CHERI C Concrete Memory Model Theory

Seung Hoon Park

October 19, 2022

Contents

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

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*

Type specifier

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

```
  | Sint32
  | Uint64
  | Sint64
  | Cap
```

Type value specifier (used primarily for semantic evaluation)

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

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

Because sizeof depends on the architeture, it shall be given via the memory model

Encoding/Decoding mathematical values and machine words

Unsigned

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

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

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

encode-u8-list w ≡ [*w*]

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

lemma *encode-decode-u8-list*:
ls = [*b*] ⟹ *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* ≤ *2* ^ *LENGTH*(*8*) − *1*
shows *i* = *decode-u8* (*encode-u8 i*)
by (*metis assms le-unat-uoi unat-minus-one-word*)

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

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

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

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

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

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

lemma *u16-split-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*)) *mod LENGTH*((*'a*::*len*)) = *0*
and *length w* = *LENGTH*(*'b*) *div LENGTH*(*'a*)
shows (*word-rsplit* :: *'b word* ⇒ *'a word list*) ((*word-rcat* :: *'a word list* ⇒ *'b word*) *w*) = *w*
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* ⇒ *8 word list*) ((*word-rcat* :: *8 word list* ⇒ *16 word*) *w*) = *w*
proof −
have *l1*: *length w* ∗ *8* = *16*
using *assms* **by** *clarsimp*
moreover have *l2*: *size* ((*word-rcat* :: *8 word list* ⇒ *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* ⇒ *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* ≤ *2* ^ *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* ⇒ *8 word list*
where
flatten-u32 x ≡ (*word-rsplit* :: *32 word* ⇒ *8 word list*) *x*

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

abbreviation *encode-u32* :: *nat* ⇒ *8 word list*

where
encode-u32 x ≡ *flatten-u32* (*word-of-nat x*)

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

lemma *flatten-u32-length*:
length (*flatten-u32 vs*) = *4*
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* ⇒ *8 word list*) ((*word-rcat* :: *8 word list* ⇒ *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* ≤ *2* ^ *LENGTH*(*32*) − *1*
shows *i* = *decode-u32* (*encode-u32 i*)
by (*metis assms le-unat-uoi unat-minus-one-word word-rcat-rsplit*)

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

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

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

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

lemma *flatten-u64-length*:
length (*flatten-u64 vs*) = *8*
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* = *encode-u64* (*decode-u64 w*)
using *assms*
by (*simp add*: *rsplit-rcat-eq*)

lemma *cat-flatten-u64-eq*:
cat-u64 (*flatten-u64 w*) = *w*
by (*simp add*: *word-rcat-rsplit*)

lemma *decode-encode-u64*:
assumes *i* ≤ *2* ^ *LENGTH*(*64*) − *1*
shows *i* = *decode-u64* (*encode-u64 i*)
by (*metis assms le-unat-uoi unat-minus-one-word word-rcat-rsplit*)

Signed

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

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

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

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

lemma *encode-decode-s8-list*:
ls = [*b*] ⟹ *ls* = *encode-s8-list* (*decode-s8-list ls*)
by *simp*

lemma *decode-encode-s8-list*:
w = *decode-s8-list* (*encode-s8-list w*)
by *simp*

lemma *encode-decode-s8*:
w = *encode-s8* (*decode-s8 w*)
by *simp*

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

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

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

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

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

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

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

lemma *flatten-s16-length*:
length (*s16-split vs*) = *2*
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* ⇒ *8 word list*) ((*word-rcat* :: *8 word list* ⇒ *16 sword*) *w*) = *w*
proof −
have *l1*: *length w* ∗ *8* = *16*
using *assms* **by** *clarsimp*
moreover have *l2*: *size* ((*word-rcat* :: *8 word list* ⇒ *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* ⇒ *16 sword*) *w*)
by *argo*
thus *?thesis*
by (*simp add*: *word-rsplit-rcat-size*)
qed

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

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

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

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

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

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

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

lemma *flatten-s32-length*:
length (*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 (*word-rsplit* :: *32 sword* $\Rightarrow$ *8 word list*) ((*word-rcat* :: *8 word list* $\Rightarrow$ *32 sword*) w) $= w$
using *rsplit-rcat-eq assms*
by *force*

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

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

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

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

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

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

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

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

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

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

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

definition *integer-of-word* :: *'a*::*len word* ⇒ *integer*
where

integer-of-word x ≡ integer-of-int (uint x)

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

lemma *word-integer-eq*:
word-of-integer (integer-of-word w) = w
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 ≤ i*
assumes *i ≤ 2 ^ LENGTH('a::len) − 1*
shows *integer-of-word ((word-of-integer :: integer ⇒ '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 ^ (LENGTH('a::len) − 1)) ≤ i*
and *i < 2 ^ (LENGTH('a) − 1)*
shows *integer-of-sword ((sword-of-integer :: integer ⇒ '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 ⇒ 8 word*
where
word8-of-integer ≡ word-of-integer

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

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

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

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

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

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

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

lemma *word8-integer-eq*:
word8-of-integer (*integer-of-word8 w*) = *w*
unfolding *word8-of-integer-def integer-of-word8-def*
using *word-integer-eq*
by *blast*

lemma *word16-integer-eq*:
word16-of-integer (*integer-of-word16 w*) = *w*
unfolding *word16-of-integer-def integer-of-word16-def*
using *word-integer-eq*
by *blast*

lemma *word32-integer-eq*:
word32-of-integer (*integer-of-word32 w*) = *w*
unfolding *word32-of-integer-def integer-of-word32-def*
using *word-integer-eq*
by *blast*

lemma *word64-integer-eq*:
word64-of-integer (*integer-of-word64 w*) = *w*
unfolding *word64-of-integer-def integer-of-word64-def*
using *word-integer-eq*
by *blast*

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

lemma *integer-word16-bounded-eq*:
assumes $0 \leq i$
and $i \leq 2 \hat{} LENGTH(16) - 1$
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 $i \leq 2 \hat{} LENGTH(32) - 1$
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 $i \leq 2 \hat{} LENGTH(64) - 1$
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 *sword8-of-integer* :: *integer* $\Rightarrow$ *8 sword*
where
sword8-of-integer $\equiv$ *sword-of-integer*

definition *sword16-of-integer* :: *integer* $\Rightarrow$ *16 sword*
where
sword16-of-integer $\equiv$ *sword-of-integer*

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

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

definition *integer-of-sword8* :: *8 sword* $\Rightarrow$ *integer*

where
integer-of-sword8 ≡ *integer-of-sword*

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

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

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

lemma *sword8-integer-eq*:
sword8-of-integer (*integer-of-sword8 w*) = *w*
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* ^ (*LENGTH*(*8*) − *1*)) ≤ *i*
and *i* < *2* ^ (*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* ^ (*LENGTH*(*16*) − *1*)) ≤ *i*

and $i < 2 \hat{} (LENGTH(16) - 1)$
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 $-(2 \hat{} (LENGTH(32) - 1)) \leq i$
and $i < 2 \hat{} (LENGTH(32) - 1)$
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 $-(2 \hat{} (LENGTH(64) - 1)) \leq i$
and $i < 2 \hat{} (LENGTH(64) - 1)$
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*

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

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

begin

— These are coprocessor 2 excessptions thrown by the hardware. BadAddressViolation is not a coprocessor 2 exception but remains one given by the hardware.
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.

datatype *logicerrtype* =
UseAfterFree
| *BufferOverrun*
| *MissingResource*
| *WrongMemVal*
| *MemoryNotFreed*
| *Unhandled String.literal*

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

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 architeture, it shall be given via the memory model

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

lemma *size-type-align*:
$|t|_\tau = x \Longrightarrow \exists\ n.\ 2\ \hat{}\ n = x$
apply (*simp add*: *sizeof-def split*: *cctype.split-asm*)

apply *fastforce+*
apply (*rule-tac x=1* **in** *exI*, *fastforce*)
apply (*rule-tac x=1* **in** *exI*, *fastforce*)
apply (*rule-tac x=2* **in** *exI*, *fastforce*)
apply (*rule-tac x=2* **in** *exI*, *fastforce*)
apply (*rule-tac x=3* **in** *exI*, *fastforce*)
apply (*rule-tac x=3* **in** *exI*, *fastforce*)
apply (*rule-tac x=5* **in** *exI*, *fastforce*)
done

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*)$|_\tau$ = *8*
unfolding *sizeof-def*
by *fastforce*

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

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

corollary *memval-size-u16-eq-word-split-len*:
assumes *val = Uint16-v v*
shows *|memval-type val|$_\tau$ = length (u16-split v)*
using *assms memval-size-u16 u16-split-length*
by *force*

corollary *memval-size-s16-eq-word-split-len*:
assumes *val = Sint16-v v*
shows *|memval-type val|$_\tau$ = 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|$_\tau$ = 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|$_\tau$ = 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|$_\tau$ = 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-v v*
shows *|memval-type val|$_\tau$ = length (flatten-s64 v)*
using *assms memval-size-s64 flatten-s64-length*
by *force*

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

instance *int* :: *comp-countable* **..**

lemma *integer-encode-eq*: (*int-encode* ∘ *int-of-integer*) *x* = (*int-encode* ∘ *int-of-integer*)

$y \longleftrightarrow x = y$
using *int-encode-eq integer-eq-iff*
by *auto*

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

instance *integer* :: *comp-countable* **..**

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

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-byte-def split*: *memval.split-asm*)

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

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

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

record *heap* =
next-block :: *block*

heap-map :: (*block*, *t*) *mapping*

1 Proving heap is an instance of a 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

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
apply *standard*
apply (*simp add*: *sep-disj-map-def Mapping.keys-def zero-map-def*)
apply (*metis Mapping.keys.rep-eq Mapping.keys-empty inf-bot-right*)
apply (*simp add*: *sep-disj-map-def Mapping.keys-def zero-map-def*)
apply (*simp add*: *inf-commute*)
apply (*simp add*: *sep-disj-map-def Mapping.keys-def zero-map-def plus-map-def*)

apply (*metis Mapping.empty-def Mapping.lookup.abs-eq map-add-empty rep-inverse*)
apply (*simp add*: *sep-disj-map-def Mapping.keys-def zero-map-def plus-map-def Mapping.lookup-def map-add-comm*)
apply (*fastforce dest*: *map-add-comm*)
apply (*simp add*: *sep-disj-map-def Mapping.keys-def zero-map-def plus-map-def Mapping.lookup-def map-add-comm*)
apply (*simp add*: *Mapping-inverse*)
apply (*simp add*: *sep-disj-map-def Mapping.keys-def zero-map-def plus-map-def Mapping.lookup-def map-add-comm*)

apply (*metis* (*no-types*, *opaque-lifting*) *Mapping.keys.abs-eq Mapping.keys.rep-eq disjoint-iff domIff map-add-dom-app-simps*(*3*))
apply (*simp add*: *sep-disj-map-def Mapping.keys-def zero-map-def plus-map-def Mapping.lookup-def map-add-comm*)
apply (*simp add*: *Mapping-inverse inf-commute inf-sup-distrib1*)
apply (*simp add*: *plus-map-def sep-disj-map-def*)
apply (*metis* (*mono-tags*, *opaque-lifting*) *Mapping.keys.abs-eq Mapping.lookup.abs-eq disjoint-iff domIff map-add-dom-app-simps*(*3*) *mapping-eqI*)
done
end

instantiation *heap-ext* :: (*cancellative-sep-algebra*) *cancellative-sep-algebra*
begin
definition *0* :: *'a heap-scheme* ≡ (| *next-block* = *0*, *heap-map* = *Mapping.empty*, ... = *0* |)
definition (*m1* :: *'a heap-scheme*) + (*m2* :: *'a heap-scheme*) ≡
(| *next-block* = *next-block m1* + *next-block m2*,
heap-map = *Mapping* ((*Mapping.lookup* (*heap-map m1*)) ++ (*Mapping.lookup* (*heap-map m2*))),
... = *heap.more m1* + *heap.more m2* |)
definition (*m1* :: *'a heap-scheme*) ## (*m2* :: *'a heap-scheme*) ≡
Mapping.keys (*heap-map m1*) ∩ *Mapping.keys* (*heap-map m2*) = {}
∧ *heap.more m1* ## *heap.more m2*
instance
apply *standard*
apply (*unfold plus-heap-ext-def sep-disj-heap-ext-def zero-heap-ext-def*)
apply *force*
apply (*simp add*: *inf-commute sep-disj-commute*)
apply (*simp add*: *Mapping.empty-def Mapping.lookup.abs-eq*, *simp add*: *Mapping.lookup.rep-eq rep-inverse*)
apply (*metis add.commute keys-dom-lookup map-add-comm sep-add-commute*)
apply (*simp add*: *Mapping.lookup.abs-eq sep-add-assoc*)
apply (*metis heap.select-convs*(*2*) *heap.select-convs*(*3*) *plus-map-def sep-disj-addD sep-disj-map-def*)
apply (*simp add*: *Mapping.lookup.abs-eq disjoint-iff keys-dom-lookup sep-disj-addI1*)
apply (*metis add-right-cancel heap.equality heap.ext-inject plus-map-def sep-add-cancel sep-disj-map-def*)
done
end

instantiation *mem-capability-ext* :: (*comp-countable*, *zero*) *zero*
begin
definition *0* :: (*'a*, *'b*) *mem-capability-scheme* ≡
(| *block-id* = *0*,
offset = *0*,
base = *0*,
len = *0*,
perm-load = *False*,
perm-cap-load = *False*,

perm-store = *False*,
perm-cap-store = *False*,
perm-cap-store-local = *False*,
perm-global = *False*,
... = *0*⦈
instance ..
end

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

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

— Section 4.5 of CHERI C/C++ Programming Guide defines what a NULL capability is.
definition *null-capability* :: *cap* (*NULL*)
where
NULL ≡ *0*

context
notes *null-capability-def*[*simp*]
begin

lemma *null-capability-block-id*[*simp*]:
block-id NULL = *0*
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

— Note that the starting block is 1, as 0 loosely refers to the null capability
definition *init-heap* :: *heap*
where
init-heap ≡ *0* (| *next-block* := *1* |)

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

definition *wellformed* :: (*block*, *t*) *mapping* ⇒ *bool* ($\mathcal{W}_\mathfrak{f}$/(-/))
where
$\mathcal{W}_\mathfrak{f}(h)$ ≡
∀ *b obj*. *Mapping.lookup h b* = *Some* (*Map obj*)
⟶ *Set.filter* (λ*x*. *x mod* $|Cap|_\tau$ ≠ *0*) (*Mapping.keys* (*tags obj*)) = {}

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

lemma *init-wellformed*:
$\mathcal{W}_\mathfrak{f}$(*heap-map init-heap*)
unfolding *init-heap-def wellformed-def zero-heap-ext-def*
by *simp*

