]

store-cond-hard-cap(4)[where ?h=h and ?c=c and ?v=v and ?ret=h', OF assms(2)]

store-cond-hard-cap(5)[where ?h=h and ?c=c and ?v=v and ?ret=h', OF assms(2)]

store-cond-hard-cap(6)[where ?h=h and ?c=c and ?v=v and ?ret=h', OF assms(2)]

```

  assms
apply (simp add: store-def wellformed-def split: option.split-asm t.split-asm if-split-asm
del: memval-type.simps)
apply safe
apply (clarsimp simp del: memval-type.simps)
apply (erule-tac x=block-id c in allE)
apply (rename-tac map nth block obj x)
apply (erule-tac x=map in allE)
apply (clarsimp simp del: memval-type.simps)
apply (subgoal-tac Set.filter (λx. 0 < x mod |Cap|τ) (Mapping.keys (tags (store-tval
map (nat (int |memval-type v|τ * nth)) v))) = {})
apply (smt (verit, best) Mapping.lookup-update Mapping.lookup-update-neq Set.member-filter
assms(1) emptyE option.sel rel-simps(70) t.sel wellformed-def)
apply (simp add: store-tval-def del: memval-type.simps split: cval.split-asm; fast-
force)
done

```

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 memcpyy-prim :: heap ⇒ cap ⇒ cap ⇒ nat ⇒ heap result
and memcpyy-cap :: heap ⇒ cap ⇒ cap ⇒ nat ⇒ heap result
where
  memcpyy-prim h - - 0 = Success h
| memcpyy-cap h - - 0 = Success h
| memcpyy-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
   memcpyy-cap ys (dst (| offset := (offset dst + 1) |)) (src (| offset := (offset src)
+ 1)) n)
| memcpyy-cap h dst src (Suc n) =
  (if (Suc n) < |Cap|τ then memcpyy-prim h dst src (Suc n)
  else
   let x = load h src Cap in
   if ¬ is-Success x then memcpyy-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|τ))) (src (| offset := (offset
src + |Cap|τ))) (Suc n - |Cap|τ)
  by (blast | metis old.nat.exhaust prod-cases3 sumE)+

```

We prove that the mutually recursive function terminates.

```

context
  notes sizeof-def[simp]
begin
termination by size-change
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  $\mathcal{W}_f(\text{heap-map } h)$ 
shows memcpy-prim h dst src n = Success h' ⇒  $\mathcal{W}_f(\text{heap-map } h')$ 
  and memcpy-cap h dst src n = Success h' ⇒  $\mathcal{W}_f(\text{heap-map } h')$ 
using assms
proof(induct h dst src n and h dst src n rule: memcpy-prim-memcpy-cap.induct)
  case (1 h uu uv)
  then show ?case
    by force
next
  case (2 h uw ux)
  then show ?case
    by force
next
  case (3 h dst src n)
  then show ?case
    by clarsimp (smt (verit) result.disc(1) result.distinct(1) result.expand result.sel(1)
store-wellformed)
next
  case (4 h dst src n)

```



```

then show ?case
  by clarsimp (smt (verit, best) 4.hyps(1) 4.hyps(2) 4.hyps(3) 4.hyps(4) re-
    sult.collapse(1) store-wellformed)
qed

```

We also prove that memcopy preserves well-formedness.

```

lemma memcopy-wellformed:
  assumes  $\mathcal{W}_f(\text{heap-map } h)$ 
  and memcopy h dst src n = Success h'
  shows  $\mathcal{W}_f(\text{heap-map } h')$ 
  using assms unfolding memcopy-def
  by (metis memcopy-rec-wellformed(2) result.distinct(1) result.sel(1))

```

```

lemma memcopy-cond:
  assumes memcopy h dst src n = Success h'
  shows  $n > 0 \longrightarrow \neg (\text{block-id } \text{dst} = \text{block-id } \text{src} \wedge$ 
     $((\text{offset } \text{src} \geq \text{offset } \text{dst} \wedge \text{offset } \text{src} < \text{offset } \text{dst} + n) \vee$ 
     $(\text{offset } \text{dst} \geq \text{offset } \text{src} \wedge \text{offset } \text{dst} < \text{offset } \text{src} + n)))$ 
  using assms unfolding memcopy-def
  by force

```

6 Miscellaneous Definitions

The following are used for catching memory leaks in Gillian.

```

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

```

```

primrec get-memory-leak-size :: heap  $\Rightarrow$  nat  $\Rightarrow$  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  $\Rightarrow$  0
    | Some n  $\Rightarrow$  n)

```

```

primrec get-unfreed-blocks :: heap  $\Rightarrow$  nat  $\Rightarrow$  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  $\Rightarrow$  get-unfreed-blocks h n | Some m  $\Rightarrow$ 
        (case m of Freed  $\Rightarrow$  get-unfreed-blocks h n | -  $\Rightarrow$  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
  using assms
  unfolding alloc-glob-var-def alloc-def
  by fastforce

```

```

lemma set-glob-var-glob-bit-lift:
  assumes set-glob-var h c s v g = Success (h', cap, g')
  shows perm-global cap
  using assms
  unfolding alloc-glob-var-def set-glob-var-def alloc-def
  by fastforce

```

```

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

```

```

by (metis alloc-updated-heap-and-cap assms capability.select-convs(1) free-global-cap
      mem-capability.select-convs(10) mem-capability.simps(21) null-capability-def
      result.sel(1)
      alloc-glob-var-def snd-conv zero-mem-capability-ext-def)

```

```

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 –
  have res: alloc-glob-var h c s = Success (h', cap)
    using assms
    unfolding set-glob-var-def alloc-glob-var-def alloc-def
    by fastforce
  show ?thesis using free-fails-on-glob-var[OF res]
    by blast
qed

```

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 PermitStore-
CapViolation
  PermitStoreLocalCapViolation LengthViolation BadAddressViolation
  UseAfterFree BufferOverrun MissingResource WrongMemVal MemoryNot-
Freed Unhandled
  in OCaml
  file-prefix CHERI-C-Memory-Model

end

```

References

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