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

lemma *heap-map h1 ## heap-map h2* $\Longrightarrow \mathcal{W}_{\mathfrak{f}}$(*heap-map h1 + heap-map h2*)
$\Longrightarrow \mathcal{W}_{\mathfrak{f}}$(*heap-map h1*) $\wedge \mathcal{W}_{\mathfrak{f}}$(*heap-map h2*)
apply (*unfold wellformed-def*)
apply *safe*
apply (*erule-tac x=b* **in** *allE*)
apply (*erule-tac x=obj* **in** *allE*)
apply (*fastforce intro*: *mapping-lookup-disj1 mapping-lookup-disj2*)+
done

definition *alloc* :: *heap* $\Rightarrow$ *bool* $\Rightarrow$ *nat* $\Rightarrow$ (*heap* $\times$ *cap*) *result*
where
alloc h c s $\equiv$
let cap = ⦇ *block-id* = (*next-block h*),
offset = *0*,
base = *0*,
len = *s*,
perm-load = *True*,
perm-cap-load = *c*,
perm-store = *True*,
perm-cap-store = *c*,
perm-cap-store-local = *c*,
perm-global = *False*,
tag = *True*
⦈ *in*
let h′ = *h* ⦇ *next-block* := (*next-block h*) + *1*,
heap-map := *Mapping.update*
(*next-block h*)
(*Map* ⦇ *bounds* = (*0*, *s*),
content = *Mapping.empty*,
tags = *Mapping.empty*
⦈
) (*heap-map h*)
⦈ *in*
Success (*h′*, *cap*)

lemma *alloc-wellformed*:

assumes $\mathcal{W}_{\mathfrak{f}}(\textit{heap-map } h)$
and *alloc h True s = Success* $(h', \textit{cap})$
shows $\mathcal{W}_{\mathfrak{f}}(\textit{heap-map } h')$
apply (*insert assms*)
apply (*simp add*: *alloc-def wellformed-def*)
apply *safe*
apply (*erule-tac x=b* **in** *allE*)
apply (*erule-tac x=obj* **in** *allE*)
apply *simp*
apply (*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)
done

lemma *alloc-always-success*:
∃! *res*. *alloc h c s = Success res*
by (*simp add*: *alloc-def*)

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

lemma *alloc-never-fails*:
alloc h c s = Error e ⟹ *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 *alloc h c s = Success* $(h', \textit{cap})$
shows *cap* ≠ *NULL*
proof −
have *perm-load cap*
using *assms alloc-def*
by *force*
moreover have ¬ *perm-load 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* $(h', \textit{cap})$
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

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)))
```

— Section 7.20.3.2 of The C Standard states free(NULL) results in no action occuring.

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

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.

lemma *free-double-free*:
assumes $c \neq NULL$
and $tag\ c = True$
and $perm\text{-}global\ c = False$
and $Mapping.lookup\ (heap\text{-}map\ h)\ (block\text{-}id\ c) = Some\ Freed$
shows $free\ h\ c = Error\ (LogicErr\ (UseAfterFree))$
using *free-def assms*
by *force*

— An incorrect offset implies the actual ptr value is not that returned by alloc. Section 7.20.3.2 of The C Standard states this leads to undefined behaviour. Clang, in practice, however, terminates the C program with an invalid pointer error.
lemma *free-incorrect-cap-offset*:
assumes $c \neq NULL$
and $tag\ c = True$
and $perm\text{-}global\ c = False$
and $Mapping.lookup\ (heap\text{-}map\ h)\ (block\text{-}id\ c) = Some\ (Map\ m)$
and $offset\ c \neq 0$
shows $free\ h\ c = Error\ (LogicErr\ (Unhandled\ 0))$
using *free-def assms*
by *force*

lemma *free-incorrect-bounds*:
assumes $c \neq NULL$
and $tag\ c = True$
and $perm\text{-}global\ c = False$
and $Mapping.lookup\ (heap\text{-}map\ h)\ (block\text{-}id\ c) = Some\ (Map\ m)$
and $offset\ c = 0$
and $bounds\ m \neq (base\ c,\ base\ c + len\ c)$
shows $free\ h\ c = Error\ (LogicErr\ (Unhandled\ 0))$
unfolding *free-def*
using *assms*
by *force*

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

shows *free h c = Success (h (| heap-map := Mapping.update (block-id c) Freed (heap-map h) |),*
c (| tag := False |))
unfolding *free-def*
using *assms*
by *simp*

lemma *free-cond*:
assumes *free h c = Success (h', cap)*
shows *c ≠ NULL ⟹ tag c = True*
and *c ≠ NULL ⟹ perm-global c = False*
and *c ≠ NULL ⟹ offset c = 0*
and *c ≠ NULL ⟹ ∃ m. Mapping.lookup (heap-map h) (block-id c) = Some (Map m) ∧*
bounds m = (base c, base c + len c)
and *c ≠ NULL ⟹ Mapping.lookup (heap-map h') (block-id c) = Some Freed*
and *c ≠ NULL ⟹ cap = c (| tag := False |)*
and *c = NULL ⟹ (h, c) = (h', cap)*
proof −
assume *c ≠ NULL*
thus *tag c = True*
using *assms* **unfolding** *free-def*
by (*meson result.simps(4)*)
next
assume *c ≠ NULL*
thus *perm-global c = False*
using *assms* **unfolding** *free-def*
by (*meson result.simps(4)*)
next
assume *c ≠ NULL*
thus *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 ≠ NULL*
thus *∃ m. Mapping.lookup (heap-map h) (block-id c) = Some (Map m) ∧*
bounds m = (base c, base c + len c)
using *assms* **unfolding** *free-def*
by (*metis assms free-double-free free-incorrect-bounds free-incorrect-cap-offset free-nonexistant-obj not-Some-eq result.distinct(1) t.exhaust*)
next
assume *c ≠ NULL*
hence *h' = h (| heap-map := Mapping.update (block-id c) Freed (heap-map h) |)*
using *assms* **unfolding** *free-def*
by (*smt (verit, ccfv-SIG) free-nonexistant-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 Mapping.lookup (heap-map h') (block-id c) = Some Freed
    by fastforce
next
  assume c ≠ NULL
  thus cap = c (| tag := 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 = NULL
  thus (h, c) = (h', 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 free-wellformed:
  assumes 𝒲𝔣(heap-map h)
    and free h cap = Success (h', cap')
  shows 𝒲𝔣(heap-map h')
proof −
  consider (null) cap = NULL | (non-null) cap ≠ 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
      apply (insert assms(2) non-null)
      apply (drule free-cond-non-null, simp)
      apply (insert assms(1) free-cond-non-null[OF assms(2) non-null])
      apply (simp add: wellformed-def)
      apply safe
      apply (erule-tac x=b in allE)
      apply (erule-tac x=obj in allE)
        apply(smt (z3) Mapping.lookup-update-neq Pair-inject Set.member-filter
assms(2)
       free-non-null-correct gr0I heap.ext-inject heap.surjective heap.update-convs(2)

          option.inject result.inject(1) t.discI zero-less-diff)
      done
    qed
qed

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

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 ⟹ 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 ⟹ 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 ⇒ nat ⇒ nat ⇒ 8 word list
  where
  retrieve-bytes m - 0 = []
| retrieve-bytes m off (Suc siz) = of-byte (the (Mapping.lookup m off)) # retrieve-bytes
m (Suc off) siz

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


definition retrieve-tval :: object ⇒ nat ⇒ cctype ⇒ bool ⇒ block ccval
  where
  retrieve-tval obj off typ pcl ≡
    if is-contiguous-bytes (content obj) off |typ|τ then
      (case typ of
        Uint8 ⇒ Uint8-v (decode-u8-list (retrieve-bytes (content obj) off |typ|τ))
      | Sint8 ⇒ Sint8-v (decode-s8-list (retrieve-bytes (content obj) off |typ|τ))
      | Uint16 ⇒ Uint16-v (cat-u16 (retrieve-bytes (content obj) off |typ|τ))
      | Sint16 ⇒ Sint16-v (cat-s16 (retrieve-bytes (content obj) off |typ|τ))
      | Uint32 ⇒ Uint32-v (cat-u32 (retrieve-bytes (content obj) off |typ|τ))
```

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

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))))
```

lemma *load-wellformed*:

assumes $\mathcal{W}_{\mathfrak{f}}(\textit{heap-map } h)$
and $\textit{load } h\ c\ t = \textit{Success } v$
shows $\mathcal{W}_{\mathfrak{f}}(\textit{heap-map } h)$
by (*insert assms*(*1*), *assumption*)

lemma *load-null-error*:
$\textit{load } h\ \textit{NULL}\ t = \textit{Error } (\textit{C2Err TagViolation})$
unfolding *load-def*
by *simp*

lemma *load-false-tag*:
assumes $\textit{tag } c = \textit{False}$
shows $\textit{load } h\ c\ t = \textit{Error } (\textit{C2Err TagViolation})$
unfolding *load-def*
using *assms*
by *presburger*

lemma *load-false-perm-load*:
assumes $\textit{tag } c = \textit{True}$
and $\textit{perm-load } c = \textit{False}$
shows $\textit{load } h\ c\ t = \textit{Error } (\textit{C2Err PermitLoadViolation})$
unfolding *load-def*
using *assms*
by *presburger*

lemma *load-bound-over*:
assumes $\textit{tag } c = \textit{True}$
and $\textit{perm-load } c = \textit{True}$
and $\textit{offset } c + |t|_\tau > \textit{base } c + \textit{len } c$
shows $\textit{load } h\ c\ t = \textit{Error } (\textit{C2Err LengthViolation})$
unfolding *load-def*
using *assms*
by *presburger*

lemma *load-bound-under*:
assumes $\textit{tag } c = \textit{True}$
and $\textit{perm-load } c = \textit{True}$
and $\textit{offset } c + |t|_\tau \leq \textit{base } c + \textit{len } c$
and $\textit{offset } c < \textit{base } c$
shows $\textit{load } h\ c\ t = \textit{Error } (\textit{C2Err LengthViolation})$
unfolding *load-def*
using *assms*
by *presburger*

lemma *load-misaligned*:
assumes $\textit{tag } c = \textit{True}$
and $\textit{perm-load } c = \textit{True}$
and $\textit{offset } c + |t|_\tau \leq \textit{base } c + \textit{len } c$
and $\textit{offset } c \geq \textit{base } c$

and *offset c mod* $|t|_\tau \neq 0$
shows *load h c t* = *Error* (*C2Err BadAddressViolation*)
unfolding *load-def*
using *assms*
by *force*

lemma *load-nonexistant-obj*:
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*) = *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 mod* $|t|_\tau = 0$
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 mod* $|t|_\tau = 0$
and *Mapping.lookup* (*heap-map h*) (*block-id c*) = *Some* (*Map m*)
and *is-contiguous-bytes* (*content m*) (*nat* (*offset c*)) $|t|_\tau$
and *t* = *Cap*
and ¬ *is-contiguous-zeros* (*content m*) (*nat* (*offset c*)) $|t|_\tau$
and *offset c* < *fst* (*bounds m*)
shows *load h c t* = *Error* (*LogicErr BufferOverrun*)
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 mod* $|t|_\tau$ = *0*
and *Mapping.lookup* (*heap-map h*) (*block-id c*) = *Some* (*Map m*)
and *is-contiguous-bytes* (*content m*) (*nat* (*offset c*)) $|t|_\tau$
and *t* = *Cap*
and ¬ *is-contiguous-zeros* (*content m*) (*nat* (*offset c*)) $|t|_\tau$
and *offset c* + $|t|_\tau$ > *snd* (*bounds m*)
shows *load h c t* = *Error* (*LogicErr BufferOverrun*)
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 *offset c mod* $|t|_\tau$ = *0*
and *Mapping.lookup* (*heap-map h*) (*block-id c*) = *Some* (*Map m*)
and *is-contiguous-bytes* (*content m*) (*nat* (*offset c*)) $|t|_\tau$
and *t* = *Cap*
and ¬ *is-contiguous-zeros* (*content m*) (*nat* (*offset c*)) $|t|_\tau$
and *offset c* $\geq$ *fst* (*bounds m*)
and *offset c* + $|t|_\tau \leq$ *snd* (*bounds m*)
shows *load h c t* = *Success Undef*
unfolding *load-def retrieve-tval-def*
using *assms*
by *fastforce*

lemma *load-null-cap-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 mod* $|t|_\tau$ = *0*
and *Mapping.lookup* (*heap-map h*) (*block-id c*) = *Some* (*Map m*)
and *is-contiguous-bytes* (*content m*) (*nat* (*offset c*)) $|t|_\tau$
and *t* = *Cap*
and *offset c* $\geq$ *fst* (*bounds m*)
and *offset c* + $|t|_\tau \leq$ *snd* (*bounds m*)
and *is-contiguous-zeros* (*content m*) (*nat* (*offset c*)) $|t|_\tau$
shows *load h c t* = *Success* (*Cap-v NULL*)
unfolding *load-def retrieve-tval-def*
using *assms*
by *fastforce*

lemma *load-u8-on-membytes*:
assumes *tag c* = *True*

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

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

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

lemma *load-s16-on-membytes*:
assumes *tag c* = *True*
and *perm-load c* = *True*
and *offset c* + $|t|_\tau$ ≤ *base c* + *len c*
and *offset c* ≥ *base c*
and *offset c mod* $|t|_\tau$ = *0*
and *Mapping.lookup* (*heap-map h*) (*block-id c*) = *Some* (*Map m*)
and *offset c* ≥ *fst* (*bounds m*)
and *offset c* + $|t|_\tau$ ≤ *snd* (*bounds m*)
and *is-contiguous-bytes* (*content m*) (*nat* (*offset c*)) $|t|_\tau$
and *t* = *Sint16*
shows *load h c t* = *Success* (*Sint16-v* (*cat-s16* (*retrieve-bytes* (*content m*) (*nat* (*offset c*)) $|t|_\tau$)))
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* + $|t|_\tau$ ≤ *base c* + *len c*
and *offset c* ≥ *base c*
and *offset c mod* $|t|_\tau$ = *0*
and *Mapping.lookup* (*heap-map h*) (*block-id c*) = *Some* (*Map m*)
and *offset c* ≥ *fst* (*bounds m*)
and *offset c* + $|t|_\tau$ ≤ *snd* (*bounds m*)
and *is-contiguous-bytes* (*content m*) (*nat* (*offset c*)) $|t|_\tau$
and *t* = *Uint32*
shows *load h c t* = *Success* (*Uint32-v* (*cat-u32* (*retrieve-bytes* (*content m*) (*nat* (*offset c*)) $|t|_\tau$)))
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* + $|t|_\tau$ ≤ *base c* + *len c*
and *offset c* ≥ *base c*
and *offset c mod* $|t|_\tau$ = *0*
and *Mapping.lookup* (*heap-map h*) (*block-id c*) = *Some* (*Map m*)
and *offset c* ≥ *fst* (*bounds m*)
and *offset c* + $|t|_\tau$ ≤ *snd* (*bounds m*)
and *is-contiguous-bytes* (*content m*) (*nat* (*offset c*)) $|t|_\tau$
and *t* = *Sint32*
shows *load h c t* = *Success* (*Sint32-v* (*cat-s32* (*retrieve-bytes* (*content m*) (*nat* (*offset c*)) $|t|_\tau$)))
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* + $|t|_\tau$ ≤ *base c* + *len c*
and *offset c* ≥ *base c*
and *offset c mod* $|t|_\tau$ = *0*
and *Mapping.lookup* (*heap-map h*) (*block-id c*) = *Some* (*Map m*)
and *offset c* ≥ *fst* (*bounds m*)
and *offset c* + $|t|_\tau$ ≤ *snd* (*bounds m*)
and *is-contiguous-bytes* (*content m*) (*nat* (*offset c*)) $|t|_\tau$
and *t* = *Uint64*
shows *load h c t* = *Success* (*Uint64-v* (*cat-u64* (*retrieve-bytes* (*content m*) (*nat* (*offset c*)) $|t|_\tau$)))
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* + $|t|_\tau$ ≤ *base c* + *len c*
and *offset c* ≥ *base c*
and *offset c mod* $|t|_\tau$ = *0*
and *Mapping.lookup* (*heap-map h*) (*block-id c*) = *Some* (*Map m*)
and *offset c* ≥ *fst* (*bounds m*)
and *offset c* + $|t|_\tau$ ≤ *snd* (*bounds m*)
and *is-contiguous-bytes* (*content m*) (*nat* (*offset c*)) $|t|_\tau$
and *t* = *Sint64*
shows *load h c t* = *Success* (*Sint64-v* (*cat-s64* (*retrieve-bytes* (*content m*) (*nat* (*offset c*)) $|t|_\tau$)))
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* + $|t|_\tau$ ≤ *base c* + *len c*
and *offset c* ≥ *base c*
and *offset c mod* $|t|_\tau$ = *0*
and *Mapping.lookup* (*heap-map h*) (*block-id c*) = *Some* (*Map m*)
and *offset c* ≥ *fst* (*bounds m*)
and *offset c* + $|t|_\tau$ ≤ *snd* (*bounds m*)
and ¬ *is-contiguous-bytes* (*content m*) (*nat* (*offset c*)) $|t|_\tau$
and ¬ *is-cap* (*content m*) (*nat* (*offset c*))
shows *load h c t* = *Success Undef*
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|$_\tau$ ≤ base c + len c*
and *offset c ≥ base c*
and *offset c mod |t|$_\tau$ = 0*
and *Mapping.lookup (heap-map h) (block-id c) = Some (Map m)*
and *offset c ≥ fst (bounds m)*
and *offset c + |t|$_\tau$ ≤ snd (bounds m)*
and *¬ is-contiguous-bytes (content m) (nat (offset c)) |t|$_\tau$*
and *is-cap (content m) (nat (offset c))*
and *mc = get-cap (content m) (nat (offset c))*
and *¬ is-contiguous-cap (content m) mc (nat (offset c)) |t|$_\tau$*
and *t ≠ Uint8*
and *t ≠ Sint8*
shows *load h c t = Success Undef*
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|$_\tau$ ≤ base c + len c*
and *offset c ≥ base c*
and *offset c mod |t|$_\tau$ = 0*
and *Mapping.lookup (heap-map h) (block-id c) = Some (Map m)*
and *offset c ≥ fst (bounds m)*
and *offset c + |t|$_\tau$ ≤ snd (bounds m)*
and *¬ is-contiguous-bytes (content m) (nat (offset c)) |t|$_\tau$*
and *is-cap (content m) (nat (offset c))*
and *mc = get-cap (content m) (nat (offset c))*
and *t = Uint8*
and *tagval = the (Mapping.lookup (tags m) (cap-offset (nat (offset c))))*
and *tg = (case perm-cap-load c of False ⇒ False | True ⇒ tagval)*
and *nth-frag = of-nth (the (Mapping.lookup (content m) (nat (offset c))))*
shows *load h c t = Success (Cap-v-frag (mem-capability.extend mc ⦇ tag = tg ⦈) 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|$_\tau$ ≤ base c + len c*

and *offset c* ≥ *base c*
and *offset c mod* |*t*|$_\tau$ = *0*
and *Mapping.lookup* (*heap-map h*) (*block-id c*) = *Some* (*Map m*)
and *offset c* ≥ *fst* (*bounds m*)
and *offset c* + |*t*|$_\tau$ ≤ *snd* (*bounds m*)
and ¬ *is-contiguous-bytes* (*content m*) (*nat* (*offset c*)) |*t*|$_\tau$
and *is-cap* (*content m*) (*nat* (*offset c*))
and *mc* = *get-cap* (*content m*) (*nat* (*offset c*))
and ¬ *is-contiguous-cap* (*content m*) *mc* (*nat* (*offset c*)) |*t*|$_\tau$
and *t* = *Sint8*
and *tagval* = *the* (*Mapping.lookup* (*tags m*) (*cap-offset* (*nat* (*offset c*))))
and *tg* = (*case perm-cap-load c of False* ⇒ *False* | *True* ⇒ *tagval*)
and *nth-frag* = *of-nth* (*the* (*Mapping.lookup* (*content m*) (*nat* (*offset c*))))
shows *load h c t* = *Success* (*Cap-v-frag* (*mem-capability.extend mc* (| *tag* = *tg* |)) *nth-frag*)
unfolding *load-def retrieve-tval-def Let-def*
using *assms*
by (*clarsimp simp add*: *sizeof-def split*: *cctype.split*)

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

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

```
    and ¬ is-contiguous-bytes (content m) (nat (offset c)) |t|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: cctype.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-after-alloc-1:
  assumes alloc h c s = Success (h', cap)
    and |t|τ ≤ s
  shows load h' cap t = Success Undef
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-load cap = True
    using assms(1) alloc-def
    by fastforce
  moreover have offset cap + |t|τ ≤ base cap + len cap
    using assms alloc-def
    by fastforce
  moreover have offset cap ≥ base cap
    using assms alloc-def
    by fastforce
  moreover have offset cap mod |t|τ = 0
    using assms alloc-def
    by fastforce
 moreover have Mapping.lookup (heap-map h') (block-id cap) = Some (Map ?m)
    using assms alloc-def
    by fastforce
  moreover have offset cap ≥ fst (bounds ?m)
    using assms alloc-def
    by fastforce
  moreover have offset cap + |t|τ ≤ snd (bounds ?m)
    using assms alloc-def
    by fastforce
  moreover have ¬ is-contiguous-bytes (content ?m) (nat (offset cap)) |t|τ
  proof −
    have ∃ n. |t|τ = Suc n
      using not0-implies-Suc sizeof-nonzero
      by force
    thus ?thesis
```

```
    using assms alloc-def
    by fastforce
  qed
  moreover have ¬ is-cap (content ?m) (nat (offset cap))
    by simp
  ultimately show ?thesis
    using load-not-cap-in-mem
    by presburger
qed

lemma load-after-alloc-2:
  assumes alloc h c s = Success (h', cap)
    and |t|_τ ≤ s
    and block-id cap ≠ block-id cap'
  shows load h' cap' t = load h cap' t
  using assms unfolding alloc-def load-def
  by force


lemma load-after-alloc-size-fail:
  assumes alloc h c s = Success (h', cap)
    and |t|_τ > 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

lemma load-after-free-1:
  assumes free h c = Success (h', cap)
  shows load h cap t = Error (C2Err TagViolation)
proof −
  consider (null) c = NULL | (non-null) c ≠ NULL by blast
  then show ?thesis
  proof (cases)
    case null
    moreover hence c = cap
```

```
    using assms free-null
    by force
  ultimately show ?thesis
    using load-null-error assms
    by blast
next
  case non-null
  hence cap = c (| tag := False |)
   using assms free-cond(6)[where ?h=h and ?c=c and ?h'=h' and ?cap=cap]

    by presburger
  moreover hence tag cap = False
    using assms
    by force
  ultimately show ?thesis using load-false-tag
    by blast
qed
qed

lemma 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

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*|$_\tau$ = *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*|$_\tau$ ≤ *snd* (*bounds m*)
∧ *is-contiguous-bytes* (*content m*) (*nat* (*offset c*)) |*t*|$_\tau$
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 ∃ *x*. *ret* = *Cap-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*
by (*insert assms load-cond-hard-cap*[**where** *?h*=*h* **and** *?c*=*c* **and** *?t*=*t* **and** *?ret*=*ret*], *clarsimp*,
unfold load-def retrieve-tval-def Let-def, *clarsimp split*: *option.split-asm*,
clarsimp split: *t.split-asm*, *subgoal-tac int* (*fst* (*bounds x2a*)) $\leq$ *int* $|t|_\tau$ * *q* $\wedge$
int $|t|_\tau$ * *q* + *int* $|t|_\tau$ $\leq$ *int* (*snd* (*bounds x2a*)), *clarsimp split*: *cc-type.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 *type-uniq*:
assumes $\exists$ *x n*. *ret* = *Cap-v-frag x n*
shows *ret* $\neq$ *Uint8-v v1 ret* $\neq$ *Sint8-v v2 ret* $\neq$ *Uint16-v v3 ret* $\neq$ *Sint16-v v4*
ret $\neq$ *Uint32-v v5 ret* $\neq$ *Sint32-v v6 ret* $\neq$ *Uint64-v v7 ret* $\neq$ *Sint64-v v8*
ret $\neq$ *Cap-v v9*
using *assms*
by *blast*+

lemma *load-cond-cap-frag*:
assumes *load h c t* = *Success ret*
and $\exists$ *x n*. *ret* = *Cap-v-frag x n*
shows $\exists$ *m mc tagval tg nth-frag*.
Mapping.lookup (*heap-map h*) (*block-id c*) = *Some* (*Map m*) $\wedge$
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)) ∧
                (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
    by (insert assms load-cond-hard-cap[where ?h=h and ?c=c and ?t=t and
?ret=ret], clarsimp,
        unfold 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))+

primrec store-bytes :: (nat, memval) mapping ⇒ nat ⇒ 8 word list ⇒ (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 ⇒ nat ⇒ cap ⇒ nat ⇒ (nat, mem-
val) mapping
  where
  store-cap obj - - 0 = obj
| store-cap obj off cap (Suc n) = store-cap (Mapping.update off (ACap (mem-capability.truncate
cap) n) obj) (Suc off) cap n

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

definition store-tval :: object ⇒ nat ⇒ block ccval ⇒ object
  where
  store-tval obj off val ≡
      case val of Uint8-v  v      ⇒ obj (| content := store-bytes (content obj) off
(encode-u8-list v),
                                    tags := store-tag (tags obj) (cap-offset off) False |)
                | Sint8-v  v      ⇒ obj (| content := store-bytes (content obj) off
(encode-s8-list v),
                                    tags := store-tag (tags obj) (cap-offset off) False |)
                | Uint16-v v      ⇒ obj (| content := store-bytes (content obj) off
(u16-split v),
                                    tags := store-tag (tags obj) (cap-offset off) False |)
                | Sint16-v v      ⇒ obj (| content := store-bytes (content obj) off
(s16-split v),
                                    tags := store-tag (tags obj) (cap-offset off) False |)
                | Uint32-v v      ⇒ obj (| content := store-bytes (content obj) off
```

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

lemma *stored-bytes-prev*:
assumes $x < off$
shows *Mapping.lookup* (*store-bytes obj off vs*) *x* = *Mapping.lookup obj x*
using *assms*
by (*induct vs arbitrary*: *obj off*) *fastforce+*

lemma *stored-cap-prev*:
assumes $x < off$
shows *Mapping.lookup* (*store-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 ⟹ ∃ 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 ⟹ ∃ 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 ⟹ ∃ 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*:
∃ b0 b1. u16-split v = [b0, b1]
using *list-length-2-realise*[**where** *?ls=u16-split v*, *OF u16-split-length*[**where** *?vs=v*]]
by *assumption*

lemma *s16-split-realise*:
∃ 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*:

∃ *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*:
∃ *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*:
∃ *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*:
∃ *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* ∈ *Mapping.keys* (*store-bytes m off* (*u16-split v*))
and *Suc off* ∈ *Mapping.keys* (*store-bytes m off* (*u16-split v*))
and ∃ *b0*. *Mapping.lookup* (*store-bytes m off* (*u16-split v*)) *off* = *Some* (*Byte b0*)
and ∃ *b1*. *Mapping.lookup* (*store-bytes m off* (*u16-split v*)) (*Suc off*) = *Some* (*Byte b1*)
proof −
show *off* ∈ *Mapping.keys* (*store-bytes m off* (*u16-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* ∈ *Mapping.keys* (*store-bytes m off* (*u16-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 ∃ *b0*. *Mapping.lookup* (*store-bytes m off* (*u16-split v*)) *off* = *Some* (*Byte b0*)
by (*metis u16-split-realise stored-bytes-instant-correctness*)
next
show ∃ *b1*. *Mapping.lookup* (*store-bytes m off* (*u16-split v*)) (*Suc off*) = *Some* (*Byte b1*)
by (*metis u16-split-realise store-bytes.simps*(*2*) *stored-bytes-instant-correctness*)
qed

lemma *store-bytes-s16*:
shows *off* ∈ *Mapping.keys* (*store-bytes m off* (*s16-split v*))
and *Suc off* ∈ *Mapping.keys* (*store-bytes m off* (*s16-split v*))
and ∃ *b0*. *Mapping.lookup* (*store-bytes m off* (*s16-split v*)) *off* = *Some* (*Byte b0*)
and ∃ *b1*. *Mapping.lookup* (*store-bytes m off* (*s16-split v*)) (*Suc off*) = *Some* (*Byte b1*)
proof −
show *off* ∈ *Mapping.keys* (*store-bytes m off* (*s16-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* ∈ *Mapping.keys* (*store-bytes m off* (*s16-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 ∃ *b0*. *Mapping.lookup* (*store-bytes m off* (*s16-split v*)) *off* = *Some* (*Byte b0*)
by (*metis s16-split-realise stored-bytes-instant-correctness*)
next
show ∃ *b1*. *Mapping.lookup* (*store-bytes m off* (*s16-split v*)) (*Suc off*) = *Some* (*Byte b1*)
by (*metis s16-split-realise store-bytes.simps*(*2*) *stored-bytes-instant-correctness*)
qed

lemma *store-bytes-u32*:
shows *off* ∈ *Mapping.keys* (*store-bytes m off* (*flatten-u32 v*))
and *Suc off* ∈ *Mapping.keys* (*store-bytes m off* (*flatten-u32 v*))
and *Suc* (*Suc off*) ∈ *Mapping.keys* (*store-bytes m off* (*flatten-u32 v*))
and *Suc* (*Suc* (*Suc off*)) ∈ *Mapping.keys* (*store-bytes m off* (*flatten-u32 v*))
and ∃ *b0*. *Mapping.lookup* (*store-bytes m off* (*flatten-u32 v*)) *off* = *Some* (*Byte b0*)
and ∃ *b1*. *Mapping.lookup* (*store-bytes m off* (*flatten-u32 v*)) (*Suc off*) = *Some* (*Byte b1*)
and ∃ *b2*. *Mapping.lookup* (*store-bytes m off* (*flatten-u32 v*)) (*Suc* (*Suc off*)) = *Some* (*Byte b2*)
and ∃ *b3*. *Mapping.lookup* (*store-bytes m off* (*flatten-u32 v*)) (*Suc* (*Suc* (*Suc off*))) = *Some* (*Byte b3*)
proof −
show *off* ∈ *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* ∈ *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*) ∈ *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*)) ∈ *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 ∃ *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 ∃ *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 ∃ *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 ∃ *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* ∈ *Mapping.keys* (*store-bytes m off* (*flatten-u64 v*))
and *Suc off* ∈ *Mapping.keys* (*store-bytes m off* (*flatten-u64 v*))
and *Suc* (*Suc off*) ∈ *Mapping.keys* (*store-bytes m off* (*flatten-u64 v*))
and *Suc* (*Suc* (*Suc off*)) ∈ *Mapping.keys* (*store-bytes m off* (*flatten-u64 v*))
and *Suc* (*Suc* (*Suc* (*Suc off*))) ∈ *Mapping.keys* (*store-bytes m off* (*flatten-u64 v*))
and *Suc* (*Suc* (*Suc* (*Suc* (*Suc off*)))) ∈ *Mapping.keys* (*store-bytes m off* (*flatten-u64 v*))
and *Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc off*))))) ∈ *Mapping.keys* (*store-bytes m off* (*flatten-u64 v*))
and ∃ *b0*. *Mapping.lookup* (*store-bytes m off* (*flatten-u64 v*)) *off* = *Some* (*Byte b0*)
and ∃ *b1*. *Mapping.lookup* (*store-bytes m off* (*flatten-u64 v*)) (*Suc off*) = *Some* (*Byte b1*)
and ∃ *b2*. *Mapping.lookup* (*store-bytes m off* (*flatten-u64 v*)) (*Suc* (*Suc off*)) = *Some* (*Byte b2*)

and ∃ *b3*. *Mapping.lookup* (*store-bytes m off* (*flatten-u64 v*)) (*Suc* (*Suc* (*Suc off*))) = *Some* (*Byte b3*)
and ∃ *b0*. *Mapping.lookup* (*store-bytes m off* (*flatten-u64 v*)) (*Suc* (*Suc* (*Suc* (*Suc off*)))) = *Some* (*Byte b0*)
and ∃ *b1*. *Mapping.lookup* (*store-bytes m off* (*flatten-u64 v*)) (*Suc* (*Suc* (*Suc* (*Suc* (*Suc off*))))) = *Some* (*Byte b1*)
and ∃ *b2*. *Mapping.lookup* (*store-bytes m off* (*flatten-u64 v*)) (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc off*)))))) = *Some* (*Byte b2*)
and ∃ *b3*. *Mapping.lookup* (*store-bytes m off* (*flatten-u64 v*)) (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc off*))))))) = *Some* (*Byte b3*)
proof −
show *off* ∈ *Mapping.keys* (*store-bytes m off* (*flatten-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 off* ∈ *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 off*) ∈ *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 off*)) ∈ *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 off*))) ∈ *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*)))) ∈ *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*))))) ∈ *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*)))))) ∈ *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 ∃ *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 ∃ *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 ∃ *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 ∃ *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 ∃ *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∃ *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
show ∃ *b2*. *Mapping.lookup* (*store-bytes m off* (*flatten-u64 v*)) (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc off*)))))) = *Some* (*Byte b2*)
by (*metis u64-split-realise store-bytes.simps*(*2*) *stored-bytes-instant-correctness*)
next
show ∃ *b3*. *Mapping.lookup* (*store-bytes m off* (*flatten-u64 v*)) (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc off*))))))) = *Some* (*Byte b3*)
by (*metis u64-split-realise store-bytes.simps*(*2*) *stored-bytes-instant-correctness*)
qed

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

and *Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc off*)))))) ∈ *Mapping.keys* (*store-bytes m off* (*flatten-s64 v*))
and ∃ *b0*. *Mapping.lookup* (*store-bytes m off* (*flatten-s64 v*)) *off* = *Some* (*Byte b0*)
and ∃ *b1*. *Mapping.lookup* (*store-bytes m off* (*flatten-s64 v*)) (*Suc off*) = *Some* (*Byte b1*)
and ∃ *b2*. *Mapping.lookup* (*store-bytes m off* (*flatten-s64 v*)) (*Suc* (*Suc off*)) = *Some* (*Byte b2*)
and ∃ *b3*. *Mapping.lookup* (*store-bytes m off* (*flatten-s64 v*)) (*Suc* (*Suc* (*Suc off*))) = *Some* (*Byte b3*)
and ∃ *b0*. *Mapping.lookup* (*store-bytes m off* (*flatten-s64 v*)) (*Suc* (*Suc* (*Suc* (*Suc off*)))) = *Some* (*Byte b0*)
and ∃ *b1*. *Mapping.lookup* (*store-bytes m off* (*flatten-s64 v*)) (*Suc* (*Suc* (*Suc* (*Suc* (*Suc off*))))) = *Some* (*Byte b1*)
and ∃ *b2*. *Mapping.lookup* (*store-bytes m off* (*flatten-s64 v*)) (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc off*)))))) = *Some* (*Byte b2*)
and ∃ *b3*. *Mapping.lookup* (*store-bytes m off* (*flatten-s64 v*)) (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc off*))))))) = *Some* (*Byte b3*)
proof −
show *off* ∈ *Mapping.keys* (*store-bytes m off* (*flatten-s64 v*))
by (*metis* (*no-types*, *opaque-lifting*) *domIff handy-if-lemma keys-dom-lookup stored-bytes-instant-correctness s64-split-realise*)
next
show *Suc off* ∈ *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 off*) ∈ *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 off*)) ∈ *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*))) ∈ *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*)))) ∈ *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)))) ∈ 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))))) ∈ 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 ∃ *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 ∃ *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 ∃ *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 ∃ *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 ∃ *b0*. *Mapping.lookup (store-bytes m off (flatten-s64 v)) (Suc (Suc (Suc (Suc off)))) = Some (Byte b0)*
by (*metis s64-split-realise store-bytes.simps*(*2*) *stored-bytes-instant-correctness*)
next
show∃ *b1*. *Mapping.lookup (store-bytes m off (flatten-s64 v)) (Suc (Suc (Suc (Suc (Suc off))))) = Some (Byte b1)*
by (*metis s64-split-realise store-bytes.simps*(*2*) *stored-bytes-instant-correctness*)
next
show ∃ *b2*. *Mapping.lookup (store-bytes m off (flatten-s64 v)) (Suc (Suc (Suc (Suc (Suc (Suc off)))))) = Some (Byte b2)*
by (*metis s64-split-realise store-bytes.simps*(*2*) *stored-bytes-instant-correctness*)
next
show ∃ *b3*. *Mapping.lookup (store-bytes m off (flatten-s64 v)) (Suc (Suc (Suc (Suc (Suc (Suc (Suc off))))))) = 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* ≠ *Undef*
and ∀ *v*. *val* ≠ *Cap-v v*
and ∀ *v n*. *val* ≠ *Cap-v-frag v n*
shows *is-contiguous-bytes* (*content* (*store-tval obj off val*)) *off* |*memval-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-bytes u64-store-bytes-imp-is-contiguous-bytes)

lemma *suc-of-32*:
32 = *Suc 31*
by *simp*

lemma *store-cap-correct-dom*:
shows *off* ∈ *Mapping.keys* (*store-cap m off cap 32*)
and *off + 1* ∈ *Mapping.keys* (*store-cap m off cap 32*)
and *off + 2* ∈ *Mapping.keys* (*store-cap m off cap 32*)
and *off + 3* ∈ *Mapping.keys* (*store-cap m off cap 32*)
and *off + 4* ∈ *Mapping.keys* (*store-cap m off cap 32*)
and *off + 5* ∈ *Mapping.keys* (*store-cap m off cap 32*)
and *off + 6* ∈ *Mapping.keys* (*store-cap m off cap 32*)
and *off + 7* ∈ *Mapping.keys* (*store-cap m off cap 32*)
and *off + 8* ∈ *Mapping.keys* (*store-cap m off cap 32*)
and *off + 9* ∈ *Mapping.keys* (*store-cap m off cap 32*)
and *off + 10* ∈ *Mapping.keys* (*store-cap m off cap 32*)
and *off + 11* ∈ *Mapping.keys* (*store-cap m off cap 32*)
and *off + 12* ∈ *Mapping.keys* (*store-cap m off cap 32*)
and *off + 13* ∈ *Mapping.keys* (*store-cap m off cap 32*)
and *off + 14* ∈ *Mapping.keys* (*store-cap m off cap 32*)
and *off + 15* ∈ *Mapping.keys* (*store-cap m off cap 32*)
and *off + 16* ∈ *Mapping.keys* (*store-cap m off cap 32*)
and *off + 17* ∈ *Mapping.keys* (*store-cap m off cap 32*)
and *off + 18* ∈ *Mapping.keys* (*store-cap m off cap 32*)
and *off + 19* ∈ *Mapping.keys* (*store-cap m off cap 32*)
and *off + 20* ∈ *Mapping.keys* (*store-cap m off cap 32*)
and *off + 21* ∈ *Mapping.keys* (*store-cap m off cap 32*)
and *off + 22* ∈ *Mapping.keys* (*store-cap m off cap 32*)
and *off + 23* ∈ *Mapping.keys* (*store-cap m off cap 32*)
and *off + 24* ∈ *Mapping.keys* (*store-cap m off cap 32*)
and *off + 25* ∈ *Mapping.keys* (*store-cap m off cap 32*)
and *off + 26* ∈ *Mapping.keys* (*store-cap m off cap 32*)
and *off + 27* ∈ *Mapping.keys* (*store-cap m off cap 32*)
and *off + 28* ∈ *Mapping.keys* (*store-cap m off cap 32*)
and *off + 29* ∈ *Mapping.keys* (*store-cap m off cap 32*)
and *off + 30* ∈ *Mapping.keys* (*store-cap m off cap 32*)
and *off + 31* ∈ *Mapping.keys* (*store-cap m off cap 32*)
proof − **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*
by (*simp add*: *eval-nat-numeral*(*3*) *numeral-Bit0*, *insert memval-byte-not-memcap*, *blast*)

lemma *stored-tval-is-cap*:
assumes ∃ *v*. *val* = *Cap-v v*
shows *is-cap* (*content* (*store-tval obj off val*)) *off*
apply (*simp add*: *assms store-tval-def split*: *ccval.split*)
apply (*safe*; ((*insert assms*, *blast*)+)*?*)
apply (*metis domIff keys-dom-lookup less-imp-Suc-add option.discI sizeof-nonzero*

stored-cap-instant-correctness)
apply (*metis memval-byte-not-memcap not0-implies-Suc not-less-eq option.sel sizeof-nonzero*
stored-cap-instant-correctness zero-less-Suc)
apply (*simp add*: *sizeof-def store-cap-correct-dom*(*1*))
apply (*metis Some-to-the cctype.simps*(*81*) *memval-byte-not-memcap sizeof-def store-cap-correct-val*(*1*))
done

lemma *stored-tval-contiguous-cap*:
assumes *val* = *Cap-v cap*
shows *is-contiguous-cap* (*content* (*store-tval obj off val*)) (*mem-capability.truncate cap*) *off* |*memval-type val*|$_\tau$
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 x3*))) *off* |*Uint16*|$_\tau$) = *x3*
apply (*clarsimp simp add*: *sizeof-def store-tval-def eval-nat-numeral*(*3*))
apply (*clarsimp simp add*: *numeral-Bit0*)
apply (*subgoal-tac of-byte* (*the* (*Mapping.lookup* (*store-bytes* (*content obj*) *off* (*u16-split x3*)) *off*)) = (*u16-split x3*) ! *0* ∧
of-byte (*the* (*Mapping.lookup* (*store-bytes* (*content obj*) *off* (*u16-split x3*)) (*Suc off*))) = (*u16-split x3*) ! *1*)
apply (*metis cat-flatten-u16-eq list-length-2-realise memval.sel*(*1*) *option.sel store-bytes.simps*(*2*)
stored-bytes-instant-correctness u16-split-length)
apply *safe*
apply (*smt* (*verit*, *best*) *Some-to-the length-nth-simps*(*3*) *memval.sel*(*1*)
stored-bytes-instant-correctness u16-split-realise)

apply (*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*)
done

lemma *decode-encoded-s16-in-mem*:
cat-s16 (*retrieve-bytes* (*content* (*store-tval obj off* (*Sint16-v x4*))) *off* |*Sint16*|$_\tau$) = *x4*
apply (*clarsimp simp add*: *sizeof-def store-tval-def eval-nat-numeral*(*3*))
apply (*clarsimp simp add*: *numeral-Bit0*)
apply (*subgoal-tac of-byte* (*the* (*Mapping.lookup* (*store-bytes* (*content obj*) *off* (*s16-split x4*)) *off*)) = (*s16-split x4*) ! *0* ∧
of-byte (*the* (*Mapping.lookup* (*store-bytes* (*content obj*) *off* (*s16-split x4*)) (*Suc off*))) = (*s16-split x4*) ! *1*)
apply (*metis cat-flatten-s16-eq list-length-2-realise memval.sel*(*1*) *option.sel store-bytes.simps*(*2*)
stored-bytes-instant-correctness flatten-s16-length)
apply *safe*
apply (*smt* (*verit*, *best*) *Some-to-the length-nth-simps*(*3*) *memval.sel*(*1*)
stored-bytes-instant-correctness s16-split-realise)
apply (*metis One-nat-def flatten-s16-length list-length-2-realise memval.sel*(*1*) *nth-Cons-0*
nth-Cons-Suc option.sel store-bytes.simps(*2*) *stored-bytes-instant-correctness*)
done

lemma *decode-encoded-u32-in-mem*:
cat-u32 (*retrieve-bytes* (*content* (*store-tval obj off* (*Uint32-v x5*))) *off* |*Uint32*|$_\tau$) = *x5*
apply (*clarsimp simp add*: *sizeof-def store-tval-def eval-nat-numeral*(*3*))
apply (*clarsimp simp add*: *numeral-Bit0*)
apply (*subgoal-tac of-byte* (*the* (*Mapping.lookup* (*store-bytes* (*content obj*) *off* (*flatten-u32 x5*)) *off*)) = (*flatten-u32 x5*) ! *0* ∧
of-byte (*the* (*Mapping.lookup* (*store-bytes* (*content obj*) *off* (*flatten-u32 x5*)) (*Suc off*))) = (*flatten-u32 x5*) ! *1* ∧
of-byte (*the* (*Mapping.lookup* (*store-bytes* (*content obj*) *off* (*flatten-u32 x5*)) (*Suc* (*Suc off*)))) = (*flatten-u32 x5*) ! *2* ∧
of-byte (*the* (*Mapping.lookup* (*store-bytes* (*content obj*) *off* (*flatten-u32 x5*)) (*Suc* (*Suc* (*Suc off*))))) = (*flatten-u32 x5*) ! *3*)
apply (*smt* (*verit*, *del-insts*) *One-nat-def Suc-1 eval-nat-numeral*(*3*) *length-nth-simps*(*3*)
length-nth-simps(*4*) *u32-split-realise word-rcat-rsplit*)
apply *safe*
apply (*metis length-nth-simps*(*3*) *memval.sel*(*1*) *option.sel stored-bytes-instant-correctness*
u32-split-realise)
apply (*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*)
apply (*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 u32-split-realise*)
apply (*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*)
done

lemma *decode-encoded-s32-in-mem*:
cat-s32 (*retrieve-bytes* (*content* (*store-tval obj off* (*Sint32-v x6*))) *off* |*Sint32*|$_\tau$)
= *x6*
apply (*clarsimp simp add*: *sizeof-def store-tval-def eval-nat-numeral*(*3*))
apply (*clarsimp simp add*: *numeral-Bit0*)
apply (*subgoal-tac of-byte* (*the* (*Mapping.lookup* (*store-bytes* (*content obj*) *off*
(*flatten-s32 x6*)) *off*)) = (*flatten-s32 x6*) ! *0* ∧
of-byte (*the* (*Mapping.lookup* (*store-bytes* (*content obj*) *off*
(*flatten-s32 x6*)) (*Suc off*))) = (*flatten-s32 x6*) ! *1* ∧
of-byte (*the* (*Mapping.lookup* (*store-bytes* (*content obj*) *off*
(*flatten-s32 x6*)) (*Suc* (*Suc off*)))) = (*flatten-s32 x6*) ! *2* ∧
of-byte (*the* (*Mapping.lookup* (*store-bytes* (*content obj*) *off*
(*flatten-s32 x6*)) (*Suc* (*Suc* (*Suc off*))))) = (*flatten-s32 x6*) ! *3*)
apply (*smt* (*verit*, *del-insts*) *One-nat-def Suc-1 eval-nat-numeral*(*3*) *length-nth-simps*(*3*)
length-nth-simps(*4*) *s32-split-realise word-rcat-rsplit*)
apply *safe*
apply (*metis length-nth-simps*(*3*) *memval.sel*(*1*) *option.sel stored-bytes-instant-correctness*
s32-split-realise)
apply (*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*)
apply (*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*)
apply (*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*)
done

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*
apply *clarsimp*
apply (*insert u64-split-realise*[**where** *?v*=*vs*])
apply *safe*
apply (*smt* (*verit*, *best*) *One-nat-def Suc-1 add.commute add-Suc-right eval-nat-numeral*(*3*)
length-nth-simps(*3*) *length-nth-simps*(*4*) *numeral-4-eq-4 numeral-Bit0 word-rcat-rsplit*)

done

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*
apply *clarsimp*
apply (*insert s64-split-realise*[**where** *?v*=*vs*])
apply *safe*
apply (*smt* (*verit*, *best*) *One-nat-def Suc-1 add.commute add-Suc-right eval-nat-numeral*(*3*)
length-nth-simps(*3*) *length-nth-simps*(*4*) *numeral-4-eq-4 numeral-Bit0 word-rcat-rsplit*)
done

lemma *decode-encoded-u64-in-mem*:
cat-u64 (*retrieve-bytes* (*content* (*store-tval obj off* (*Uint64-v x7*))) *off* |*Uint64*|$_\tau$) = *x7*
apply (*clarsimp simp add*: *sizeof-def store-tval-def eval-nat-numeral*(*3*))
apply (*clarsimp simp add*: *numeral-Bit0*)
apply (*subgoal-tac of-byte* (*the* (*Mapping.lookup* (*store-bytes* (*content obj*) *off* (*flatten-u64 x7*)) *off*)) = (*flatten-u64 x7*) ! *0* ∧
of-byte (*the* (*Mapping.lookup* (*store-bytes* (*content obj*) *off* (*flatten-u64 x7*)) (*Suc off*))) = (*flatten-u64 x7*) ! *1* ∧
of-byte (*the* (*Mapping.lookup* (*store-bytes* (*content obj*) *off* (*flatten-u64 x7*)) (*Suc* (*Suc off*)))) = (*flatten-u64 x7*) ! *2* ∧
of-byte (*the* (*Mapping.lookup* (*store-bytes* (*content obj*) *off* (*flatten-u64 x7*)) (*Suc* (*Suc* (*Suc off*))))) = (*flatten-u64 x7*) ! *3* ∧
of-byte (*the* (*Mapping.lookup* (*store-bytes* (*content obj*) *off* (*flatten-u64 x7*)) (*Suc* (*Suc* (*Suc* (*Suc off*)))))) = (*flatten-u64 x7*) ! *4* ∧
of-byte (*the* (*Mapping.lookup* (*store-bytes* (*content obj*) *off* (*flatten-u64 x7*)) (*Suc* (*Suc* (*Suc* (*Suc* (*Suc off*))))))) = (*flatten-u64 x7*) ! *5* ∧
of-byte (*the* (*Mapping.lookup* (*store-bytes* (*content obj*) *off* (*flatten-u64 x7*)) (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc off*)))))))) = (*flatten-u64 x7*) ! *6* ∧
of-byte (*the* (*Mapping.lookup* (*store-bytes* (*content obj*) *off* (*flatten-u64 x7*)) (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc off*))))))))) = (*flatten-u64 x7*) ! *7*)
apply (*presburger add*: *cat-flatten-u64-contents-eq*)
apply (*smt* (*verit*, *best*) *length-nth-simps*(*3*) *length-nth-simps*(*4*) *memval.sel*(*1*) *option.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*)
done

lemma *decode-encoded-s64-in-mem*:
cat-s64 (*retrieve-bytes* (*content* (*store-tval obj off* (*Sint64-v x8*))) *off* |*Sint64*|$_\tau$)

```
= x8
  apply (clarsimp simp add: sizeof-def store-tval-def eval-nat-numeral(3))
  apply (clarsimp simp add: numeral-Bit0)
   apply (subgoal-tac of-byte (the (Mapping.lookup (store-bytes (content obj) off
(flatten-s64 x8)) off)) = (flatten-s64 x8) ! 0 ∧
                              of-byte (the (Mapping.lookup (store-bytes (content obj) off
(flatten-s64 x8)) (Suc off))) = (flatten-s64 x8) ! 1 ∧
                              of-byte (the (Mapping.lookup (store-bytes (content obj) off
(flatten-s64 x8)) (Suc (Suc off)))) = (flatten-s64 x8) ! 2 ∧
                              of-byte (the (Mapping.lookup (store-bytes (content obj) off
(flatten-s64 x8)) (Suc (Suc (Suc off))))) = (flatten-s64 x8) ! 3 ∧
                              of-byte (the (Mapping.lookup (store-bytes (content obj) off
(flatten-s64 x8)) (Suc (Suc (Suc (Suc off)))))) = (flatten-s64 x8) ! 4 ∧
                              of-byte (the (Mapping.lookup (store-bytes (content obj) off
(flatten-s64 x8)) (Suc (Suc (Suc (Suc (Suc off))))))) = (flatten-s64 x8) ! 5 ∧
                              of-byte (the (Mapping.lookup (store-bytes (content obj) off
(flatten-s64 x8)) (Suc (Suc (Suc (Suc (Suc (Suc off)))))))) = (flatten-s64 x8) ! 6
∧
                              of-byte (the (Mapping.lookup (store-bytes (content obj) off
(flatten-s64 x8)) (Suc (Suc (Suc (Suc (Suc (Suc (Suc off))))))))) = (flatten-s64
x8) ! 7)
   apply (presburger add: cat-flatten-s64-contents-eq)
  apply (smt (verit, best) length-nth-simps(3) length-nth-simps(4) memval.sel(1)
option.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)
  done

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)
   apply (unfold retrieve-tval-def; clarsimp simp add: stored-tval-contiguous-cap;
safe)
                  apply (metis is-contiguous-bytes.simps(2) less-numeral-extra(3)
not0-implies-Suc sizeof-nonzero)
                   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|τ)
                  apply clarsimp
                  apply (unfold store-tval-def get-cap-def sizeof-def)[1]
                  apply clarsimp
                  apply (subst suc-of-32)
                  apply (simp only: stored-cap-instant-correctness)
                  apply simp
              apply (unfold mem-capability.extend-def mem-capability.truncate-def,
clarsimp)[1]
              apply (metis cctype.simps(81) get-cap-def is-contiguous-cap.simps(2)
```

memval-size-cap sizeof-def
stored-tval-contiguous-cap suc-of-32)
apply (*insert stored-tval-is-cap*, *force*)[*1*]
apply (*insert stored-tval-is-cap*, *force*)[*1*]
apply (*insert stored-tval-is-cap*, *force*)[*1*]
apply (*insert stored-tval-is-cap*, *force*)[*1*]
apply (*metis is-contiguous-bytes.simps*(*2*) *less-numeral-extra*(*3*) *not0-implies-Suc sizeof-nonzero*)
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 (*unfold store-tval-def get-cap-def sizeof-def*)[*1*]
apply *clarsimp*
apply (*subst suc-of-32*)
apply (*simp only*: *stored-cap-instant-correctness*)
apply *simp*
apply (*unfold mem-capability.extend-def mem-capability.truncate-def*, *clarsimp*)[*1*]
apply (*metis cctype.simps*(*81*) *get-cap-def is-contiguous-cap.simps*(*2*) *memval-size-cap sizeof-def*
stored-tval-contiguous-cap suc-of-32)
apply (*insert stored-tval-is-cap*, *force*)[*1*]
apply (*insert stored-tval-is-cap*, *force*)[*1*]
apply (*metis cctype.simps*(*81*) *is-contiguous-bytes.simps*(*2*) *sizeof-def suc-of-32*)
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 (*unfold store-tval-def get-cap-def sizeof-def*)[*1*]
apply *clarsimp*
apply (*subst suc-of-32*)
apply (*simp only*: *stored-cap-instant-correctness*)
apply *simp*
apply (*unfold mem-capability.extend-def mem-capability.truncate-def*, *clarsimp*)[*1*]
apply (*metis cctype.simps*(*81*) *get-cap-def is-contiguous-cap.simps*(*2*) *memval-size-cap sizeof-def*
stored-tval-contiguous-cap suc-of-32)
apply (*insert stored-tval-is-cap*, *force*)[*1*]
apply (*insert stored-tval-is-cap*, *force*)[*1*]
apply (*insert stored-tval-is-cap*, *force*)[*1*]
apply (*insert stored-tval-is-cap*, *force*)[*1*]
apply (*metis gr-implies-not-zero is-contiguous-bytes.simps*(*2*) *old.nat.exhaust sizeof-nonzero*)
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 (*unfold store-tval-def get-cap-def sizeof-def*)[*1*]
apply *clarsimp*
apply (*subst suc-of-32*)
apply (*simp only*: *stored-cap-instant-correctness*)
apply *simp*
apply (*unfold mem-capability.extend-def mem-capability.truncate-def*, *clarsimp*)[*1*]
apply (*metis cctype.simps*(*81*) *get-cap-def is-contiguous-cap.simps*(*2*) *memval-size-cap sizeof-def*
stored-tval-contiguous-cap suc-of-32)
apply (*insert stored-tval-is-cap*, *force*)[*1*]
apply (*insert stored-tval-is-cap*, *force*)[*1*]
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*)
apply (*unfold retrieve-tval-def*; *clarsimp simp add*: *stored-tval-contiguous-cap*; *safe*)
apply (*metis is-contiguous-bytes.simps*(*2*) *less-numeral-extra*(*3*) *not0-implies-Suc sizeof-nonzero*)
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 (*unfold store-tval-def get-cap-def sizeof-def*)[*1*]
apply *clarsimp*
apply (*subst suc-of-32*)
apply (*simp only*: *stored-cap-instant-correctness*)
apply *simp*
apply (*unfold mem-capability.extend-def mem-capability.truncate-def*, *clarsimp*)[*1*]
apply (*metis cctype.simps*(*81*) *get-cap-def is-contiguous-cap.simps*(*2*) *memval-size-cap sizeof-def*
stored-tval-contiguous-cap suc-of-32)
apply (*insert stored-tval-is-cap*, *force*)[*1*]
apply (*insert stored-tval-is-cap*, *force*)[*1*]
apply (*insert stored-tval-is-cap*, *force*)[*1*]
apply (*insert stored-tval-is-cap*, *force*)[*1*]
apply (*metis is-contiguous-bytes.simps*(*2*) *less-numeral-extra*(*3*) *not0-implies-Suc sizeof-nonzero*)
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 (*unfold store-tval-def get-cap-def sizeof-def*)[*1*]
apply *clarsimp*
apply (*subst suc-of-32*)

apply (*simp only*: *stored-cap-instant-correctness*)
apply *simp*
apply (*unfold mem-capability.extend-def mem-capability.truncate-def*, *clar-simp*)[*1*]
apply (*metis cctype.simps*(*81*) *get-cap-def is-contiguous-cap.simps*(*2*) *memval-size-cap sizeof-def*
stored-tval-contiguous-cap suc-of-32)
apply (*insert stored-tval-is-cap*, *force*)[*1*]
apply (*insert stored-tval-is-cap*, *force*)[*1*]
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*
apply (*clarsimp simp add*: *assms*)
apply (*unfold retrieve-tval-def*; *clarsimp simp add*: *stored-tval-contiguous-bytes*; *safe*)
apply (*simp add*: *sizeof-def*)
apply (*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*)
apply (*simp add*: *sizeof-def*)
apply (*metis cctype.simps*(*73*) *ccval.distinct*(*15*) *ccval.distinct*(*17*) *ccval.distinct*(*19*)
memval-size-u8 sizeof-def stored-tval-contiguous-bytes)
apply (*clarsimp simp add*: *sizeof-def store-tval-def*)
apply (*metis cctype.simps*(*73*) *ccval.distinct*(*15*) *ccval.distinct*(*17*) *ccval.distinct*(*19*)
memval-size-u8 sizeof-def stored-tval-contiguous-bytes)
done

lemma *retrieve-stored-tval-s8*:
assumes *val* = *Sint8-v v*
shows *retrieve-tval* (*store-tval obj off val*) *off* (*memval-type val*) *b* = *val*
apply (*clarsimp simp add*: *assms*)
apply (*unfold retrieve-tval-def*; *clarsimp simp add*: *stored-tval-contiguous-bytes*; *safe*)
apply (*simp add*: *sizeof-def*)
apply (*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*)
apply (*simp add*: *sizeof-def*)
apply (*metis cctype.simps*(*74*) *ccval.distinct*(*33*) *ccval.distinct*(*35*) *ccval.distinct*(*37*)
memval-size-s8 sizeof-def stored-tval-contiguous-bytes)
apply (*clarsimp simp add*: *sizeof-def store-tval-def*)
apply (*metis cctype.simps*(*74*) *ccval.distinct*(*33*) *ccval.distinct*(*35*) *ccval.distinct*(*37*)
memval-size-s8 sizeof-def stored-tval-contiguous-bytes)

done

lemma *retrieve-stored-tval-u16*:
assumes *val* = *Uint16-v v*
shows *retrieve-tval* (*store-tval obj off val*) *off* (*memval-type val*) *b* = *val*
apply (*clarsimp simp add*: *assms*)
apply (*unfold retrieve-tval-def*; *clarsimp simp add*: *stored-tval-contiguous-bytes*;
safe)
apply (*presburger add*: *decode-encoded-u16-in-mem*)
apply (*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*)
apply (*presburger add*: *decode-encoded-u16-in-mem*)
apply (*metis Suc-1 ccval.distinct*(*49*) *ccval.distinct*(*51*) *ccval.distinct*(*53*)
is-contiguous-bytes.simps(*2*) *memval-size-u16 stored-tval-contiguous-bytes*)
apply (*presburger add*: *decode-encoded-u16-in-mem*)
apply (*metis cctype.simps*(*75*) *ccval.distinct*(*49*) *ccval.distinct*(*51*) *ccval.distinct*(*53*)

memval-size-u16 sizeof-def stored-tval-contiguous-bytes)
done

lemma *retrieve-stored-tval-s16*:
assumes *val* = *Sint16-v v*
shows *retrieve-tval* (*store-tval obj off val*) *off* (*memval-type val*) *b* = *val*
apply (*clarsimp simp add*: *assms*)
apply (*unfold retrieve-tval-def*; *clarsimp simp add*: *stored-tval-contiguous-bytes*;
safe)
apply (*presburger add*: *decode-encoded-s16-in-mem*)
apply (*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)
apply (*presburger add*: *decode-encoded-s16-in-mem*)
apply (*metis Suc-1 ccval.distinct*(*63*) *ccval.distinct*(*65*) *ccval.distinct*(*67*)
is-contiguous-bytes.simps(*2*) *memval-size-s16 stored-tval-contiguous-bytes*)
apply (*presburger add*: *decode-encoded-s16-in-mem*)
apply (*metis cctype.simps*(*76*) *ccval.distinct*(*63*) *ccval.distinct*(*65*) *ccval.distinct*(*67*)

memval-size-s16 sizeof-def stored-tval-contiguous-bytes)
done

lemma *retrieve-stored-tval-u32*:
assumes *val* = *Uint32-v v*
shows *retrieve-tval* (*store-tval obj off val*) *off* (*memval-type val*) *b* = *val*
apply (*clarsimp simp add*: *assms*)
apply (*unfold retrieve-tval-def*; *clarsimp simp add*: *stored-tval-contiguous-bytes*;
safe)
apply (*presburger add*: *decode-encoded-u32-in-mem*)
apply (*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)

apply (*presburger add*: *decode-encoded-u32-in-mem*)
apply (*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)
apply (*presburger add*: *decode-encoded-u32-in-mem*)
apply (*metis cctype.simps*(*77*) *ccval.distinct*(*75*) *ccval.distinct*(*77*) *ccval.distinct*(*79*)

memval-size-u32 sizeof-def stored-tval-contiguous-bytes)
done

lemma *retrieve-stored-tval-s32*:
assumes *val* = *Sint32-v v*
shows *retrieve-tval* (*store-tval obj off val*) *off* (*memval-type val*) *b* = *val*
apply (*clarsimp simp add*: *assms*)
apply (*unfold retrieve-tval-def*; *clarsimp simp add*: *stored-tval-contiguous-bytes*;
safe)
apply (*presburger add*: *decode-encoded-s32-in-mem*)
apply (*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)
apply (*presburger add*: *decode-encoded-s32-in-mem*)
apply (*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*)

apply (*presburger add*: *decode-encoded-s32-in-mem*)
apply (*metis cctype.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)
done

lemma *retrieve-stored-tval-u64*:
assumes *val* = *Uint64-v v*
shows *retrieve-tval* (*store-tval obj off val*) *off* (*memval-type val*) *b* = *val*
apply (*clarsimp simp add*: *assms*)
apply (*unfold retrieve-tval-def*; *clarsimp simp add*: *stored-tval-contiguous-bytes*;
safe)
apply (*presburger add*: *decode-encoded-u64-in-mem*)
apply (*metis cctype.simps*(*79*) *ccval.distinct*(*93*) *ccval.distinct*(*95*) *ccval.distinct*(*97*)

memval-size-u64 sizeof-def stored-tval-contiguous-bytes)
apply (*presburger add*: *decode-encoded-u64-in-mem*)
apply (*metis cctype.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)
apply (*presburger add*: *decode-encoded-u64-in-mem*)
apply (*metis cctype.simps*(*79*) *ccval.distinct*(*93*) *ccval.distinct*(*95*) *ccval.distinct*(*97*)

memval-size-u64 sizeof-def stored-tval-contiguous-bytes)

done

lemma *retrieve-stored-tval-s64*:
assumes *val* = *Sint64-v v*
shows *retrieve-tval* (*store-tval obj off val*) *off* (*memval-type val*) *b* = *val*
apply (*clarsimp simp add*: *assms*)
apply (*unfold retrieve-tval-def*; *clarsimp simp add*: *stored-tval-contiguous-bytes*; *safe*)
apply (*presburger add*: *decode-encoded-s64-in-mem*)
apply (*metis cctype.simps*(*80*) *ccval.distinct*(*101*) *ccval.distinct*(*103*) *ccval.distinct*(*99*)
memval-size-s64 sizeof-def stored-tval-contiguous-bytes)
apply (*presburger add*: *decode-encoded-s64-in-mem*)
apply (*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*)

apply (*presburger add*: *decode-encoded-s64-in-mem*)
apply (*metis cctype.simps*(*80*) *ccval.distinct*(*101*) *ccval.distinct*(*103*) *ccval.distinct*(*99*)
memval-size-s64 sizeof-def stored-tval-contiguous-bytes)
done

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 ∀ *v*. *val* ≠ *Cap-v v*
and ∀ *v n*. *val* ≠ *Cap-v-frag v n*
shows *retrieve-tval* (*store-tval obj off val*) *off* (*memval-type val*) *b* = *val*
by (*clarsimp simp add*: *assms split*: *ccval.split*)
(*insert retrieve-stored-tval-prim*[**where** *?obj*=*obj* **and** *?off*=*off* **and** *?val*=*val* **and** *?b*=*b*], *fastforce*)

lemma *retrieve-stored-tval-with-perm-cap-load*:
assumes *val* ≠ *Undef*
and ∀ *v n*. *val* ≠ *Cap-v-frag v n*
shows *retrieve-tval* (*store-tval obj off val*) *off* (*memval-type val*) *True* = *val*
by (*clarsimp simp add*: *assms split*: *ccval.split*)
(*insert 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*], *simp*)

definition *store* :: *heap* ⇒ *cap* ⇒ *block ccval* ⇒ *heap result*
where
store h c v ≡
if tag c = *False then*
Error (*C2Err TagViolation*)
else if perm-store c = *False then*
Error (*C2Err PermitStoreViolation*)
else if (*case v of Cap-v cv* ⇒ ¬ *perm-cap-store c* ∧ *tag cv* | - ⇒ *False*) *then*
Error (*C2Err PermitStoreCapViolation*)
else if (*case v of Cap-v cv* ⇒ ¬ *perm-cap-store-local c* ∧ *tag cv* ∧ ¬ *perm-global cv* | - ⇒ *False*) *then*
Error (*C2Err PermitStoreLocalCapViolation*)
else if offset c + |*memval-type v*|$_\tau$ > *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*|$_\tau$ ≠ *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) |)))
```

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 *tag c = True*
    and *perm-store c = True*
    and *perm-cap-store c = True*
    and *perm-cap-store-local c = False*
    and ∃ *cv*. *v = Cap-v cv* ∧ *tag cv = True* ∧ *perm-global cv = False*
  shows *store h c v = Error (C2Err PermitStoreLocalCapViolation)*
  unfolding *store-def*
  using *assms*
  by *force*

lemma *store-bound-over*:

assumes *tag c = True*
and *perm-store c = True*
and $\bigwedge$ *cv*. ⟦ *v = Cap-v cv*; *tag cv* ⟧ ⟹ *perm-cap-store c* ∧ (*perm-cap-store-local c* ∨ *perm-global cv*)
and *offset c* + |*memval-type v*|$_\tau$ > *base c* + *len c*
shows *store h c v = Error* (*C2Err LengthViolation*)
unfolding *store-def*
using *assms*
by (*clarsimp split*: *ccval.split*)

lemma *store-bound-under*:
assumes *tag c = True*
and *perm-store c = True*
and $\bigwedge$ *cv*. ⟦ *v = Cap-v cv*; *tag cv* ⟧ ⟹ *perm-cap-store c* ∧ (*perm-cap-store-local c* ∨ *perm-global cv*)
and *offset c* + |*memval-type v*|$_\tau$ ≤ *base c* + *len c*
and *offset c* < *base c*
shows *store h c v = Error* (*C2Err LengthViolation*)
unfolding *store-def*
using *assms*
by (*clarsimp split*: *ccval.split*)

lemma *store-misaligned*:
assumes *tag c = True*
and *perm-store c = True*
and $\bigwedge$ *cv*. ⟦ *v = Cap-v cv*; *tag cv* ⟧ ⟹ *perm-cap-store c* ∧ (*perm-cap-store-local c* ∨ *perm-global cv*)
and *offset c* + |*memval-type v*|$_\tau$ ≤ *base c* + *len c*
and *offset c* ≥ *base c*
and *offset c mod* |*memval-type v*|$_\tau$ ≠ *0*
shows *store h c v = Error* (*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*. ⟦ *v = Cap-v cv*; *tag cv* ⟧ ⟹ *perm-cap-store c* ∧ (*perm-cap-store-local c* ∨ *perm-global cv*)
and *offset c* + |*memval-type v*|$_\tau$ ≤ *base c* + *len c*
and *offset c* ≥ *base c*
and *offset c mod* |*memval-type v*|$_\tau$ = *0*
and *v = Undef*
shows *store h c v = Error* (*LogicErr* (*Unhandled 0*))
unfolding *store-def*
using *assms*
by *auto*

lemma *store-nonexistant-obj*:
assumes *tag c = True*
and *perm-store c = True*
and $\bigwedge$ *cv*. ⟦ *v = Cap-v cv*; *tag cv* ⟧ ⟹ *perm-cap-store c* ∧ (*perm-cap-store-local c* ∨ *perm-global cv*)
and *offset c* + |*memval-type v*|$_\tau$ ≤ *base c* + *len c*
and *offset c* ≥ *base c*
and *offset c mod* |*memval-type v*|$_\tau$ = *0*
and *v* ≠ *Undef*
and *Mapping.lookup* (*heap-map h*) (*block-id c*) = *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*. ⟦ *v = Cap-v cv*; *tag cv* ⟧ ⟹ *perm-cap-store c* ∧ (*perm-cap-store-local c* ∨ *perm-global cv*)
and *offset c* + |*memval-type v*|$_\tau$ ≤ *base c* + *len c*
and *offset c* ≥ *base c*
and *offset c mod* |*memval-type v*|$_\tau$ = *0*
and *v* ≠ *Undef*
and *Mapping.lookup* (*heap-map h*) (*block-id c*) = *Some Freed*
shows *store h c v = Error* (*LogicErr UseAfterFree*)
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*. ⟦ *v = Cap-v cv*; *tag cv* ⟧ ⟹ *perm-cap-store c* ∧ (*perm-cap-store-local c* ∨ *perm-global cv*)
and *offset c* + |*memval-type v*|$_\tau$ ≤ *base c* + *len c*
and *offset c* ≥ *base c*
and *offset c mod* |*memval-type v*|$_\tau$ = *0*
and *v* ≠ *Undef*
and *Mapping.lookup* (*heap-map h*) (*block-id c*) = *Some* (*Map m*)
and *offset c* < *fst* (*bounds m*)
shows *store h c v = Error* (*LogicErr BufferOverrun*)
unfolding *store-def* **using** *assms*
by (*clarsimp split*: *ccval.split*)

lemma *store-bound-violated-2*:
assumes *tag c = True*
and *perm-store c = True*
and $\bigwedge$ *cv*. ⟦ *v = Cap-v cv*; *tag cv* ⟧ ⟹ *perm-cap-store c* ∧ (*perm-cap-store-local*

$c \vee$ *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* (*Map m*)
and *offset c* + |*memval-type v*|$_\tau >$ *snd* (*bounds m*)
shows *store h c v* = *Error* (*LogicErr BufferOverrun*)
unfolding *store-def* **using** *assms*
by (*clarsimp split*: *ccval.split*)

lemma *store-success*:
assumes *tag c* = *True*
and *perm-store c* = *True*
and $\bigwedge$ *cv*. ⟦ *v* = *Cap-v cv*; *tag cv* ⟧ $\Longrightarrow$ *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* (*Map m*)
and *offset c* $\geq$ *fst* (*bounds m*)
and *offset c* + |*memval-type v*|$_\tau \leq$ *snd* (*bounds m*)
shows $\exists$ *ret*. *store h c v* = *Success ret* $\wedge$
next-block ret = *next-block h* $\wedge$
heap-map ret = *Mapping.update* (*block-id c*) (*Map* (*store-tval m* (*nat*
(*offset c*)) *v*)) (*heap-map h*)
unfolding *store-def*
using *assms*
by (*clarsimp split*: *ccval.split*)

lemma *store-cond-hard-cap*:
assumes *store h c v* = *Success ret*
shows *tag c* = *True*
and *perm-store c* = *True*
and $\bigwedge$ *cv*. ⟦ *v* = *Cap-v cv*; *tag cv* ⟧ $\Longrightarrow$ *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*
proof −
show *tag c* = *True*
using *assms* **unfolding** *store-def*
by (*meson result.simps*(*4*))
next
show *perm-store c* = *True*
using *assms* **unfolding** *store-def*
by (*meson result.simps*(*4*))
next

show $\bigwedge cv.\ [\![\ v = Cap\text{-}v\ cv;\ tag\ cv\ ]\!] \Longrightarrow perm\text{-}cap\text{-}store\ c \wedge (perm\text{-}cap\text{-}store\text{-}local\ c \vee perm\text{-}global\ cv)$
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 $offset\ c + |memval\text{-}type\ v|_\tau \leq base\ c + len\ c$
using *assms* **unfolding** *store-def*
by (*meson linorder-not-le result.simps*(*4*))
next
show $offset\ c \geq base\ c$
using *assms* **unfolding** *store-def*
by (*meson linorder-not-le result.simps*(*4*))
next
show $offset\ c\ mod\ |memval\text{-}type\ v|_\tau = 0$
using *assms* **unfolding** *store-def*
by (*meson linorder-not-le result.simps*(*4*))
qed

lemma *store-cond-bytes-bounds*:
assumes *store h c val* = *Success h′*
and ∀ *x*. *val* ≠ *Cap-v x*
shows ∃ *m*. *Mapping.lookup* (*heap-map h*) (*block-id c*) = *Some* (*Map m*)
∧ *offset c* ≥ *fst* (*bounds m*)
∧ $offset\ c + |memval\text{-}type\ val|_\tau \leq snd\ (bounds\ m)$
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 linorder-not-le result.distinct*(*1*))+

lemma *store-cond-bytes*:
assumes *store h c val* = *Success h′*
and ∀ *x*. *val* ≠ *Cap-v x*
shows ∃ *m*. *Mapping.lookup* (*heap-map h′*) (*block-id c*) = *Some* (*Map m*)
∧ *offset c* ≥ *fst* (*bounds m*)
∧ $offset\ c + |memval\text{-}type\ val|_\tau \leq snd\ (bounds\ m)$
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*)
(*auto split*: *if-split-asm simp add*: *store-tval-def*)

lemma *store-cond-cap-bounds*:
assumes *store h c val* = *Success h′*
and *val* = *Cap-v x*

shows $\exists\ m.\ Mapping.lookup\ (heap\text{-}map\ h)\ (block\text{-}id\ c) = Some\ (Map\ m)$
$\wedge\ offset\ c \geq fst\ (bounds\ m)$
$\wedge\ offset\ c + |memval\text{-}type\ val|_\tau \leq snd\ (bounds\ m)$
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*=*x*, *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
apply (*simp split*: *ccval.split*)
apply (*unfold store-def*)
apply *clarsimp*
apply (*subgoal-tac* $\neg(\neg\ perm\text{-}cap\text{-}store\ c \wedge tag\ x) \wedge \neg(\neg\ perm\text{-}cap\text{-}store\text{-}local\ c \wedge tag\ x \wedge \neg\ perm\text{-}global\ x)$; *blast?*)
apply *clarsimp*
apply (*simp split*: *option.split-asm t.split-asm*)
apply (*metis linorder-not-le result.distinct*(*1*))
done

lemma *store-cond-cap*:
assumes *store h c val* = *Success h'*
and *val* = *Cap-v v*
shows $\exists\ m.\ Mapping.lookup\ (heap\text{-}map\ h')\ (block\text{-}id\ c) = Some\ (Map\ m)$
$\wedge\ offset\ c \geq fst\ (bounds\ m)$
$\wedge\ offset\ c + |memval\text{-}type\ val|_\tau \leq snd\ (bounds\ m)$
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
apply (*simp split*: *ccval.split*)
apply (*unfold store-def*)
apply *clarsimp*
apply (*subgoal-tac* $\neg(\neg\ perm\text{-}cap\text{-}store\ c \wedge tag\ v) \wedge$

```
                ¬(¬ perm-cap-store-local c ∧ tag v ∧ ¬ perm-global v); blast?)
  apply clarsimp
  apply (simp split: option.split-asm t.split-asm)
  apply (case-tac int |Cap|_τ * q < int (fst (bounds x2a)) ∨
                  int (snd (bounds x2a)) < int |Cap|_τ * q + int |Cap|_τ; force?)
  apply (simp add: store-tval-def, force)
  done

lemma store-cond:
  assumes store h c val = Success h'
  shows ∃ m. Mapping.lookup (heap-map h') (block-id c) = Some (Map m)
        ∧ offset c ≥ fst (bounds m)
        ∧ offset c + |memval-type val|_τ ≤ 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)] unfolding store-def
  apply (simp split: ccval.split-asm)
        prefer 9
        apply (subgoal-tac ¬(¬ perm-cap-store c ∧ tag x9) ∧
                  ¬(¬ perm-cap-store-local c ∧ tag x9 ∧ ¬ perm-global x9); blast?)
        apply (simp split: if-split-asm add: store-tval-def, (auto)[1], (auto)[1])+
  done

lemma store-bytes-domain-1:
  assumes x + length vs ≤ n
  shows Mapping.lookup (store-bytes m n vs) x = Mapping.lookup m x
  using assms
  by (induct vs arbitrary: x m n) simp-all

lemma store-bytes-domain-2:
  assumes n + length vs ≤ x
  shows Mapping.lookup (store-bytes m n vs) x = Mapping.lookup m x
  using assms
  by (induct vs arbitrary: x m n) simp-all

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

Set.filter (λ *x*. *n* + *length vs* ≤ *x*) (*Mapping.keys* (*store-bytes m n vs*))
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* ≤ *p*
shows *Mapping.lookup* (*store-cap m p c n*) *x* = *Mapping.lookup m x*
using *assms*
by (*induct n arbitrary*: *x m p*) *simp-all*

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

lemma *store-cap-keys-1*:
Set.filter (λ *x*. *x* + *n* ≤ *p*) (*Mapping.keys m*) =
Set.filter (λ *x*. *x* + *n* ≤ *p*) (*Mapping.keys* (*store-cap m p c n*))
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*:
Set.filter (λ *x*. *p* + *n* ≤ *x*) (*Mapping.keys m*) =
Set.filter (λ *x*. *p* + *n* ≤ *x*) (*Mapping.keys* (*store-cap m p c n*))
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 (λ *x*. *x* < *n*) (*Mapping.keys m*) =
Set.filter (λ *x*. *x* < *n*) (*Mapping.keys* (*store-tag m n b*))
by *fastforce*

lemma *store-tags-keys-2*:
Set.filter (λ *x*. *n* < *x*) (*Mapping.keys m*) =
Set.filter (λ *x*. *n* < *x*) (*Mapping.keys* (*store-tag m n b*))
by *fastforce*

lemma *cap-offset-aligned*:

```
  (cap-offset n) mod |Cap|_τ = 0
  unfolding sizeof-def
  by force

lemma store-tags-offset:
  assumes Set.filter (λ x. x mod |Cap|_τ ≠ 0) (Mapping.keys m) = {}
  shows Set.filter (λ x. x mod |Cap|_τ ≠ 0) (Mapping.keys (store-tag m (cap-offset
n) b)) = {}
  using assms
  unfolding sizeof-def
  by force

lemma store-wellformed:
  assumes 𝒲_𝔣(heap-map h)
    and store h c v = Success h'
  shows 𝒲_𝔣(heap-map h')
proof (cases v)
  case (Uint8-v x1)
  then show ?thesis
    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 split: option.split-asm t.split-asm if-split-asm)
     apply (simp add: wellformed-def)
     apply safe
     apply clarsimp
     apply (erule-tac x=block-id c in allE)
     apply (erule-tac x=x2a in allE) apply clarsimp
      apply (subgoal-tac Set.filter (λx. 0 < x mod |Cap|_τ) (Mapping.keys (tags
(store-tval x2a (nat (int |Uint8|_τ * q)) (Uint8-v x1))))) = {})
      apply (smt (verit, best) Mapping.lookup-update Mapping.lookup-update-neq
Set.member-filter
        assms(1) empty-iff of-nat-less-iff option.sel semiring-1-class.of-nat-0 t.sel
wellformed-def)
     apply (simp add: store-tval-def)
     apply safe
     apply fastforce
     apply fastforce
     done
next
  case (Sint8-v x2)
```

then show *?thesis*
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 split*: *option.split-asm t.split-asm if-split-asm*)
apply (*simp add*: *wellformed-def*)
apply *safe*
apply *clarsimp*
apply (*erule-tac x=block-id c* **in** *allE*)
apply (*erule-tac x=x2a* **in** *allE*) **apply** *clarsimp*
apply (*subgoal-tac Set.filter* ($\lambda x.\ 0 < x\ mod\ |Cap|_\tau$) (*Mapping.keys* (*tags* (*store-tval x2a* (*nat* (*int* $|Sint8|_\tau * q$)) (*Sint8-v x2*)))) = {})
apply (*metis* (*mono-tags*, *lifting*) *Mapping.lookup-update Mapping.lookup-update-neq Set.member-filter*
assms(1) empty-iff less-numeral-extra(3) option.sel t.sel wellformed-def)
apply (*simp add*: *store-tval-def*)
apply *safe*
apply *fastforce*
apply *fastforce*
done
next
case (*Uint16-v x3*)
then show *?thesis*
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 split*: *option.split-asm t.split-asm if-split-asm*)
apply (*simp add*: *wellformed-def*)
apply *safe*
apply *clarsimp*
apply (*erule-tac x=block-id c* **in** *allE*)
apply (*erule-tac x=x2a* **in** *allE*) **apply** *clarsimp*
apply (*subgoal-tac Set.filter* ($\lambda x.\ 0 < x\ mod\ |Cap|_\tau$) (*Mapping.keys* (*tags*

(*store-tval x2a* (*nat* (*int* $|Uint16|_\tau$ ∗ *q*)) (*Uint16-v x3*)))) = {})
apply (*metis* (*mono-tags*, *lifting*) *Mapping.lookup-update Mapping.lookup-update-neq Set.member-filter*
assms(*1*) *empty-iff less-numeral-extra*(*3*) *option.sel t.sel wellformed-def*)
apply (*simp add*: *store-tval-def*)
apply *safe*
apply *fastforce*
apply *fastforce*
done
next
case (*Sint16-v x4*)
then show *?thesis*
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 split*: *option.split-asm t.split-asm if-split-asm*)
apply (*simp add*: *wellformed-def*)
apply *safe*
apply *clarsimp*
apply (*erule-tac x*=*block-id c* **in** *allE*)
apply (*erule-tac x*=*x2a* **in** *allE*) **apply** *clarsimp*
apply (*subgoal-tac Set.filter* (λ*x*. *0* < *x mod* $|Cap|_\tau$) (*Mapping.keys* (*tags* (*store-tval x2a* (*nat* (*int* $|Sint16|_\tau$ ∗ *q*)) (*Sint16-v x4*)))) = {})
apply (*metis* (*mono-tags*, *lifting*) *Mapping.lookup-update Mapping.lookup-update-neq Set.member-filter*
assms(*1*) *empty-iff less-numeral-extra*(*3*) *option.sel t.sel wellformed-def*)
apply (*simp add*: *store-tval-def*)
apply *safe*
apply *fastforce*
apply *fastforce*
done
next
case (*Uint32-v x5*)
then show *?thesis*
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 split*: *option.split-asm t.split-asm if-split-asm*)
apply (*simp add*: *wellformed-def*)
apply *safe*
apply *clarsimp*
apply (*erule-tac x*=*block-id c* **in** *allE*)
apply (*erule-tac x*=*x2a* **in** *allE*) **apply** *clarsimp*
apply (*subgoal-tac Set.filter* (λ*x*. *0* < *x mod* |*Cap*|$_\tau$) (*Mapping.keys* (*tags* (*store-tval x2a* (*nat* (*int* |*Uint32*|$_\tau$ ∗ *q*)) (*Uint32-v x5*)))) = {})
apply (*metis* (*mono-tags*, *lifting*) *Mapping.lookup-update Mapping.lookup-update-neq Set.member-filter*
assms(*1*) *empty-iff less-numeral-extra*(*3*) *option.sel t.sel wellformed-def*)
apply (*simp add*: *store-tval-def*)
apply *safe*
apply *fastforce*
apply *fastforce*
done
next
case (*Sint32-v x6*)
then show *?thesis*
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 split*: *option.split-asm t.split-asm if-split-asm*)
apply (*simp add*: *wellformed-def*)
apply *safe*
apply *clarsimp*
apply (*erule-tac x*=*block-id c* **in** *allE*)
apply (*erule-tac x*=*x2a* **in** *allE*) **apply** *clarsimp*
apply (*subgoal-tac Set.filter* (λ*x*. *0* < *x mod* |*Cap*|$_\tau$) (*Mapping.keys* (*tags* (*store-tval x2a* (*nat* (*int* |*Sint32*|$_\tau$ ∗ *q*)) (*Sint32-v x6*)))) = {})
apply (*metis* (*mono-tags*, *lifting*) *Mapping.lookup-update Mapping.lookup-update-neq Set.member-filter*
assms(*1*) *empty-iff less-numeral-extra*(*3*) *option.sel t.sel wellformed-def*)
apply (*simp add*: *store-tval-def*)
apply *safe*
apply *fastforce*
apply *fastforce*

done
next
case (*Uint64-v x7*)
then show *?thesis*
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 split*: *option.split-asm t.split-asm if-split-asm*)
apply (*simp add*: *wellformed-def*)
apply *safe*
apply *clarsimp*
apply (*erule-tac x=block-id c* **in** *allE*)
apply (*erule-tac x=x2a* **in** *allE*) **apply** *clarsimp*
apply (*subgoal-tac Set.filter* ($\lambda x.\ 0 < x\ mod\ |Cap|_\tau$) (*Mapping.keys* (*tags* (*store-tval x2a* (*nat* (*int* $|Uint64|_\tau * q$)) (*Uint64-v x7*)))) = {})
apply (*metis* (*mono-tags*, *lifting*) *Mapping.lookup-update Mapping.lookup-update-neq Set.member-filter*
assms(1) empty-iff less-numeral-extra(3) option.sel t.sel wellformed-def)
apply (*simp add*: *store-tval-def*)
apply *safe*
apply *fastforce*
apply *fastforce*
done
next
case (*Sint64-v x8*)
then show *?thesis*
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 split*: *option.split-asm t.split-asm if-split-asm*)
apply (*simp add*: *wellformed-def*)
apply *safe*
apply *clarsimp*

apply (*erule-tac x=block-id c* **in** *allE*)
apply (*erule-tac x=x2a* **in** *allE*) **apply** *clarsimp*
apply (*subgoal-tac Set.filter* (λ*x*. *0* < *x mod* |*Cap*|$_\tau$) (*Mapping.keys* (*tags* (*store-tval x2a* (*nat* (*int* |*Sint64*|$_\tau$ ∗ *q*)) (*Sint64-v x8*)))) = {})
apply (*metis* (*mono-tags*, *lifting*) *Mapping.lookup-update Mapping.lookup-update-neq Set.member-filter*
assms(*1*) *empty-iff less-numeral-extra*(*3*) *option.sel t.sel wellformed-def*)
apply (*simp add*: *store-tval-def*)
apply *safe*
apply *fastforce*
apply *fastforce*
done
next
case (*Cap-v x9*)
then show *?thesis*
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(*3*)[**where** *?h=h* **and** *?c=c* **and** *?v=v* **and** *?ret=h'* **and** *?cv=x9*, *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 split*: *ccval.split-asm*)
apply (*unfold store-def*)
apply *clarsimp*
apply (*subgoal-tac* ¬(¬ *perm-cap-store c* ∧ *tag x9*) ∧
¬(¬ *perm-cap-store-local c* ∧ *tag x9* ∧ ¬ *perm-global x9*); *blast?*)
apply (*clarsimp simp add*: *wellformed-def split*: *option.split-asm t.split-asm if-split-asm*)
apply (*erule-tac x=block-id c* **in** *allE*)
apply (*erule-tac x=x2a* **in** *allE*) **apply** *clarsimp* **apply** *safe*
apply (*subgoal-tac Set.filter* (λ*x*. *0* < *x mod* |*Cap*|$_\tau$) (*Mapping.keys* (*tags* (*store-tval x2a* (*nat* (*int* |*Cap*|$_\tau$ ∗ *q*)) (*Cap-v x9*)))) = {})
apply (*smt* (*verit*) *Mapping.lookup-update Mapping.lookup-update-neq Set.member-filter assms*(*1*)
neq0-conv option.sel t.sel wellformed-def)
apply (*simp add*: *store-tval-def*)
apply *safe*
apply *fastforce*
apply *fastforce*
apply *clarsimp*
apply (*subgoal-tac Set.filter* (λ*x*. *0* < *x mod* |*Cap*|$_\tau$) (*Mapping.keys* (*tags* (*store-tval x2a* (*nat* (*int* |*Cap*|$_\tau$ ∗ *q*)) (*Cap-v x9*)))) = {})
apply (*metis* (*mono-tags*, *lifting*) *Set.member-filter assms*(*1*) *empty-iff le-eq-less-or-eq*

linorder-not-le lookup-update′ option.sel t.sel wellformed-def)
apply (*simp add*: *store-tval-def*)
apply *safe*
apply *fastforce*
apply *fastforce*
done
next
case (*Cap-v-frag x101 x102*)
then show *?thesis*
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 split*: *option.split-asm t.split-asm if-split-asm*)
apply (*simp add*: *wellformed-def*)
apply *safe*
apply *clarsimp*
apply (*erule-tac x*=*block-id c* **in** *allE*)
apply (*erule-tac x*=*x2a* **in** *allE*) **apply** *clarsimp*
apply (*subgoal-tac Set.filter* (λ*x*. *0* < *x mod* |*Cap*|$_\tau$) (*Mapping.keys* (*tags* (*store-tval x2a* (*nat* (*int* |*Uint8*|$_\tau$ ∗ *q*)) (*Cap-v-frag x101 x102*)))) = {})
apply (*smt* (*verit*, *best*) *Mapping.lookup-update Mapping.lookup-update-neq Set.member-filter*
assms(*1*) *empty-iff of-nat-less-iff option.sel semiring-1-class.of-nat-0 t.sel wellformed-def*)
apply (*simp add*: *store-tval-def*)
apply *safe*
apply *fastforce*
apply *fastforce*
done
next
case *Undef*
then show *?thesis*
using *assms*(*2*) *store-cond-bytes*(*1*)
by (*metis ccval.distinct*(*107*) *result.distinct*(*1*) *store-cond-hard-cap*(*1*) *store-cond-hard-cap*(*2*)
store-cond-hard-cap(*4*) *store-cond-hard-cap*(*5*) *store-cond-hard-cap*(*6*) *store-undef-val*)
qed

lemma *store-cond-cap-frag*:
assumes *store h c val* = *Success h′*
and *val* = *Cap-v-frag v n*

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

lemma *load-after-store-prim*:
assumes *store h c val = Success h'*
and $\forall\ v.\ val \neq Cap\text{-}v\ v$
and $\forall\ v\ n.\ val \neq Cap\text{-}v\text{-}frag\ v\ n$
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 t.split-asm ccval.split*)
(*safe*; *clarsimp simp add*: *assms*)

lemma *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 (*unfold load-def*; *clarsimp split*: *option.split*)
apply (*simp split*: *t.split*, *safe*)
apply ((*unfold sizeof-def*, *simp*)[*1*])+
apply (*blast dest*: *store-cond-cap*)
apply (*metis option.sel store-cond-cap t.distinct*(*1*))
apply (*unfold sizeof-def*, *simp*)[*1*]
apply ((*unfold store-def*)[*1*], *clarsimp*)
apply (*subgoal-tac* ¬(¬ *perm-cap-store c* ∧ *tag v*) ∧
¬(¬ *perm-cap-store-local c* ∧ *tag v* ∧ ¬ *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*) ‹*offset c* + *int* |*memval-type* (*Cap-v v*)|$_\tau$ ≤ *int* (*base c* + *len c*)›
‹*offset c mod int* |*memval-type* (*Cap-v v*)|$_\tau$ = *0*› *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*)
apply (*unfold sizeof-def*, *simp*)[*1*]
apply ((*unfold store-def*)[*1*], *clarsimp*)
apply (*subgoal-tac* ¬(¬ *perm-cap-store c* ∧ *tag v*) ∧
¬(¬ *perm-cap-store-local c* ∧ *tag v* ∧ ¬ *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*)
apply (*simp add*: *store-def*)
apply (*subgoal-tac* ¬(¬ *perm-cap-store c* ∧ *tag v*) ∧
¬(¬ *perm-cap-store-local c* ∧ *tag v* ∧ ¬ *perm-global v*); *presburger?*)
apply (*clarsimp split*: *if-split-asm option.split-asm t.split-asm*)
apply (*cases perm-cap-load c*; *clarsimp*)
done

lemma *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*)]
using *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*)
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*) *result.simps*(*4*) *store-bounds-preserved*)
apply (*metis assms*(*1*) *result.simps*(*4*) *store-bounds-preserved*)
apply (*simp split*: *if-split-asm*)
apply (*metis Mapping.lookup-update heap.select-convs*(*2*) *heap.surjective heap.update-convs*(*2*)
option.sel t.sel)
done

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 *store-after-alloc*:
assumes *alloc h True s* = *Success* (*h'*, *cap*)
and |*memval-type v*|$_\tau$ ≤ *s*
and *v* ≠ *Undef*
shows ∃ *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 ⋀*cv*. ⟦*v* = *Cap-v cv*; *tag cv*⟧ ⟹ *perm-cap-store cap* ∧ (*perm-cap-store-local cap* ∨ *perm-global cv*)
proof −
have ¬ (*case v of Cap-v cv* ⇒ ¬ *perm-cap-store cap* ∧ *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* ∧ *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* ∧ (*perm-cap-store-local cap* ∨ *perm-global cv*)
by *force*

```
  qed
  moreover have offset cap + |memval-type v|_τ ≤ base cap + len cap
    using assms alloc-def
    by fastforce
  moreover have offset cap ≥ base cap
    using assms alloc-def
    by fastforce
  moreover have offset cap mod |memval-type v|_τ = 0
    using assms alloc-def
    by fastforce
 moreover have Mapping.lookup (heap-map h') (block-id cap) = Some (Map ?m)
    using assms alloc-def
    by fastforce
  moreover have offset cap ≥ fst (bounds ?m)
    using assms alloc-def
    by fastforce
  moreover have offset cap + |memval-type v|_τ ≤ snd (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


definition u8-cast :: block ccval ⇒ block ccval
  where
  u8-cast v ≡
     case v of Uint8-v  v ⇒ Uint8-v v
             | Sint8-v  v ⇒ Uint8-v (UCAST(8 signed → 8) v)
             | Uint16-v v ⇒ Uint8-v (UCAST(16 → 8) v)
             | Sint16-v v ⇒ Uint8-v (UCAST(16 signed → 8) v)
             | Uint32-v v ⇒ Uint8-v (UCAST(32 → 8) v)
             | Sint32-v v ⇒ Uint8-v (UCAST(32 signed → 8) v)
             | Uint64-v v ⇒ Uint8-v (UCAST(64 → 8) v)
             | Sint64-v v ⇒ Uint8-v (UCAST(64 signed → 8) v)
             | Cap-v    v ⇒ Cap-v-frag v 31
             | Cap-v-frag v n ⇒ Cap-v-frag v n
             | Undef ⇒ Undef

lemma u8-cast-size:
  v ≠ Undef ⟹ |memval-type (u8-cast v)|_τ = 1
  by (simp add: sizeof-def u8-cast-def split: ccval.split)

lemma zero-not-undef:
  Uint8-v 0 ≠ Undef
  by simp
```

lemma *zero-size*:
|memval-type (Uint8-v 0)|$_\tau$ = 1
unfolding *sizeof-def*
by *fastforce*

primrec *memset-prim* :: *heap ⇒ cap ⇒ block ccval ⇒ nat ⇒ heap result*
where
memset-prim h - - 0 = Success h
| *memset-prim h c v (Suc n) =*
(let hs = store h c v in
if ¬ is-Success hs then
hs
else
memset-prim (res hs) (c (| offset := offset c + |memval-type v|$_\tau$ |)) v n)

lemma *memset-store-success-step*:
assumes *is-Success (memset-prim h c v (Suc n))*
shows *is-Success (store h c v)*
using *assms*
by *(smt (verit, best) memset-prim.simps(2))*

definition *memset* :: *heap ⇒ cap ⇒ block ccval ⇒ nat ⇒ heap result*
where
memset h c v n ≡ memset-prim h c (u8-cast v) n

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

```
        if xs = Undef then memcpy-prim h dst src (Suc n) else
        let y = store h dst xs in
        if ¬ is-Success y then memcpy-prim h dst src (Suc n) else
        let ys = res y in
       memcpy-cap ys (dst (| offset := (offset dst + |Cap|τ) |)) (src (| offset := (offset
src + |Cap|τ) |)) (Suc n − |Cap|τ))
           apply (metis old.nat.exhaust prod-cases3 sumE)
          apply force
         apply blast
        apply force
       apply force
      apply blast
     apply blast
    apply fast
   apply blast
  apply blast
 apply force
 done

context
 notes sizeof-def[simp]
begin
termination by size-change
end

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 Wf(heap-map h)
 shows memcpy-prim h dst src n = Success h' ⟹ Wf(heap-map h')
   and memcpy-cap h dst src n = Success h' ⟹ Wf(heap-map h')
  using assms
  apply (induct h dst src n and h dst src n rule: memcpy-prim-memcpy-cap.induct)
    apply force
   apply force
    apply (smt (verit, ccfv-SIG) memcpy-prim.simps(2) result.collapse(1) re-
sult.distinct(1) store-wellformed)
  apply (smt (z3) memcpy-cap.simps(2) result.collapse(1) store-wellformed)
  done
```

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

lemma *memcpy-cond*:
assumes *memcpy h dst src n* = *Success h'*
shows *n* > *0* ⟶ ¬ (*block-id dst* = *block-id src* ∧
((*offset src* ≥ *offset dst* ∧ *offset src* < *offset dst* + *n*) ∨
(*offset dst* ≥ *offset src* ∧ *offset dst* < *offset src* + *n*)))
using *assms* **unfolding** *memcpy-def*
by *force*

definition *memmove* :: *heap* ⇒ *cap* ⇒ *cap* ⇒ *nat* ⇒ *heap result*
where
memmove h dst src n ≡
let (*h1*, *tmp*) = *res* (*alloc h True n*) *in*
let h2 = *res* (*memcpy h1 tmp src n*) *in*
let h3 = *res* (*memcpy h2 dst tmp n*) *in*
let (*h4*, -) = *res* (*free h3 tmp*) *in*
Success h4

primrec *memcmp* :: *heap* ⇒ *cap* ⇒ *cap* ⇒ *nat* ⇒ *bool result*
where
memcmp h s1 s2 0 = *Success True*
| *memcmp h s1 s2* (*Suc n*) = (
let v1 = *load h s1 Uint8 in*
let v2 = *load h s2 Uint8 in*
if ¬ *is-Success v1 then*
Error (*err v1*)
else if ¬ *is-Success v2 then*
Error (*err v2*)
else if v1 = *Success Undef* ∨ *v2* = *Success Undef then*
Error (*LogicErr WrongMemVal*)
else if v1 ≠ *v2 then*
Success False
else memcmp h s1 s2 n)

definition *malloc* :: *heap* ⇒ *nat* ⇒ (*heap* × *cap*) *result*
where
malloc h n ≡ *alloc h True n*

definition *calloc* :: *heap* ⇒ *nat* ⇒ (*heap* × *cap*) *result*
where
calloc h n ≡
let hres = *res* (*alloc h True n*) *in*
let h'' = *memset* (*fst hres*) (*snd hres*) (*Uint8-v 0*) *n in*

```
    Success (res h'', snd hres)

definition realloc :: heap ⇒ cap ⇒ nat ⇒ (heap × cap) result
  where
  realloc h cap n ≡
    if cap = NULL then
      alloc h True n
    else
      let (h1, cap') = res (alloc h True n) in
      let h2 = res (memcpy h1 cap' cap (min n (len cap))) in
      let (h3, -) = res (free h2 cap) in
    Success (h3, cap')


lemma store-after-alloc-gen:
  assumes alloc h True s = Success (h', cap)
    and |memval-type v|_τ ≤ s
    and v ≠ Undef
    and n mod |memval-type v|_τ = 0
    and offset cap + n + |memval-type v|_τ ≤ base cap + len cap
  shows ∃ h''. store h' (cap (| offset := offset cap + n |)) 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 (| offset := offset cap + n |)) = True
    using assms(1) alloc-def
    by fastforce
  moreover have ⋀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)
  proof −
    have ¬ (case v of Cap-v cv ⇒ ¬ perm-cap-store (cap (| offset := 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*|$_\tau$ = *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*|$_\tau$ ≤ *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

lemma *store-is-success*:
is-Success (*store h c v*) = (∃ *h'*. *store h c v* = *Success h'*)
by (*simp add*: *is-Success-def*)

lemma *calloc-never-fails*:
is-Success (*calloc h n*)
by (*simp add*: *is-Success-def calloc-def*) (*metis*)

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

primrec *get-memory-leak-size* :: *heap* ⇒ *nat* ⇒ *nat*
where
get-memory-leak-size - 0 = *0*
| *get-memory-leak-size h* (*Suc n*) = *get-memory-leak-size h n* +
(*case get-block-size h* (*integer-of-nat* (*Suc n*)) *of*
None ⇒ *0*

```
    | Some n ⇒ n)

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

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



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

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

alloc free load store
memset memcpy memmove memcmp malloc calloc realloc

set-glob-var

word8-of-integer word16-of-integer word32-of-integer word64-of-integer
integer-of-word8 integer-of-word16 integer-of-word32 integer-of-word64
sword8-of-integer sword16-of-integer sword32-of-integer sword64-of-integer
integer-of-sword8 integer-of-sword16 integer-of-sword32 integer-of-sword64
integer-of-nat cast-val

C2Err LogicErr
TagViolation PermitLoadViolation PermitStoreViolation PermitStoreCapViolation
PermitStoreLocalCapViolation LengthViolation BadAddressViolation
UseAfterFree BufferOverrun MissingResource WrongMemVal MemoryNotFreed Unhandled
in *OCaml*
file-prefix *CHERI-C-Memory-Model*

end